Geneva Astronomical Data Centre

AstroROOT Containers
User Manual

Reference	:	AstroROOT Containers - UM
Issue	:	1.3
Date	:	14 March 08

Geneva Astronomical Data Centre
Chemin d'Écogia 16
CH–1290 Versoix
Switzerland

http://isdc.unige.ch/index.cgi?Soft+astroroot

Authors and Approvals

AstroROOT	**AstroROOT Containers User Manual**	
`14 March 08`	1.3	AstroROOT Containers UM 1.3

Prepared by : R. Rohlfs

Document Status Sheet

AstroROOT	AstroROOT Containers User Manual	
23 SEP 03	1.0	first release
09 MAR 04	1.0.1	better description of the parameters of the TFRead* functions
27 APR 04	1.1	image to TTree and TTree to FITS table conversion.
04 MAR 05	1.2	as default files are opened as FITS files if they cannot be identified as ROOT or ASRO files
14 MAR 08	1.3	use ISDC_ENV instead of WORK_ENV, new format class for bool columns
14 MAR 2008	Printed	

Contents

1 Introduction

1.1 Scope

This is the user manual of the AstroROOT Containers - software system. It describes the components like a table, or an image, but not all the implemented methods and functions. These are described as usual for ROOT classes in a documentation tree on the web

`http://isdc.unige.ch/Soft/AstroRoot/html/USER_Index.html`

The AstroROOT Containers - software system introduce also a new file format to store astronomical data, called AstroROOT File Format (ASRO). This is described in detail in appendix C ASRO File Format.

1.2 Overview

The AstroROOT Containers is part of the AstroROOT package, which itself is an extension to ROOT (http://root.cern.ch/) for astronomical data analysis. These AstroROOT Containers provide C++ classes to organize the data in tables and images with header information. Tables consist of any number of columns and of (a limited number of) rows. All data in one column have the same data format while different columns in one table can have different data formats. A table is therefore very much like a FITS table. An image is a n - dimensional data array of one data format with header information. It is therefore very much like a FITS image.

These data can be saved in files of three different file formats. First tables and images with their headers can be saved in and can be read from FITS files. Second they can be read from and can be written into ROOT files. Third a new file format is implemented to read and write the data as fast as possible: ASRO.

It is the goal of this software library to provide application an easy to use interface to their data. An application is able to access the data from a file with just one function call. For example, no memory allocation of a data buffer is necessary. This is all done by the data classes themselves. Every value in a table can be accessed with one statement. For example, to get the 5th element of row number 8 of an array column one can write:[1]

```
value = table["colName"][8][5];
```

or to access an element of a 4 dimensional image one can write

```
value = image[5][2][4][2];
```

And there is just one function call needed to create a table / an image, or to read a table / an image from a file or to save a table / an image to a file.

An other goal is to provide fast file access for huge amount of data. This software is based on ROOT which is developed at CERN. ROOT is designed to be able to organize huge amount of data in files. The data rate of experiments in high energy physics is much higher than the data rate of astronomical telescopes. But this software uses already the capability of ROOT to be ready for the future.

[1] table["colName"] will return a base column which has to be casted into a column of a specific data type. Therefore
value = dynamic_cast<TFIntArrCol&>(table["colName"])[8][5]
is the correct statement.

ROOT provides a huge functionality for their data structure TTree, like selections and drawing capabilities. Therefore it is important to convert a table into a ROOT Tree. This is done with a simple function call:

```
TTree * tree = table->MakeTree();
```

Once the tree is created with the MakeTree() method, the ROOT - TreeViewer can be used to select and to display the data.
On the other side an image can be converted into a ROOT histogram (TH2D). Again there exist several graphical tools within ROOT to visualize the data of a ROOT histogram.

Like in FITS the grouping of several data structures is implemented. Using an iterator it is very easy to access the required data structures of a group. Several selection criteria are implemented to access the data from different files in a simple loop. Data which are not selected are not read from the file. This is valid for a whole data structure as well as for columns of a table.

Template files can be used to describe the data structures. With such a template file it is much easier to create a table with several columns than to hard - code this in a program. The software can read FITS template files to create the data structures. But also easier human readable template files are supported.

Header attributes, columns of tables and images are implemented as template classes (not to mix with template files). Therefore these data structures can store every data format. Not only the basic C - data formats but also a user can define its own data format. Than the user can save his data with his user defined data format for example in a column. With such a user defined data format row wise as well as column wise tables are implemented.

1.3 Hierarchical Structure

The whole system of the AstroROOT Containers consist of more than 50 classes, but not all of them are designed to be used directly by an application. The following table shows the most important classes which should be used by an application program and will be described in this user manual:

TFIOElement	TFTable	TFGroup
TFBaseImage	template TFImage	
TFHeader	TFBaseAttr	template TFAttr
TFBaseCol	template TFColumn	template TFArrColumn

Iterators:

TFAttrIter	TFRowIter	TFColIter
TFNullIter	TFGroupIter	TFFileIter

The figure on the next page show the simplified hierarchical structure of the C++ classes of the AstroROOT Containers. Note: not all dependencies and not every inheritance is shown.

