Reference Manual for the XMM SAS Data Access Layer

January 12, 2017

1 Introduction

This document contains reference material for the XMM SAS Data Access Layer (DAL) software.

The DAL's Application Programmable Interface (API) supports four languages; F90, C++, C and Perl.

Primarily, this document will serve as reference material for the F90 DAL API. The C++ and C APIs are also described but with less detail.

Where appropriate, a language-independent approach has been taken and this will partially serve as general user information.

It is a mandatory requirement that all SAS task developers must use the DAL APIs to access FITS files.

2 Data Model

The DAL is principally concerned with data organisation and access.

The DAL Data Model organises data into a collection of data sets.

A data set, which, for aesthetic reasons, will be written as the collocation dataset throughout this document, is an attributable set together with an ordered collection of zero or more blocks. Sometimes a dataset is said to be an attributable set or simply attributable.

A block is one of:

- an array
- a table.

An array is an attributable set together with an n-dimensional array (1 ¡¡ n ¡¡ 3) of numeric scalars. The scalars all have the same type. Sometimes an array is said to be an attributable set or simply attributable.

A table is an attributable set together with an ordered collection of zero or more columns. Sometimes a table is said to be an attributable set or simply attributable.

A column is an attributable set together with a vector of cells. Sometimes a column is said to be an attributable set or simply attributable. A cell is one of:

- a string
- a scalar
- an n-dimensional array (1 ¡¡ n ¡¡ 4) of scalars

A scalar is one of:

- a numeric scalar
- a boolean

A numeric scalar is one of:

- an integer value with 8 bits of precission
- an unsigned integer value with 16 bits of precission
- an integer value with 16 bits of precission
- an unsigned integer value with 32 bits of precission
- an integer value with 32 bits of precission
- a real value with 32 bits of precission
- a real value with 64 bits of precission

The scalars within a particular array or column all have the same type.

An attributable set, which for the purposes of brevity will be shortened to attributable throughout this document, is the quintuple
{ name, label, setofattributes, setofhistoryrecords, setofcommentrecords }
where:

- name A string which is used to provide unique identification.
- label A textual description.
- setofattributes is a ordered collection of zero or more attributes
- setofhistoryrecords is an ordered collection of zero or more history records. A history record is a string.
- setofcommentrecords is an ordered collection of zero or more comment records. A comment record is a string.

An attribute is a quadruple { name, label, unit, value }, where:

- name A string which is used to provide unique identification.
- label A textual description.
- unit A string which is used to provide information on the units of the data.
- value Supported types are:
  - string
  - integer
  - real
  - boolean

3 Abstract API

The DAL is essentially implemented in C++. The design takes the form of a set of abstract classes which provides the fundamental DAL interface. There are currently two implementations (both in C++) of this abstract interface; one is the High Memory Model and the other is the HighLow Memory Model. A third implementation, not yet fully implemented, is the Low Memory Model. The exact nature of these Memory Models is described later in this document.

Three additional interfaces are provided. These are the F90, C and Perl APIs, each of which is implemented, through a transitional layer of C++ code (sometimes called a glue-layer), in terms of the C++ abstract API.

Whilst it is not necessary to understand the underlying Abstract C++ API, it will certainly be beneficial to have at least an overview of the main designs aspects.

In particular, an understanding of the C++ class hierarchy will lead to more generic algorithms e.g. use BlockT rather than ArrayT if only the BlockT methods are needed.

. Class Hierarchy . Object Hierarchy

4 Supported File Formats

The DAL supports three file formats:

- FITS, the preferred format, which conforms to the platform-independent FITS standard.
- DAL, a special internal format (which should be considered platform-specific) that will only be used for temporary intermediate files, probably for (high memory mode) tasks within meta-tasks.
- DECEIT, a special format used by the dataset creation tool deceit. To be used only by specialized tasks only.

When reading an existing file, the DAL will attempt to detect the file format; in the event that a file was opened with the modify mode, the same format will be used when it is rewritten.

When creating a new file, the default output format is FITS. This behaviour may be altered by defining the environment variable SAS_FORMAT. It can have the following values (the corresponding format for each each value is also shown):

- 1, FITS
- 2, DAL
- 3, DECEIT

The output format of a task can be overridden with a command line option.

The data model is abstract in that it is not dependant on an underlying representation (e.g.FITS). The following table shows the mapping to FITS, DAL and deceit files.