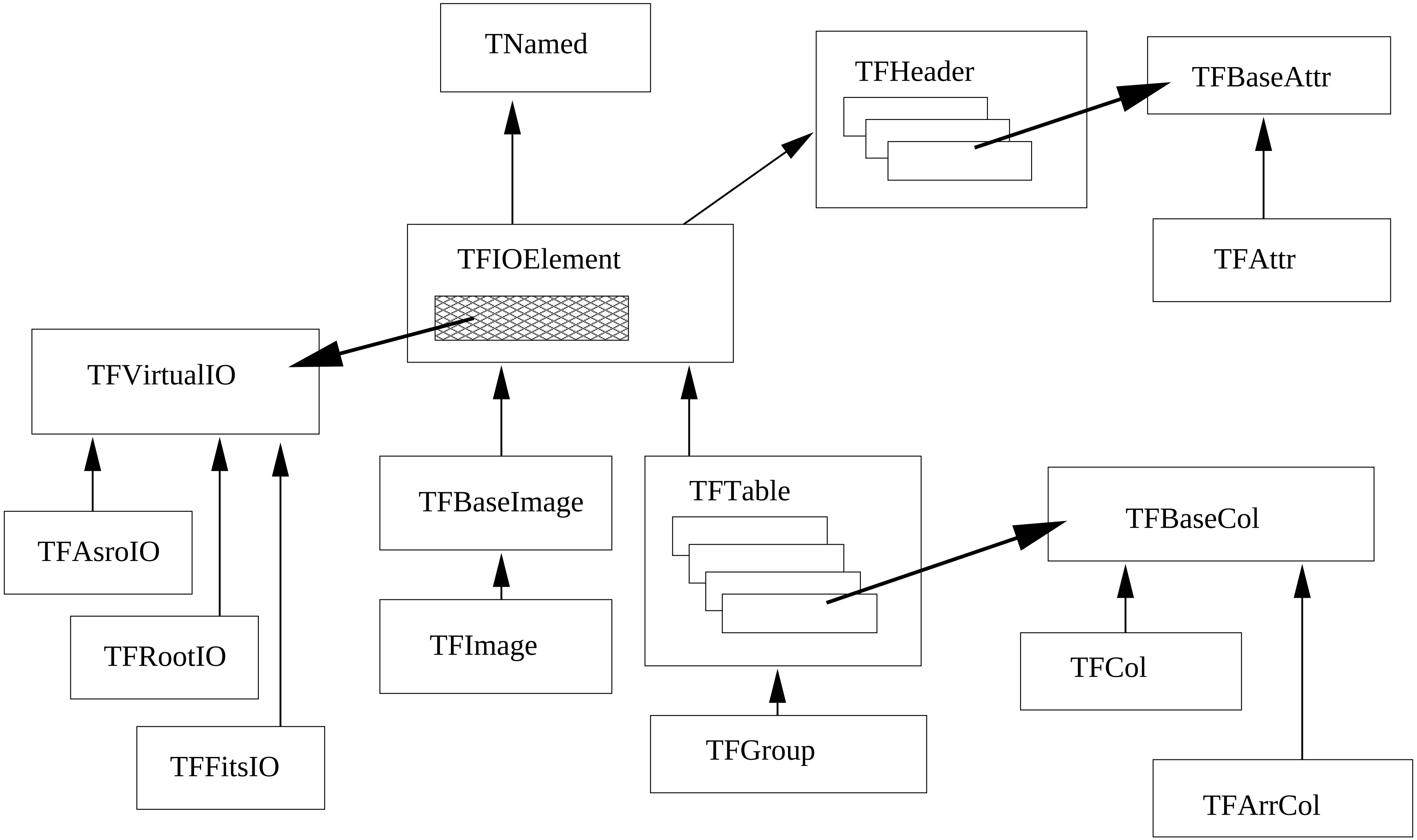


Figure 1: The hierarchical structure of the main classes of the AstroROOT Containers

The class in the center is the **TFIOElement**. It dose not store any data by itself, but this class and its derived classes can be saved in a file. It inherits from the ROOT class TNamed and from the **TFHeader** class to ensure that every container has a name and a header. A container can be created just in memory without any connection to a file, or it can be created while it reads its data from a file. With one exception any update of the data is done only in memory. To save the data of a container a member function (SaveElement()) has to be called. This member function can also be used to save a previously in memory created container into a file.

TFHeader has a list of pointers to the base class of header attributes: **TFBaseAttr**. **TFBaseAttr** stores the name, the unit and the comment of an header attribute. The value of an attribute is stored in one of the derived classes **TFAttr**. **TFAttr** is a template class. Therefore any data type can be stored in an attribute of a header and can be saved in a file.

The table container **TFTable** has a set of pointers to the base class of columns: **TFBaseCol**. **TFBaseCol** stores some general information like the column name, the size of a column and the NULL values. The data itself are stored in one of the derived classes **TFCol** and **TFArrCol**. These derived classes are again template classes to be able to store every data type in a column. It is even possible to store user defined structures and classes in a column. The **TFCol** stores a single value per bin, while **TFArrCol** can store an array of values per bin. Variable bins size is implemented. The functions to add and to remove bins of a columns are private and can be accessed only by the **TFTable**. This ensures that all columns of a table always have the same number of rows.

The grouping is implemented with the **TFGroup** container. It inherits from **TFTable** but does not store any additional data. It just has two methods to attach and to detach a **TFIOElement** or one of its derived classes to the group.

The image container **TFImage** again is implemented as a template class to be able to store any data type. It is derived from **TFBaseImage**, which is derived from TFIOElement. **TFBaseImage** stores common information like the size, the dimension and the NULL pixels of an image, while the template class **TFImage** stores the data of the image. There is no limit in the number of dimensions of the image. Subsections of an image can be defined and one or more dimensions of an image can be freezed. For example it is possible to access any two dimensional plane from a n - dimensional image (n >= 2).

These container classes are independent of any file format. Every read and write operation is performed through the interface definition of the **TFVirtualIO** class. As soon as a Container is associated with a file the **TFIOElement** class keeps a pointer to this interface class. In the current version there are three implementation of this interface: One to read and write FITS files, one to read and write ROOT files and one implementation for the AstroROOT file format ASRO.

2 How to Build

It is assumed that ROOT is already installed and the environment variable ROOTSYS is set!

Download the tar file containing the full AstroROOT package from

```
http://isdc.unige.ch/index.cgi?Soft+astroroot
```

Define two directories. The first for the source code. The second to install the header files, the compiled library and a linked program. The second can be ${ROOTSYS}. This has the advantage that the header files and the library are installed in the already known directories ${ROOTSYS}/include and ${ROOTSYS}/lib, respectively.
But the installation procedure will create some new subdirectories in ${ROOTSYS} and will copy some more files in the include, the lib and the bin subdirectory.

Untar the downloaded tar file astro_root-4.0.tar.gz in the first directory you choose. It can be any directory, but it should be empty!

```
gunzip astro_root-4.0.tar.gz
tar xvf astro_root-4.0.tar
```

Set some environment variables:
ISDC_ENV has to point to the second directory, where you want to install the software.
PFILES defines the directory the installed programs will look for their parameter files. CC and CXX defines the c and c++ compiler, respectively.
You should use the same compilers ROOT was build with.
For example for SUN:

```
setenv ISDC_ENV ${ROOTSYS}
setenv CC cc
setenv CXX CC
setenv CFLAGS -KPIC
setenv CXXFLAGS -KPIC
setenv PFILES ".:${ISDC_ENV}/pfiles:${PFILES}"
```

or for linux:

```
setenv ISDC_ENV /home/astro_root
setenv CC gcc
setenv CXX g++
setenv CFLAGS -fPIC
setenv CXXFLAGS -fPIC
setenv PFILES ".:${ISDC_ENV}/pfiles:${PFILES}"
```

in case ISDC_ENV does NOT point to the same directory than ROOTSYS the environment variables PATH and LD_LIBRARY_PATH have to be updated:

```
setenv PATH ${ISDC_ENV}/bin:${PATH}
setenv LD_LIBRARY_PATH ${ISDC_ENV}/lib:${LD_LIBRARY_PATH}
```

Note: The environment variables ISDC_ENV, PFILES, PATH and LD_LIBRARY_PATH must be set not only during the installation procedure but also at run time of the executables.

Configure the Makefiles:
cd in the directory where you untared the astro_root-4.0.tar.gz file, the first directory. Than type:

```
makefiles/ac_stuff/configure
```

Build the library:
Still in the first directory type:

```
gmake install
```

This will compile all components and will install the necessary files in some subdirectories of ${ISDC_ENV}.

You need GNU make v 3.79.1!! It will not work with SUN's make, for instance.

An other useful make command:

```
make distclean
```

cleans the source code.
After this command the environment variables can be reset and the configure has to issued again!

3 Iterators

For easy data access this AstroROOT Containers package support iterators. There is an iterator for nearly every data class :

Iterator	**created by container**	**"points to"**
TFAttrIter	TFHeader	TFBaseAttr
TFRowIter	TFTable	UInt_t (row number)
TFColIter	TFTable	TFBaseCol
TFNullIter	TFBaseCol	TFNullIndex
TFGroupIter	TFGroup	TFIOElement
TFFileIter		TFIOElement

TFAttrIter returns all attributes of a TFHeader.
TFRowIter returns all row numbers of a TFTable. The rows can be sorted and a filter can be applied.
TFColIter returns all columns of a TFTable.
TFNullIter returns TFNullIndex which gives access to row and cell numbers of a column which are NULL values.
TFGroupIter iterates through all TFIOElements of a TFGroup. It returns never a TFGroup, but iterates again through all TFIOElements if it finds a TFGroup. The rows of the top level TFGroup can be sorted and a filter to the top level TFGroup can be applied. A selection depending on the names of the returned TFIOElements can be applied.
TFFileIter returns all TFIOElement of one disk file.