There are restrictions imposed by FITS ... e.g. Keyword Length is limited to 8 characters.

5 Clobber

By default when the dal is used to create a new dataset, an existing dataset with the same name will be overwritten. This is referred to as clobbering.

If the environment variable `SAS_CLOBBER` is defined, to determines the clobber behaviour. It can take the following values:

0 no clobber: a task will produce an error when trying to overwrite an existing file.

1 clobber: a task will overwrite existing files. This is the default behaviour, in the event that the `SAS_CLOBBER` variable is undefined.

6 Errors

This section documents warnings and errors generated by this task (if any). Note that warnings and errors can also be generated in the SAS infrastructure libraries, in which case they would not be documented here. Refer to the index of all errors and warnings available in the HTML version of the SAS documentation.

DataSetNonActive *(error)*
found in:../../src/datasetserver/DataSetRecord.cc

FITSError *(error)*
found in:../../src/readerwriter/FitsReaderWriter.cc

FITSFailure *(error)*
found in:../../src/readerwriter/HiLowFitsReaderWriter.cc,../../src/readerwriter/MemFitsReaderWriter.cc

FITSIO *(error)*
found in:../../src/common/dal_utilities.cc

InconsistentDataSet *(error)*
found in:../../src/readerwriter/FitsReaderWriter.cc

InternalError *(error)*
found in:../../src/highlow/HiLowDataSet.cc,../../src/memory/MemDataSet.cc

InvalidCompoundName *(error)*
found in:../../src/common/DalDataSet.cc

InvalidDataType *(error)*
found in:../../src/highlow/Slicer.cc

InvalidMemoryModel *(error)*
found in:../../src/extensions/extdal/ExtDal.cc

InvalidSeek *(error)*
found in:../../src/highlow/Slicer.cc

InvalidType *(error)*
found in:../../src/highlow/Slicer.cc

NoDataSetRecord *(error)*
found in:../../src/datasetserver/DataSetTracker.cc

NoValueAttribute *(error)*
found in:../../src/common/DalAttribute.cc

NotImplemented *(error)*
found in:../../src/highlow/Slicer.cc,../../src/readerwriter/HiLowDalDatabase.cc,../../src/readerwriter/HiLowFit

accessingAsWrongDataType *(error)*
found in:../../src/highlow/HiLowDataStorage.cc,../../src/memory/DataStorage.cc

arrayReadError *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

arrayWriteError *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

attributableNotFound *(error)*
found in:../../src/common/DalAttributable.cc

attributeNameExpected *(error)*
found in:../../src/common/CompoundName.cc

attributeNotFound *(error)*
found in:../../src/common/DalAttributable.cc

attributeNumberNotFound *(error)*
found in:../../src/common/DalAttributable.cc

attributeReadError *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

attributeWriteError *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

blockExists *(error)*
found in:../../src/common/DalDataSet.cc

blockNotFound *(error)*
found in:../../src/common/DalDataSet.cc

blockNumberNotFound *(error)*
found in:../../src/common/DalDataSet.cc

clobberFailed *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc,../../src/readerwriter/HiLowFitsDatabase.cc,../../src/read

columnAlreadyExists *(error)*
found in:../../src/common/DalTable.cc

columnExists *(error)*
found in:../../src/common/DalTable.cc

columnNotFound *(error)*
found in:../../src/common/DalTable.cc

columnReadError *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

copyDataSetFailed *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

couldNotOpenDataSet *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

createDatasetFailed *(error)*
found in:../../src/readerwriter/HiLowFitsDatabase.cc,../../src/readerwriter/MemDalReaderWriter.cc,../../src/r

creatorOfDataSetNotFound *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

dalInternalError *(error)*
found in:../../src/readerwriter/HiLowFitsDatabase.cc

datasetAlreadyExists *(error)*
found in:../../src/readerwriter/HiLowFitsDatabase.cc,../../src/readerwriter/MemFitsReaderWriter.cc

datasetCouldNotBeRead *(error)*
found in:../../src/readerwriter/HiLowFitsReaderWriter.cc,../../src/readerwriter/MemDalReaderWriter.cc,../../s