3.1 Common Functionality

With some exceptions all iterators have the same functions and are created by their container:
The method **MakeIterator()** (or similar name) creates an iterator and returns it. With the function **Next()** the iterator points to the next element. After the first call of **Next()** the iterator points to its first element. Therefore this function has to be called before the first element can be accessed. **Next()** returns kTRUE if there is a further element and kFALSE if the iterator reached its end. If there is no element in the container the first call of **Next()** returns already kFALSE. The iterator can be reset with the function **Reset()**
The iterators overload two operators to return one element: **operator*()** and **operator->()** With these overloaded operators an iterator can be used very much like a C - pointer. Of course these operators must not be used after the **Next()** function returned kFALSE!

The following example prints the names and the data type of all column of a table:

```
TFColIter colIter = table->MakeColIterator();
while (colIter.Next())   {
   printf("Column Name: %-18s  Data type: %-10s\n",
          colIter->GetName(), colIter->GetTypeName());
   }
```

This example prints all row number of a column which are NULL values:

```
TFNullIter nullIter = column->MakeNullIterator();
while (nullIter.Next())   {
   printf("row number with NULL values: %u\n", (Uint_t)(*nullIter));
   }
```

3.2 Exceptions

There are two exceptions of the common functionality described above:

- The iterators TFRowIter do NOT support the **operator->()**. It cannot because it "points" not to a class but to a UInt_t.
- the TFFileIter is not created by a function of its container. Its "container" is a disk file. To create a TFFileIter its constructor has to be used. The first parameter of the constructor is the file name. Than this iterator can "point" to all TFIOElements in this file.

3.3 TFRowIter and TFGroupIter

These two iterators have two further functions to select and to sort the returned rows and TFIOElements, respectively. Actually these function of the TFGroupIter just calls the corresponding function of the TFRowIter of the top level group.

The **Sort(const char * colName)** - function sorts the rows depending on one of the columns of the table. Note: The Sort() - function does not modify the table it only affects the order of row numbers the iterator points to after one call to the Next() - function after the other call.

The **Filter(const char * filter)** applies a filter to filter rows of a TFTable or TFGroup. A row which does not pass the filter is not returned from the operator*() and the operator->().

filter is a c - expression without ; at the end. Column names of the table can be used as variable names in this filter string. They have the same data type as their column. If the column has more than one value per row the first value is used in this filter. Of course, the column names must fulfill the requirement for c - variable names. Beside that "row" can be used to define the row number (0 based). row is the row number of the original table without sorting and without filter. The variable "row_" is the row number (0 based) of the sorted and already filtered row number before the call of this Filter() - function.

For each row in the table the column variables in the filter string are assigned to the value of the columns at the given row and "row" and "row_" in the filter string are set to the row number (0 based). Than the filter statement is processed with the ROOT interpreter. If the result is kTRUE the given row will be returned from the operator*() and the operator->(). If the result is kFALSE the given row will not be returned. A second call of Filter() will not reset the previous filter but will apply the new filter on the already filtered rows.

The function returns kFALSE if the filter cannot be processed. This means either there is a syntax error in the filter string or a used column name in the filter string does not exist in the table. The function will write an error message into the error stack (see TFError).

Example filter strings (assuming c1, c2 and c3 are column names):

```
"c1 + 2.4 * c2 >= c3"
"row > 4 && row < 20"
"row_ < 40 || c2 <= c1"
```

The TFGroupIter has a further function to apply a selection on the TFIOElements it returns: **SetSelector(TFSelector * select)**. An application can define a selection criteria by itself, defining a class derived from the TFSelector class. An often used selection criteria depending on the name of the TFIOElement is already implemented: class TFNameSelector. An example is shown at 10.1 Group Iterators.

4 IOElement

The **TFIOElement** is the base class for all containers which can be stored in a file. For example TFTable and TFImage. But nevertheless every container can be created in memory without any connection to a file.

The TFIOElement class does not store any data by itself, but it is derived from the ROOT class TNamed to ensure that every container has a name, and it is derived from TFHeader to be able to store header information to every container.

4.1 Create a new Container Versus Open a Container

To create a new container either with or without a connection to a new container in a file one of the constructors of the container must be used. If the specified file already exist a new container, for example a TFTable or TFImage, will be created in this file. The file will be created if the file does not exist. The user must be able to read and to write to the specified file.
If a fileName is defined in the constructor the extension of the file defines the file type and the format of the data in the file:
.fits, *.fts, *fit and *.fits.gz defines a FITS file.
.root defines a ROOT file
.asro and anything else defines an ASRO file .

To open an already existing container in a file one of the TFRead*() - functions must be used:

```
TFTable * table = TFReadTable("fileName.fits",  "tableName",  cycleNr,
                              kFReadWrite);
TFGrooup * group = TFReadGroup("fileName.root", "groupName", cycleNr,
                               kFRead);
TFDoubleImg * img = TFReadImage("fileName.asro", "imageName", cycleNr,
                                kFReadWrite);
```

These functions return a NULL pointer or throws an exception (see 12 Error Handling and Exceptions) if the required container does not exist in the file or if the container is of different data type. Therefore it is always a good idea to test the returned value against NULL!

There are two ways to read a container in case the data-type and / or the name of the container is not known:
1) Use the FileIterator **TFFileIter** (see 3 Iterators) or
2) Use the function

```
TFIOElement * element = TFRead("fileName.fits", "name", 0, kFRead, NULL);
```

and test the returned element For example: if (element->IsA() == TFTable::Class())

The calling function must delete the container read by one of the TFRead*() - functions.

4.1.1 Function Parameters

The first parameter defines the filename. It can be either a FITS file, a ROOT file or an AstroRoot (ASRO) file.
The second parameter is the name of the container. For FITS files this parameter can be set to NULL or an empty string ("") if the third parameter is defined. But if it is defined and the third parameter is not set to 0 if must match the extension name of the extension defined by the cycleNr.

This is a mandatory parameter for ROOT files and ASRO files.
The third parameter is the cycle number. If it is set to 0 (the default value) the first container in the file with the specified name is returned. For FITS files this is the HDU number. The first HDU, the primary array, has number 1. In ROOT and ASRO files each element name has its own cycle counter. Therefor cycle number 1 may exist several times in a ROOT or ASRO file. That is the reason why the name (2nd parameter) is mandatory for ROOT and ASRO files.
The fourth parameter can be set either to kFRead or to kFReadWrite. Obviously it has to be set to kFReadWrite if the user want to update the container. The default value is kFRead.

4.2 Common Container - Functions

As a general rule all modifications like changing a value or increasing the number of rows of a table are done only in memory. To save these modification the method **SaveElement()** has to be called without parameters. There is one exception to this rule: A deletion of a column of a table is automatically also performed in the file if the table is associated with a table in a file without calling the **SaveElement()** method.

The **SaveElement(const char * fileName)** method can also be used to store a container in an other file or to save a container which was previously created only in memory without connection to a container in a file. In this case the first parameter fileName has to be defined. As default this parameter is set to NULL.
This function does nothing, also no error, if the container is not associated to a file and the fileName is not defined.

The method **CloseElement()** closes the element in the file, i.e the connection to the element in a file is cut. The Container is not updated before the file is closed. Use **SaveElement()** to update the container in the file before it is closed. The container still exist in memory after this call and still can be used.

The method **DeleteElement()** deletes the associated container in the file but not in memory. The container still exist in memory after this call and still can be used. The file will be deleted if the container was the last one in the file.