dateOfDataSetNotFound *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

deleteDataSetFailed *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

deleteDatasetFailed *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

deletingFile *(error)*
found in:../../src/readerwriter/HiLowFitsDatabase.cc

expectedArrayName *(error)*
found in:../../src/common/CompoundName.cc

incompatibleNumberOfRows *(error)*
found in:../../src/common/dal_utilities.cc

incompatibleTables *(error)*
found in:../../src/common/dal_utilities.cc

internalError *(error)*
found in:../../src/common/CompoundName.cc,../../src/common/DalDataComponent.cc,../../src/common/Dal

interrnalError *(error)*
found in:../../src/highlow/Slicer.cc

invalidArrayDataRange *(error)*
found in:../../src/highlow/HiLowArrayData.cc,../../src/memory/MemArrayData.cc

invalidArrayDimension *(error)*
found in:../../src/readerwriter/MemFitsReaderWriter.cc

invalidAttributeDataType *(error)*
found in:../../src/common/DalAttribute.cc,../../src/readerwriter/FitsReaderWriter.cc

invalidAttributeTypeDescription *(error)*
found in:../../src/common/dal_utilities.cc

invalidBlockAccess *(error)*
found in:../../src/common/DalDataSet.cc

invalidBlockPosition *(error)*
found in:../../src/common/DalDataSet.cc

invalidBlockType *(error)*
found in:../../src/readerwriter/HiLowDalReaderWriter.cc,../../src/readerwriter/MemDalReaderWriter.cc

invalidCloneCreateCombination *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

invalidCloneReadCombination *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

invalidColumn *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

invalidColumnDataType *(error)*
found in:../../src/highlow/HiLowColumn.cc,../../src/memory/MemColumn.cc

invalidColumnName *(error)*
found in:../../src/highlow/HiLowColumn.cc,../../src/memory/MemColumn.cc

invalidColumnNumber *(error)*
found in:../../src/common/DalTable.cc

invalidColumnOperation *(error)*
found in:../../src/common/DalColumn.cc,../../src/highlow/HiLowColumn.cc,../../src/memory/MemColumn.cc

invalidColumnPosition *(error)*
found in:../../src/common/DalTable.cc

invalidCompoundName *(error)*
found in:../../src/common/CompoundName.cc,../../src/common/DalAttributable.cc,../../src/common/DalBloc

invalidCopyMode *(error)*
found in:../../src/common/dal_utilities.cc

invalidCountDimensions *(error)*
found in:../../src/highlow/HiLowArrayData.cc,../../src/memory/MemArrayData.cc

invalidDataObject *(error)*
found in:../../src/common/DalDataComponent.cc

invalidDataType *(error)*
found in:../../src/common/dal_utilities.cc,../../src/highlow/HiLowArray.cc,../../src/highlow/HiLowColumn.cc,..

invalidFloatFormat *(error)*
found in:../../src/readerwriter/FitsReaderWriter.cc

invalidFromDimensions *(error)*
found in:../../src/highlow/HiLowArrayData.cc,../../src/memory/MemArrayData.cc

invalidNullOperation *(error)*
found in:../../src/common/DalNullable.cc

invalidNullableOperation *(error)*
found in:../../src/common/DalNullable.cc

invalidPosition *(error)*
found in:../../src/common/DalArrayData.cc,../../src/highlow/HiLowColumnData.cc,../../src/memory/MemCol

invalidRow *(error)*
found in:../../src/highlow/HiLowColumnData.cc,../../src/memory/MemColumnData.cc

invalidRowNumber *(error)*
found in:../../src/common/DalColumn.cc,../../src/highlow/HiLowColumn.cc,../../src/memory/MemColumn.cc

invalidTableName *(error)*
found in:../../src/common/DalBlock.cc

invalidTrailingCharacters *(error)*
found in:../../src/common/DalAttributable.cc

invalidTypeCode *(error)*
found in:../../src/common/dal_utilities.cc

invalidTypeDescription *(error)*
found in:../../src/common/dal_utilities.cc

invalidVectorColumnSize *(error)*
found in:../../src/common/DalTable.cc

missingQuoteFromFITSKeyword *(error)*
found in:../../src/readerwriter/FitsReaderWriter.cc

noDebugHistory *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

noProcessHistory *(error)*
found in:../../src/datasetserver/DalDataSetServer.cc

notDalFormat *(error)*
found in:../../src/readerwriter/MemDalReaderWriter.cc

notImplemented *(error)*
found in:../../src/highlow/HiLowArrayData.cc,../../src/highlow/HiLowCellData.cc,../../src/memory/DataStora

nullNotDefined *(error)*
found in:../../src/common/DalNullable.cc

nullValueFound *(error)*
found in:../../src/common/dal_utilities.cc

nullValueNotDefined *(error)*
found in:../../src/common/DalNullable.cc

openForReading *(error)*
found in:../../src/readerwriter/HiLowDalReaderWriter.cc,../../src/readerwriter/MemDalReaderWriter.cc

openForWriting *(error)*
found in:../../src/readerwriter/HiLowDalReaderWriter.cc

overflow *(error)*
found in:../../src/common/dal_utilities.cc

readOnly *(error)*
found in:../../src/common/DalDal.cc

unhandledNull *(error)*
found in:../../src/common/dal_utilities.cc

unreleasedObject *(error)*
found in:../../src/common/DalDataComponent.cc

unsupportedAttributeDataType *(error)*
found in:../../src/common/dal_utilities.cc

unsupportedColumnType *(error)*
found in:../../src/readerwriter/HiLowFitsReaderWriter.cc,../../src/readerwriter/MemFitsReaderWriter.cc

unsupportedObjectType *(error)*
found in:../../src/common/dal_utilities.cc

wrongArrayDimensions *(error)*
found in:../../src/readerwriter/MemFitsReaderWriter.cc

AttributeNaN *(warning)*
found in:../../src/common/DalAttribute.cc
corrective action:

IllegalColumnAttributeName *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

IllegalTableAttributeName *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

KeywordTooLong *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

ReservedKeyword *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

creatorKeywordNotFound *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

dateKeywordNotFound *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

duplicateTableName *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

overflowDetected *(warning)*
found in:../../src/common/dal_utilities.cc
corrective action:

unnamedTable *(warning)*
found in:../../src/readerwriter/FitsReaderWriter.cc
corrective action:

unreleasedData *(warning)*
found in:../../src/common/DalDataComponent.cc,../../src/highlow/HiLowArray.cc,../../src/highlow/HiLowCol
corrective action:

unsignedAccessedAsSigned *(warning)*
found in:../../src/f90/dal_implementation.cc
corrective action:

7 Test Programs

The DAL comes with a suite of test programs. These are run by doing `make test` in the DAL package directory.

The test program will display a number of warning messages (but no error messages) which may be noted for information, but safely ignored.

8 Example Programs

This document contains a large number of example programs which also are available separately. They can be found in directory dal/doc/reference/examples. The programs are built with the command: make test.

9 Environment Variable

The following environment variables may be set by the user:

- `SAS_CLOBBER`
  Determines the clobber behaviour. 0 implies that datasets may not be overridden, whilst 1 implies otherwise.
- `SAS_FORMAT`
  Determines the output format of new datasets. The following settings are allowed:
  - 1, FITS
  - 2, DAL
  - 3, DECEIT
- `SAS_MEMORY_MODEL`
  Determines which memory model the DAL uses to open datasets. The following settings are allowed:
  - 1, high
  - 2, highlow
  - 3, low - as the low memory model is not implemented, this will default to highlow
- `SAS_ROWS`
  Determines the number of rows selected when iterating over a table. Any positive integer value is allowed.
- `SAS_COLUMN_WISE`
  The DAL can write table columns (to the underlying FITS file) either in a column-by-column or a row-by-row fashion. The default behaviour is row-by-row, but setting the environment variable `SAS_COLUMN_WISE` ensures the column-by-column method is used.

10 Configuation

The DAL uses apart from `dal_conf.txt` also the following file in the SAS configuration directory `$SAS_DIR/config`:

`DAL.msg` Message file. Do not change this file.

Note: the use of the configuration files is subject to change.

11 Access Modes

DataSets are accessed with one of the following modes:

- READ Read an existing dataset with the given name. An error is raised if the dataset is not found, or cannot be opened.
- CREATE Create a new dataset with the given name. If an dataset already exisits with the given name, the behaviour is dependent on the setting of the environment variable `SAS_FORMAT`. Any changes made to the dataset will be discarded upon closure,
- MODIFY Open an existing dataset with the given name. All changes made to the dataset will be written saved upon closure.
- TEMP Open a new dataset. The dataset is discarded upon closure.