Two or more containers in the same file can be opened by one process at the same time and can be updated without problem. It is also possible to open the same container twice in read - only mode. **But to open the same container twice and to update it will fail!**

5 Accepted File Formats

Every AstroROOT container can be written into three different file formats: These are FITS, ROOT and ASRO. It is even possible to mix these file formats in a TFGroup. That means a group can consist of any combination of these file formats. For example a pointer of a TFGroup written into a ROOT file can point to a FITS file.

An application defines the file format with the extension of the filename:

file format	**file extension**
FITS	.fits .fts .fit and .fits.gz
ROOT	.root
ASRO	.asro

If a file with any other extension is created it will be a FITS file. At a open operation of a file with any other extension the containers will try to read the first 4 bytes of a file. If these 4 bytes are "root" it will be opened as a ROOT file if they are "ASRO" it will be opened as and ASRO file and in any other case it will be opened as a FITS file. Note: these first 4 characters in a ROOT and in an ASRO file are automatically written. The user does not have to take care about this at a creation time of a new file.

All three file formats can be read and written from every computer, big endian as well as little endian machines.

5.1 FITS File Format

The AstroROOT library uses cfitsio to read and write FITS files. There are some limitations for FITS files which makes it necessary to modify for example the column names or the attribute names while AstroROOT containers are written into a FITS file. But AstroROOT does its best to write every data into FITS files. It is ensured that data previously read from a FITS file can be updated in the FITS file or be written into a new FITS file without losing data and without changing the names.

5.2 ROOT File Format

AstroROOT supports the fully schema evolution of ROOT. Therefore data written into a ROOT file can be read with every version of AstroROOT (it is upward and downward compatible). These data even can be read when the AstroROOT software is not available any more. All necessary information are stored in the ROOT file.

Columns of a table are stored in their own buffer. Therefore it is possible to read just the required columns without touching the other columns of a table in the ROOT file. This is done automatically. An application program does not have to care.

5.3 ASRO File Format

ROOT files are save in various aspects but have some overhead to store the data. The ROOT file format was designed to store huge amount of data but has some disadvantages if you want to store a lot of small junks of data, for example a lot of small tables with a few rows.

Therefore AstroROOT Containers comes with its own file format: the AstroROOT File Format (ASRO). ASRO still uses the streamer function of ROOT to convert the container classes into a stream of bits and bytes which finally are written into the ASRO file. But ASRO has less overhead in the structure of the file. As a result it is faster than the ROOT format for small data sets.

Appendix C ASRO File Format describes this file format in detail.

6 Headers

The header class **TFHeader** stores a list of attributes. An attribute can be either unique in a header or several attributes with the same name can exist in the same header. While a new attribute is added to a header with the AddAttribute(const TFBaseAttr & attr, Bool_t replace = kTRUE) method the application can define if already existing attributes with the same name should be deleted and be replaced by the new attribute (replace = kTRUE) or if the old attribute can stay in the header (replace = kFALSE).

Not only the **TFIOElement** is derived from **TFHeader** to ensure that every container can store header information but also a column of a table is derived from **TFHeader**. Therefore also columns can be saved into a file with additional header information.

6.1 Header Attributes

One header attribute consist of a name, a value, a unit and a comment. Attributes are implemented as **TFBaseAttr** class and a derived template class **TFAttr**. TFBaseAttr is derived from the ROOT class TNamed. The name and the unit are stored as name and title of the TNamed class, respectively. The value is stored in the template class TFAttr.

In the current version ROOT dictionary information are created for fife different data types (fife TFAttr classes). For a better readable source code also typedef are defined for these data types:

data type of attribute value	**template class**	**typedef**
Bool_t	TFAttr<Bool_t, BoolFormat>	TFBoolAttr
Int_t	TFAttr<Int_t, IntFormat>	TFIntAttr
UInt_t	TFAttr<UInt_t, UIntFormat>	TFUIntAttr
Double_t	TFAttr<Double_t, DoubleFormat>	TFDoubleAttr
TString	TFAttr<TString, StringFormat>	TFStringAttr

To add a new Attribute to a table one can write:

```
table->AddAttribute(TFStringAttr("telescope", "INTEGRAL", "", "mission name"));
table->AddAttribute(TFIntAttr("int time", 3000, "sec", total integration time"));
```

There is no limit in the size of the name and attribute names are case sensitive.

Without knowledge of the data type of the value of an attribute one can always use the GetStringValue() - method and the SetString() - method of the **TFBaseAttr** class to get and to set the value, respectively.

The cast operator is defined for every TFAttr template class. It casts the Attribute class to its value. Therefore it is easy to access for example the value of an TFIntAttr named size:

```
   TFIntAttr & attr = dynamic_cast<TFIntAttr&>(table->GetAttribute("size"));
   Int_t size = attr;
```

or even simpler

```
   Int_t size = dynamic_cast<TFIntAttr&>(table->GetAttribute("size"));
```

6.2 Attribute Iterator

If an application wants to access not only a few attributes but all or at least most of the attributes of a header it is more efficient to use the **TFAttrIter** than to use the GetAttribute() methods of the TFHeader.

Following example prints all attributes of a table with their name, value, unit and comment:

```
TFAttrIter i_attr = table->MakeAttrIterator();
char hstr[500];
while (i_attr.Next())
   {
   printf("    %-16s : %20s %-5s | %s\n",
            i_attr->GetName(),
            i_attr->GetStringValue(hstr),
            i_attr->GetUnit(),
            i_attr->GetComment() );
   }
```

7 Tables

A table is implemented with the **TFTable** class. A **TFTable** can store any number of columns. The maximum number of rows in a table is 4294967295. The column name is the key to find a column in the table and has to be unique. There is no column number. The columns are sorted by name to have faster access to a requested column.

To ensure that all columns of a table have always the same number of rows it is not possible to increase or decrease the row number for a singel column. The InsertRows() and DeleteRows() - method are public methods of the **TFTable** but not of a single column. The size of a column, i. e. the number of rows, is adopted to the number of rows of a table while the column is added to a table (method AddColumn()).

7.1 Columns

There are two template classes for columns: **TFColumn** and **TFArrColumn**. Both are derived from a base column class: **TFBaseCol**. The first stores a single value in a bin while the second can store an array of values per bin. The [ ] - operator are defined to access the data like an array. Assuming col is a column of integers and arrCol is an array column of double one can write following lines to get and to set values into a column:

```
col[4] = 23;
int val = col[0];

arrCol[4][0] = 4.6;
double dVal = arrCol[32][39];
```

Note: the bin as well as the cell number of array columns are 0 - based indexes. As usual for the [ ] - operator there is no limit check. This operator will be used very often and a limit check would decrease the performance a lot. This limit check can be done much more effician by the application. But of course a program can crash and has at least unpredicted results if a row bejond the maximum number of rows is accessed!

In the current version ROOT dictionary information are created for several data types of columns. For a better readable source code also typedef are defined for these data types:

data type of one cell	**template class**	**typedef**
Bool_t	TFColumn<Char_t, BoolCharFormat>	TFBoolCol
Char_t	TFColumn<Char_t, CharFormat>	TFCharCol
UChar_t	TFColumn<UChar_t, UCharFormat>	TFUCharCol
Short_t	TFColumn<Short_t, ShortFormat>	TFShortCol
UShort_t	TFColumn<UShort_t, UShortFormat>	TFUShortCol
Int_t	TFColumn<Int_t, IntFormat>	TFIntCol
UInt_t	TFColumn<UInt_t, UIntFormat>	TFUIntCol
Float_t	TFColumn<Float_t, FloatFormat>	TFFloatCol
Double_t	TFColumn<Double_t, DoubleFormat>	TFDoubleCol
TString	TFStringCol	

The TFStringCol is not a template class, but can be used exactly like the other Column classes.

For array columns following is defined:

data type of one cell	template class	typedef
Bool_t	TFArrColumn<Char_t, BoolCharFormat>	TFBoolArrCol
Char_t	TFArrColumn<Char_t, CharFormat>	TFCharArrCol
UChar_t	TFArrColumn<UChar_t, UCharFormat>	TFUCharArrCol
Short_t	TFArrColumn<Short_t, ShortFormat>	TFShortArrCol
UShort_t	TFArrColumn<UShort_t, UShortFormat>	TFUShortArrCol
Int_t	TFArrColumn<Int_t, IntFormat>	TFIntArrCol
UInt_t	TFArrColumn<UInt_t, UIntFormat>	TFUIntArrCol
Float_t	TFArrColumn<Float_t, FloatFormat>	TFFloatArrCol
Double_t	TFArrColumn<Double_t, DoubleFormat>	TFDoubleArrCol

To add a new column to a table one can write:

```
table->AddColumn(new TFIntCol("colName"));
table->AddColumn("colName2", TFShortArrCol::Class(), kTRUE);
```

Never change the name of a column after it is inserted into a table! For faster access the columns are sorted by their names. The result of several functions of the the TFTable are unpredictable after a name of a column has changed!

To get one column of a table one can use the GetColumn(const char * name) - method or the operator [] (const char * name). Both return a reference to **TFBaseCol**. Some care has to be taken to cast the returned TFBaseCol to the data type of the column. For more information see B Casting of Column Data Types.

If an application does not know the data type of a column it can test the data type as follows:

```
TFBaseCol & col = table["ColName"];
if (col.IsA() == TFShortCol::Class())
   {
   // col is of type TFShortCol
   }
else if (col.IsA() == TFIntCol::Class())
   {
   //  col is of type TFIntCol
   }

 ...
```

7.1.1 NULL - values

Every bin of a column and every cell of an array column can be set to be a NULL value, like in FITS files. But there is not a specific value which is used to identify a cell to be NULL. The bin and cell number of NULL bins and cells themselve are stored in the column. In most cases this is more efficiant, because usually only a few bins are set to NULL.

There are methods of the TFBaseCol - class to set, to clear and to test for NULL and there is a iterator to get all bins and cells which are set to NULL:

```
virtual Bool_t       IsNull(UInt_t row, UInt_t bin = 0) const;
virtual void         SetNull(UInt_t row, UInt_t bin = 0);
virtual void         ClearNull(UInt_t row, UInt_t bin = 0);
```

```
TFNullIter   MakeNullIterator() const;
```

7.1.2 Variable Bin Size

Variable bin size of array columns is implemented. For example to define the number of cells to 18 for bin number 4 one has to call:

```
arrCol[4].resize(18);
```

This call has to be performed instead of the the SetNumBins(UInt_t cells) - method, which would set the number of cells for all bins of a column. In any case the number of cells per bin has to be set before one can use an array column!

7.2 Iterators of Tables and Columns

A **TFTable** has two iterators: **TFColIter** and **TFRowIter** and two methods to create them: MakeColIterator() and MakeRowIterator(), respectively. The **TFColIter** is a standard Astro-ROOT - iterator as described at 3 Iterators. This iterator is very usefull if an application does not know all the column names.

The **TFRowIter** iterates through all rows of a column, which is not very usefull by itself. Every for - loop can do this as well. But this iterator has two further function to sort the rows depending on one column and to filter rows depending on values of any column. See 3.3 TFRowIter and TFGroupIter for a full description of these functions.

Every column can create the Null - iterator **TFNullIter** with the method MakeNullIterator(); To print all bins of a colum and all bins and cells of an array column one can write:

```
TFNullIter nullIter = column->MakeNullIterator();
while (nullIter.Next())   {
   UInt_t row = *nullIter;
   printf("row number with NULL values: %u\n", row);
   }

TFNullIter nullIter2 = arrColumn->MakeNullIterator();
while (nullIter2.Next())   {
   printf("NULL - cell: [%u][%u]\n", nullIter2->Bin(), nullIter2->Cell());
   }
```

8 Images

An image is implemented as template class **TFImage** which is derived from the base image class **TFBaseImage**. An image is a n - dimensional array of data. There is no limit in the number of dimensions. The maximum number of pixels of an image is 4294967295 (if there is no other hardware limitation).

In the current version ROOT dictionary information are created for several data types of images. For a better readable source code also typedef are defined for these data types:

data type of one pixel	**template class**	**typedef**
Bool_t	TFImage<Bool_t, BoolFormat>	TFBoolImg
Char_t	TFImage<Char_t, CharFormat>	TFCharImg
UChar_t	TFImage<UChar_t, UCharFormat>	TFUCharImg
Short_t	TFImage<Short_t, ShortFormat>	TFShortImg
UShort_t	TFImage<UShort_t, UShortFormat>	TFUShortImg
Int_t	TFImage<Int_t, IntFormat>	TFIntImg
UInt_t	TFImage<UInt_t, UIntFormat>	TFUIntImg
Float_t	TFImage<Float_t, FloatFormat>	TFFloatImg
Double_t	TFImage<Double_t, DoubleFormat>	TFDoubleImg

8.1 Image Sub - Section

A sub - section of an image, which can have even less dimensions than the original image can be defined with the MakeSubSection(UInt_t * begin, UInt_t * end) - function. begin defines the first pixel in each dimension of the original image which will be become the index 0 of the sub section. end defines the pixel in each dimension of the original image which will be behind the last pixel of the sub - image. The size of the sub - image in a dimension will be end[dimX] - begin[dimX] The first index (index 0) of begin and end defines the least frequently changing dimension. For example in a 2 - dimensional image the Y - axis.

A dimension can be freezed if begin[dimX] == end[dimX] A pixel of the sub - image must be accessed with the operator () instead of the opertor []. Frozen dimensions must be skipped.

For example:

```
TFIntImg img("name", 10, 5, 20);
UInt_t begin[3] = {3 , 3, 10};
UInt_t end[3]   = {10, 3, 15};
img.MakeSubSection(begin, end);
```

The sub - image of img has 2 dimensions and is of size 7 X 5
img[4][3][12] will now access the same pixel as img(1)(2)

8.2 Access of Image Pixels

A single pixel of an image can be accessed (read and write) like a element of an C - array:

```
TFDoubleImg * img = TFReadImage("filename.root", "imageName");
(*img)[3][4] = 4.56;
double val = (*img)[30][234];
img->SaveElement();
delete img;
```

If a sub - section of an image is defined the operator () instead of the operator [] hast to be used to access a pixel value:

```
TFDoubleImg * img = TFReadImage("filename.root", "imageName");
UInt_t begin[3] = {3 , 3, 10};
UInt_t end[3]   = {10, 3, 15};
img->MakeSubSection(begin, end);
(*img)(3)(4) = 4.56;
double val = (*img)(30)(234);
img->SaveElement();
delete img;
```

Unfortunately the ROOT interpreter rcint does not accept the operator() in the current version 3.05.05.

8.3 Build a ROOT histogram or a ROOT TTree

It is possible with one function call to create a one or a two dimensional ROOT histogram from every TFImage. See 9.3 Images -> Histograms. Beside that a ROOT TTree can be created from a TFBaseImage. See 9.4 Images -> Tree.