When accessing tables, arrays and columns the mode is understood to be only a hint, and gives the DAL an opportunity to be more memory efficient. It should be noted that it is not intended to safeguard the developer from making logical programming mistakes. The underlying reason for this is that the DAL hands out data pointers, and has no way of preventing modifications or even knowing if the data has been modified.

This simple scenario has been adopted because the full bookkeeping on what data has to be written and what data has to be extracted from the original file would have been too expensive in terms of performance and coding.

By default an object inherits the access mode from its parent.

12 Qualified Names

Most objects in the Dal Model have a name, and an owner (usually referred to as the parent). This gives rise to an ancestral sequence of objects, beginning with the dataset and continuing to the object. The Qualified Name, of an object is the token-separated concatenation of names of each object in this sequence.

The colon character is used to separate the names of datasets, blocks and columns, but the % character is used before an attribute name.

e.g. "set:tab:col%att" is the fully qualified name of the column-attribute with name "att", whose parent column has name "col", whose parent table has name "tab", whose parent dataset has name "set".

It is possible to omit the block, column or attribute name from a qualified name. The effect of this is equivalent to specifying the name of the first block in a dataset, the first column in a table, or the first attribute in an attributable e.g.the qualified name "set::col" may be used to access the column with name "col" in the first table contained in the dataset with name "set".

The qualified name of an object may be passed to many of the functions and subroutines. The DAL will parse these names and raise an error if inconsistencies are detected.

13 Long Strings

The FITS standard supports the notion of long strings. These are FITS keywords with string values which exceed 68 characters and are continued across several lines. The continuation is specified with the & character (at the end of each string value to be continued) followed by one or more lines beginning with the reserved keyword CONTINUE (followed by the contined string value).

Long string values are handled transparently by the DAL.

14 Dimensions

It should be noted that when creating either an array or a column in c++ the dimensions are transferred to FITS in the reverse order.

For example, if an array with dimensions (2,4) is created in c++ it will appear in a FITS file as a (4,2) array.

15 Null Values

The DAL supports Null Values (both integer and real).

For objects (i.e. arrays or columns) containing integer data, it is possible to define a null value. A data value is then is considered null if it is equal to the object's null value.

Integer data containing null values may be operated on safely i.e. arithmetic operations.

In the case of real data, a value is considered null if it has an IEEE NaN representation. Real data containing null values must be treated carefully i.e. testing for nullity before carrying out any further operations.

16 Memory Models

There are currently two memory models available; the High Memory and the High/Low Models (The Low Memory Mode, is only partially prototyped and is unlikely to be released in the foreseable future.).

The memory model may be selected by setting the environment variable SAS_MEMORY_MODEL. The allowed settings are:

- `high`
  Select the High Memory Model.
- `highlow` Select the High/Low Memory Model.
- `low` Select the low memory model.

  N.B. The low memory model is not yet implemented, and the highlow memory model will be selected.

When a dataset is opened, with the High Memory Model mode (HMM), it is loaded into memory in its entirety. All subsequent operations are performed on the memory-loaded dataset. When the dataset is closed, the memory is flushed back to disk. The High Memory Mode gives high performance but is memory expensive. With the HMM mode enabled, it is more probable that the operating swapping mechanism will be invoked, giving rise to unacceptibly poor performance.

In the HighLow Memory Model (LMM) mode, when the dataset is opened, only the attributes are loaded into memory. Only when the data is accessed is it loaded into memory. When the data is no longer required it may be released, in which case it is flushed to disk before the memory is freed.

In principle, the LMM mode is capable of using almost no memory, or as much as the high memory mode. A task programmer should take the steps necessary to ensure that the memory consumption is kept to a minimum, and that the task is both high and low memory compatible. The guidelines for this are given later in this document.

The DAL supports the notion of subranges, which essentailly is a convenient mechanism of restricting access to all or only part of the data in a column or an array.

In the HMM mode, a subrange amounts to manipulating memory offsets to the data.

In the LMM mode, however, subranges become very significant because only the data specified in a subrange will be loaded into memory.

17 Iterators

TBD

18 F90 DAL API

18.1 Introduction

The DAL f90 API is contained in the file dal/interface/dal.f90.

The API is specified in the module dal, which contains large number subroutines and functions (many of which are overloaded through interface definitions) together with a number of derived type defintions.

18.2 Handles

Handles hide the internal details of the underlying C++ classes. Essentailly a handle is a pointer to an object, but this detail will be irrelevant to most users.