9 ROOT Trees (TTree) and ROOT Histograms (TH2D)

9.1 Tables -> Trees

Every TFTable can be converted into a ROOT TTree. The member function **MakeTree()** creates a TTree and returns its pointer. This TTree has to be deleted by the calling function. Every column of the table TFTable is copied into its own branch of the TTree. The header information of the TFTable is not copied into the TTree.

Following example reads the table named "tableName" from the FITS file t.fits, creates a TTree and opens the TTreeViewer of this TTree:

```
TFTable * table = TFReadTable("t.fits", "tableName");
if (table)
   table->MakeTree()->StartViewer();
```

The **MakeTree()** - function has one optional parameter to convert the table name and the names of the columns. The parameter is a pointer to a TFNameConvert class or a derived class. The MakeTree calls the virtual function const char * TFNameConvert::Conv(const char * name) to convert the table name and the name of the columns. The user can create its own class, derived from TFNameConvert, to modify these names. The default is no conversion at all. But it may be convenient to lower case the table and columns names of the FITS files.

The executable tf2tree reads one or all TFTables from a FITS file, a ROOT file or an ASRO file, converts them into a TTree and saves them in a new ROOT file. The program also writes the header information of a TFTable into the new ROOT file as a separate object. This program also shows an example how to define and to use a name conversion class, derived from TFNameConvert.

9.2 Tables-> TGraphErrors

Two columns of one or two tables can be used to build a TGraph or a TGraphErrors. Two further columns can be used to define the errors in X and Y of a TGraphErrors. In the first example two columns of the same table are used to build a TGraph while the second example shows how to build a TGraphErrors from two tables.

```
TFTable * table = TFReadTable("t.fits", "tableName");
TGraph * graph = table->MakeGraph("XColName", "YColName");
graph->Draw("AL");
```

```
TFTable * table1 = TFReadTable("t1.fits", "tableName");
TFTable * table2 = TFReadTable("t2.fits", "tableName");
TGraphErrors * graph = table1->MakeGraph("XColName", NULL, XErrColName");
table2->MakeGraph(NULL, "YColName", NULL, YErrColName, graph);
graph->Draw("AL");
```

The TGraphError returned by the TFTable::MakeGraph() - function must be deleted by the calling function.

The first table always define the number of dots of the TGraph. Therefore it must have at least 2 rows. An error message is written to TFError if the table has less than 2 rows and the function will not create a TGraph but will return NULL. If one of the columns does not exist in the table an error message will be written to TFError and the TGraph is not updated with any value but it will be created anyhow and the values will be set to 0.

9.3 Images -> Histograms

Every image can be converted into a one or a two dimensional ROOT histogram:

```
TFBaseImage * image = TFReadImage("i.fits", "imageName");
if (image)
   TH2D * hist = dynamic_cast<TH2D*>image->MakeHisto();
```

As default a TH2D image is created. But the makeHist() - functions accepts a parameter which define the histogram class which should be created. For example to create a 1 - dimensional histogram of short values one can write:

```
TFBaseImage * image = TFReadImage("i.fits", "imageName");
if (image)
   TH1S * hist = dynamic_cast<TH1S*>image->MakeHisto(TH1S::Class());
```

Only one and tow dimensional histograms are supported If a sub - image is defined (function MakeSubSection()) the histogram will be created using only the sub - image, else the full image is used. If the image has more dimensions than the histogram the least frequently dimensions of the image are used. To create for example a two dimensional histogram of the X - and Y - dimension from a 3 - dimensional image first a sub - section with a freezing third dimension has to be defined. For example:

```
UInt_t begin[3] = {3, 0, 20};
UInt_t end[3]   = {3, 0, 20};
img.MakeSubSection(begin, end);
TH2D * hist = dynamic_cast<TH2D*>(img.MakeHisto());
```

The returned histogram has to be deleted by the calling function.

9.4 Images -> Tree

Every TFBaseImage can be converted into a ROOT TTree. The TTree will have one branch per image dimension as well as one branch for the pixel value. One entry per image pixel which is not NULL will be added to the ROOT TTree.

```
TFBaseImage * image = TFReadImage("i.fits", "imageName");
if (image)
   TTree * tree = image->MakeTree();
```

The returned TTree has to be deleted by the calling program.

Similar to the TFTable::MakeTree() function (see above) also the TFBaseImage::MakeTree has one optional parameter to convert the image name before it is assigned to the TTree name.

The executable tf2tree can read one or all TFBaseImages from a FITS file, a ROOT file or an ASRO file, converts them into a TTree and saves them in a new ROOT file. The program also writes the header information of a TFBaseImage into the new ROOT file as a separate object. This program also shows an example how to define and to use a name conversion class, derived from TFNameConvert.

9.5 TTrees -> Tables

One constructor of the TFTable fills the table with data of a TTree. Only simple branches with a single variable of the TTree are converterd into a column of the new TFTable. The other branches of the TTree are ignored.

```
TFile *   file  = new TFile("trees.root");
TTree *   tree  = file->Get("Exposure;1");
TFTable * table = new TFTable(tree);
table->SaveElement("expo.fits");
delete table;
```

The executable tree2tf can read one or all TTrees from a ROOT file, convert them into TFTables and save them in a FITS file.

10 Grouping Implementation

Grouping, very similar to FITS grouping, is implemented with the **TFGroup** class. It is derived from **TFTable**, i. e. it is a table. The grouping class **TFGroup** has three additional methods: Attach(), Detach() and MakeIterator()

With Attach(TFIOElement * element, Bool_t relativePath = kTRUE) a new TFIOElement or one of its derived classes can be attached to a group. The file name can be stored in the group as absolute path (relativePath = kFALSE) or as relative path, relative to the group (relativePath = kTRUE). Detach(TFIOElement * element) detaches one element from the group. Remember these changes are done only in the TFGroup actually stored in memory. To update the group in a file the SaveElement() has to be called. Finally MakeIterator() creates a group - iterator **TFGroupIter** and returns it. This group - iterator has to be used to get access to the members of a group.

A **TFGroup** stores the information of its direct children in one column called "_GROUP_". Therefore it is important not to add another column with this name to a group table. The column "_GROUP_" is automatically created by the Attach() - method when a first TFIOElement is attached to the group. The application must not create this column.

In the current version all members of a group have to be on the same file system. Access to members using the FTP or HTTP protocol is not implemented.

10.1 Group Iterator

The group Iterator **TFGroupIter**, created with the function TFGroup::MakeItererator() has to be used to access all children of every level of a group. As usual the operator*() and the operator->() return one element the iterator "points" to. In this case a TFIOElement or one of its derived classes. These operators never return a TFGroup. If a group has as child a TFGroup the iterator will also iterate through all children of this child TFGroup and so on.

The **TFGroupIter** has two further functions to select and to sort the returned TFIOElements. But these functions sort and select the TFIOElements only of the first level group. They cannot be applied to second order groups because the sorting and selecting depend on other columns of the group. But children groups properly have other further columns than the top level group. For more information how to sort and to select the TFIOElements of a TFGroupIter see 3.3 TFRowIter and TFGroupIter.

There exist an other method to select the TFIOElements which applies to all TFIOElements returned by the TFGroupIter independent of the group level: SetSelector(TFSelector * select); The iterator will call the TFSelector::Select() method to every TFIOElement and will return only TFIOElements for which this method returns kTRUE. To use this selection method an application has to create its own class derived from TFSelector and has to code the Select() method of its class.

One example is already implemented: **TFNameSelector**. This selector takes as parameter in its constructor a name. This name can have the wildcard *. Its Select() method will return kTRUE if the name of a TFIOElement fits to the name passed to its constructor. For example following code will print the file names of all children which name consist at least of the string "eng":

```
TFGroupIter i_group = group->MakeItererator();
i_group.SetSelector(new TFNameSelector("*eng*"));
while (i_group.Next())
   printf("fileName: %s\n", i_group->GetFileName());
```

It is possible to define more than one Selector for one group iterator. Only TFIOElements which pass all Selectors are returned by the iterator.

11 Template files

Not yet implemented.

12 Error Handling and Exceptions

This library supports exceptions as well as returning an error status. It is easier to check the error status of a function while the classes and functions are used in an interactive session. But it may be more convenient to use exceptions in a compiled program. As default the functions do not throw any exception but will return an error in case of a failure. To switch to exceptions a static function of the **TFError** - class has to be called, typically at the beginning of a program:

```
TFError::SetErrorType();
```

At any time you can switch back to return error codes from a function:

```
TFError::SetErrorType(kStoreErr);
```

After a call of the function TFError::SetErrorType() the functions of this AstrROOT - container classes throw an exception, return an error code or do both:
TFError::SetErrorType(kStoreErr) : Functions will write an error message to the TFError class and return an error code.
TFError::SetErrorType(kExceptionErr) : Functions will throw an TFException - exception. The error message can be read from the exception.
TFError::SetErrorType(kAllErr) : Functions will write an error message to the TFError class and will throw an TFException - exception.

If no error occurred the functions will return 0 or a valid pointer.

At any time the error messages stored in the TFError class can be printed:

```
TFError::PrintErrors();
```

This function will do nothing if there was no error.

Following functions may throw an exception or return an error code:

function	**return value in case of an error**
TFHeader::GetAttribute	a TFBoolAttr with name Error and value false
TFRead	NULL pointer
TFIOElement::TFIOElement	IsFileConnected() return kFALSE
TFFileIter::TFFileIter	IsFileConnected() return kFALSE
TFIOElement::SaveElement	-1
TFIOElement::DeleteElement	-1
Int_t TFTable::AddColumn	-1
TFBaseCol TFTable::AddColumn	reference to a NULL pointer: *((TFBaseCol*)0)
TFBaseCol TFTable::GetColumn	reference to a NULL pointer: *((TFBaseCol*)0)
TFTable::MakeGraph	NULL pointer

Note: the functions TFTable::AddColumn() and TFTable::GetColumn() return a reference of a TFBaseCol. Following c++ standard these functions should in any case throw an exception if they cannot return a valid column. But the developer wanted to avoid this in case these function are used in an interactive session. The user can easily make a typo in the column name and would quit also the root program. Therefore these functions return a non standard value: a reference to a TFBaseCol which is mapped at address NULL (0).

13 Executables

There are some executables coming with the AstroROOT containers.

All executables are like FTOOLS with parameter files. To run the programs the environment variable PFILES must be set to the directory of the parameter files or the parameter files *.par must be in the local directory.
Noty: during installation the parameter files are installed in the directory ${ISDC_ENV}/pfiles. Therefore the easiest is to define

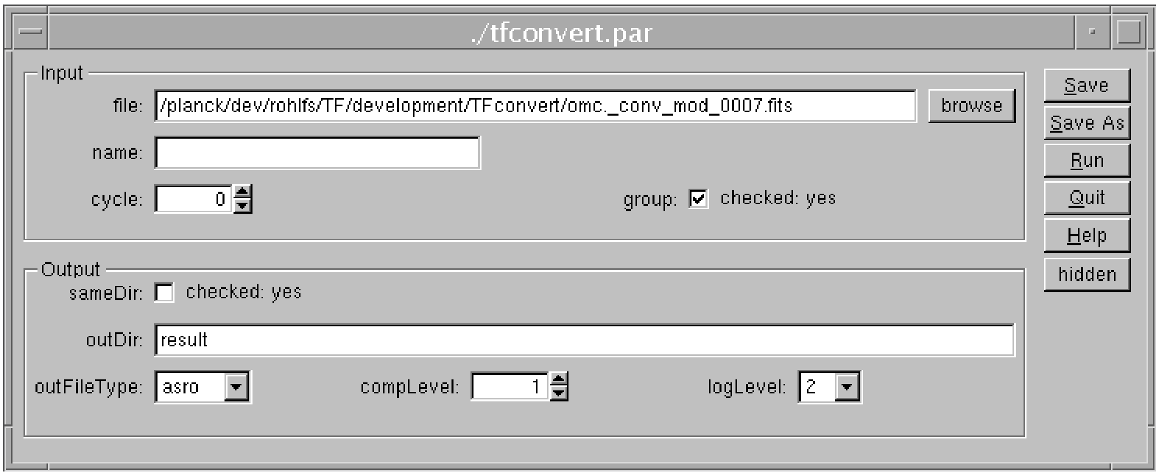
```
setenv PFILES ${ISDC_ENV]/pfiles
```

More information about paraemter files can be found in the PIL user manual ${ISDC_ENV}/help/pil.ps

The GUI to edit the program parameters is launched if the program is called with the flag –g (minus minus g) or –gui (minus minus gui). For example tfconvert –g Without this flag the programs ask for the parameters. More information about the program parameters are in a help text of each program (*.txt). This help text can be displayed with the Help button of the GUI.

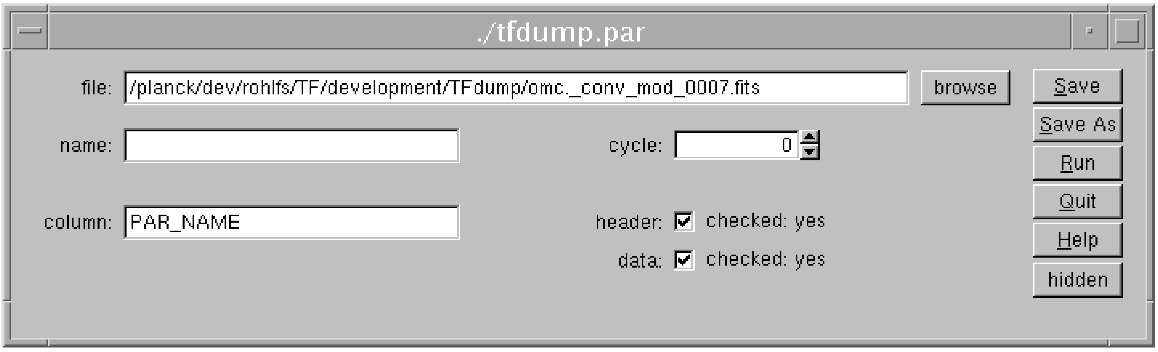
13.1 tfconvert

The tfconvert program can convert one element in a file, all elements in a file or all elements of a group from one file format into an other supported file format (See 5 Accepted File Formats).



13.2 tfdump

tfdump is a simple program to dump either one or all elements of a file to standard output. The input file must be one of the accepted file formats (FITS, ROOT or ASRO).