Corresponding to each of the classes in the Abstract C++ API, there is a derived type.

The derived types ArrayT, AttributableT, AttributeT, BLockT; ColumnT, DataSetT, LabelledT and TableT are used to declare handles of objects.

By having different derived types for each C++ class, the API becomes more robust and type-safe.

18.3 Class Relationships

The C++ classes are related to each other in two ways.

The first is the Base-Derived relationship

The F90 API supports the Base-Derived relationships through Base-Class conversion functions. These are: block, attributable, and labelled.

and the second is the Parent-Child relationship.

These relationships are handled in the F90 API through the parent() function. This function has been overloaded, and returns the parent object of the given object.

The following table shows this simple relationship:

Handle	Class	Base Classes	Parent Class
ArrayT	Array	Block Nullable DataComponent	-
AttributableT	Attributable	Labelled	-
AttributeT	Attribute	Labelled	Attributable
BlockT	Block	Attributable	DataSet
ColumnT	Column	Attributable Nullable DataComponent	Table
DataComponentT	DataComponent	Nullable	-
NullableT	Nullable	-	-
DataSeT	DataSet	Attributable	DataSetServer
LabelledT	Labelled	-	-
TableT	Table	Block	-

Further details are given in the section on the C++ API.

18.4 API Overview

F90 applications must use the module dal, to gain access to the DAL API.

The DAL is concerned with dataset access. A dataset is accessed with the function:

- `dataSet( dataSetName, mode, memoryModel )`

where,

- character(len=*), intent(in) :: dataSetName
  The name of the dataset.
- integer, intent(in) :: mode
  The access mode which the dataset should be used with. It must be one of the following values:

- READ Read an existing dataset with the given name. An error is raised if the dataset is not found, or cannot be opened.
- CREATE Create a new dataset with the given name. If an dataset already exisits with the given name, the behaviour is dependent on the setting of the environment variable SAS_FORMAT. Any changes made to the dataset will be discarded upon closure,
- MODIFY Open an existing dataset with the given name. All changes made to the dataset will be written saved upon closure.
- TEMP Open a new dataset. The dataset is discarded upon closure.

- integer, intent(in), optional :: memoryModel
  This specifies a hint to which the memory model should be used. The following values are possible:
  - HIGH_MEMORY
  - HIGH_LOW_MEMORY
  - LOW_MEMORY
  - USE_ENVIRONMENT

The code extract, shown below, is a typical example of how to use the `dataSet()` function:

```
program test
  use dal

  type(DataSetT) set                   ! Declare a dataset handle.

  set = dataSet( "myset", CREATE )      ! Create a dataset called "myset", with
                                        ! the High Memory Mode. A handle of
                                        ! the newly created dataset is returned
                                        ! and assigned to the variable set.

  ! .......................................
  !
  ! Do some operations on the dataset, using
  ! the handle set.
  !
  ! .......................................

  call release( set )                  ! Close the dataset.

end program
```

The dataset() function returns a handle to a dataset. This handle is used to specify the dataset in subsequent operations. The handle itself is opaque, in that its contents may not be accessed to perform dataset operations. The idea is to think of the handle as being an abstract object called a dataset.

The `release` function is used to close the dataset (the handle of the dataset is passed as a parameter). The behaviour of the release is dependent on the access mode (see **??**) which was used to access the dataset.

Most DAL procedures(functions and subroutines) require a handle as one of the arguments and/or returns a handle value. In many cases a procedure has been overloaded to operate on objects of different types. These have been provided in the form on interfaces. For example, the release interface operates on datasets, blocks, arrays, tables and columns.

18.4.1 Data types

At the lowest level the DAL provides access to data. The following data types are supported:

- `boolean` 1 byte boolean
- `int8` 1 byte unsigned signed integer
- `uint16` 2 byte unsigned integer
- `int16` 2 byte signed integer
- `uint32` 4 byte unsigned integer
- `int32` 4 byte signed integer
- `real32` 4 byte floating point
- `real64` 8 byte floating point
- `string` character string

Note: at this moment the DAL uses 4 bytes logicals instead of 1 byte logicals.

18.4.2 Data set

A data set can be accessed and released again with the following procedures:

- `dataSet` Returns a data set handle, given the name of the dataset and the access mode (see ??).
- `release` Release the data set again. The program should do this as soon as the set is no longer needed.