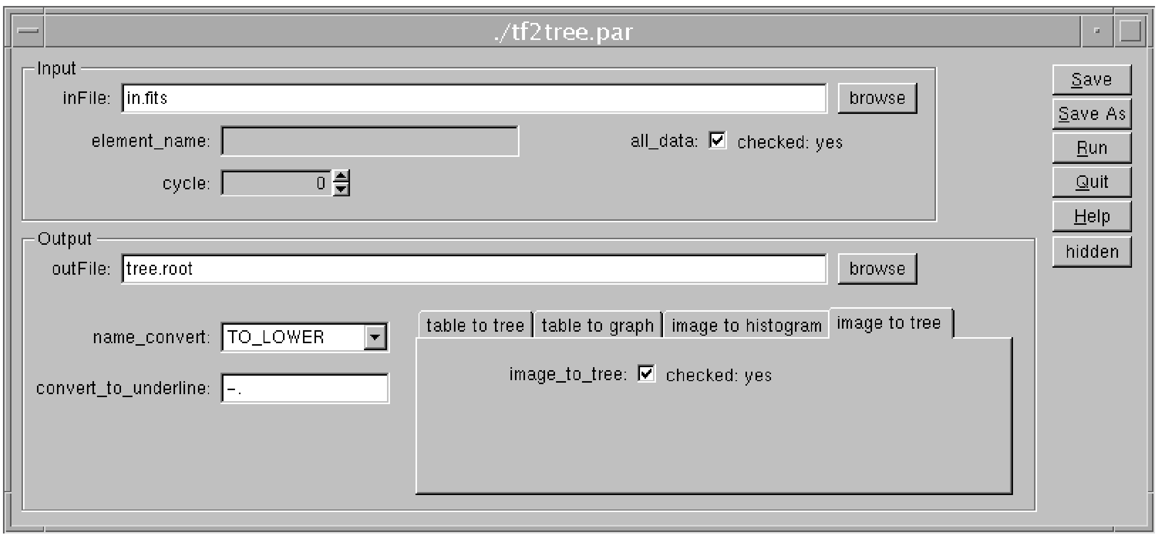


13.3 tfasro_map

tfasro_map is a program to print the name, the size and the position of all Containers in an ASRO file.

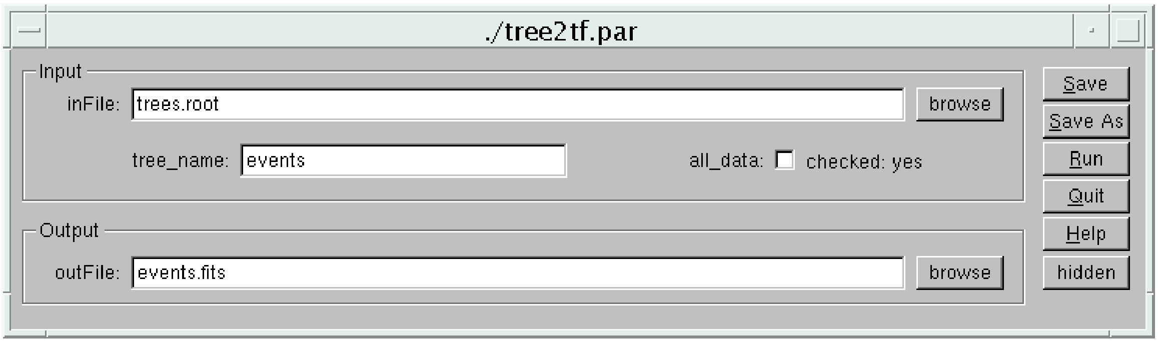
13.4 tf2tree

tf2tree converts one or all tables in a file to ROOT TTree(s) and / or to ROOT TGraph(s). FITS images can be converted into ROOT histograms or ROOT TTrees. The converted data are save together with the header information in a new ROOT file. The input file must be one of the accepted file formats (FITS, ROOT or ASRO).



13.5 tree2tf

tree2tf converts one or all TTrees in a ROOT file to TFTables which are saved in a FITS table, a new ROOT file or an ASRO file. The extension of the output file defines the output format.



14 To Do or Not Yet Implemented

Some functionalities described in this user manual are not yet implemented. These are

- templates files

A Creating a User Defined Column Type

To be written.

B Casting of Column Data Types

Columns of a table themselves as well as a single value in a table column have a C - data type. While all values in a column have the same data type different columns in the same table can have different data type. Therefore the "Get" - functions [2] of the table to retrieve a column can return only a data type independent column. This data type independent column is the **TFBaseCol** - class. All other columns are derived from this base column.

To access a column and its data from a table requires a cast either to the data type of a column or to the data type of a single value of the column. This cast operation is done most of the time either by the compiler or by defined cast - operators. But the ROOT - interpreter sometimes behaves differently than a C++ compiler and in some seldom situations it is better that the application forces a cast. Therefore this chapter described in detail the necessary cast operations.

B.1 Cast to a Column Type

To get a column from a table one can write

```
TFIntCol & col = table["ColName"];
```

As mentioned before table["ColName"] will return a reference to **TFBaseCol**. The TFBaseCol defines cast operators to all by the library known column - data types. Therefore the assignment to a derived column class, like TFIntCol in this example, can be performed.

If an application defines its own column class (see A Creating a User Defined Column Type) the TFBaseCol cannot be automatically casted to this user defined data type of the column. The application has to do it. For example:

```
MyCol & col = dynamic_cast<Mycol&>(table["ColName"]);
```

This is necessary for compiled code. The ROOT interpreter knows all the data types of the columns and is smart enough to cast even to user defined column - data types by itself. Therefore this does work in an interpreter macro:

```
MyCol & col = table["ColName"];
```

All these cast operations cast a reference. If the column is not of the required type the dynamic_cast - operator will throw a bad_cast - exception which should be catched by an application. If this exception is not catched the program, also the root - program, will terminate.

B.2 Cast to a Value Type

To access a single value of a column one can write

```
double val = table["ColName"][3];
table["ColName"][3] = 19;
```

This can be very useful if an application does not know the data type of the column. But in these statements no cast operators to a specific column - data type can be performed because neither a

[2] The "Get" - functions of a table are
TFBaseCol & GetColumn(const char * name) const; and
TFBaseCol & operator[] (const char * name) const;

compiler nor the ROOT interpreter can know to which column data type it has to cast. Therefore table["ColName"] returns a TFBaseCol and the operator[] of the TFBaseCol is executed to access a single value of the column. But this operator[] of the TFBaseCol cannot assign any data type. It can assign only one data type and we choose the double data type. To be more precise: To be able to set a value like in the second line of the example above the operator[] does not directly return a reference to a double but a reference to a **TFSetDbl** class which itself defines a cast operator to double and the assignment operator to double. As a consequence following line does not work in the ROOT interpreter (assuming the column named ColName is a column of integers):

```
int val = table["ColName"][3];
```

but this does work:

```
int val = (double)table["ColName"][3];
```

(Both does work as you woud expect in compiled code).

In any case the value is always casted to a double before it is assigned to a value in the column or while it is read from a column. While the value is finally casted to the data type of the column the library does not simple cast the double value to an integer value but for example all values between 0.5 and 1.499 are assigned to 1.

If the column is of a TFStringCol data type, the operator[] converts the double value to a string while a value is assigned to the row of a column. And the operator[] tries to convert a string to a double value while it reads a value from the column. Therefore the operator [] will return 0.0 if the string in the requested row does not start with a number.

This operator[] of the TFBaseCol - class will do nothing for array - columns. It will return always 0 and will not set any value in the column.

For several reasons the cast to a column data type is better than the cast to a value type: It is much faster and it never casts the value to double before it is casted to the finale data type. This cast to a double can change the value!

C ASRO File Format

To be written