A data set has an associated set of attributes (see 18.4.3).

The following procedures can be applied to a data set:

- `addTable` Add a table. Requires the name, the data type and shape of a cell and the number of rows for the new table.
- `addArray` Add an array. Requires the name, the datatype and the shape of the array.
- `table` Given a block number or a name, returns a reference to that table.
- `array` Given a block number or a name, returns a reference to that array.
- `block` Given a block number or a name, returns a reference to that block.
- `blockNr` Given a name, returns the number of the block with that name.
- `deleteBlock` Deletes a block. Requires the block number or name.
- `numberOfBlocks` Returns the nr of blocks in the data set.
- `name` Returns the name of the set.

18.4.3 Attributes

A data set, a block (either array or table), and a column have associated *attributes*. An attribute is a keyword-value pair, optionally with a string describing the units of the value and a comment. The following procedures can be used to set and enquire attributes:

- <*type*>`Attribute` Returns the value of an attribute of the specified type. Requires a handle to the object and the name of the attribute.
- `units` Returns a string describing the units of an attribute, given a handle to the object and the name of the attribute.
- `label` Returns the comment string of an attribute, given a handle to the object and the name of the attribute.
- `setAttribute` Sets the value of an attribute. In the case of a numeric attribute, requires the handle of the object, the name of the attribute, the value, the units and a comment. For non-numeric attributes the units are not provided.

There is a utility function available to copy attributes from one object to another:

- `setAttributes` Replaces the attributes by the attributes found in another object. Requires two Attributable handles. A dataset, a block and a column can be converted into an attributable with the attributable call.
- `addAttributes` Adds the attributes that are found in another object. Requires two Attributable handles. A dataset, a block and a column can be converted into an attributable with the attributable call.
- `attribute` Return a handle to the attribute with the specified name.
- `setAttribute` Set an attribute using the result of a previous attribute call.

18.4.4 Block

A data set is an ordered list of blocks. Each block has a name and a set of attributes (see 18.4.3). Two types of blocks exist:

- Table (see 18.4.6)
- Array (see 18.4.5)

Blocks are accessed by number. The number of a block can be found from its name with the `blockNr` function.

The following functions can be called with a block handle:

- `name` Returns the name of the block.
- `number` Returns the block number.

18.4.5 Array

An array is a block (see 18.4.4) that is an n-dimensional array of scalars. An array can be accessed and released with the following procedures:

- `array` Returns an array handle. Requires a data set handle, the name or number of the array and the access mode (see ??).
- `release` Release the array again. The program should do this as soon as the array is no longer needed.

Once the handle is available, the following properties can be enquired:

- `numberOfElements` Returns the number of elements in the array.
- `dimensions` Returns an integer vector with the dimensions of the array.
- `numberOfDimensions` Returns the number of dimensions of the array.

The data in an array can be accessed via access function. There is a large variety of those to support the different types and dimensions.

- <*type*><*dimension*>`Data` Returns a pointer to the data in the array. Requires an array handle.

The following example illustrates how to change the values in an array.

```
program modifyColumn
  use dal
  type(DataSetT) :: set
  type(ArrayT) :: arr
  integer(kind=int32), dimension(:,:), pointer :: x

  set = dataSet("test.dat",Modify)
  arr = array(set,"test")
  x => int32Array2Data(arr)
  x = 124
  call release(arr)
  call release(set)
end program
```

To reduce the size of the array that is accessed in one go (and thereby reducing memory usage), one can select a certain range:

- `arraySeek` Set the range of interest. Requires an array handle and two vectors of integers, indicating the starting position and the size of the range of interest.

Note: the current DAL does not support the seek function.

18.4.6 Table

A table is a block (see 18.4.4) that consists of a set of columns (see 18.4.7), each of which has its own data type. A table can be accessed and released with the following procedures:

- `table` Returns a handle to the table. Requires a data set handle, the table name of number and the access mode (see ??).
- `release` Release the table again. The program should call release as soon as the table is no longer needed.

The table number can be found from its name with the `blockNr` function.

Columns in a table are accessed by name or by number. The number can be found from the name with the `columnNr` function.

The following procedures can operate on a table:

- `addColumn` Adds a column and returns a reference to the new column. Requires the name, the data type and the units for the new column.
- `column` Given a column number, returns a reference to the column.
- `columnNr` Given a name, returns the number of the column with that name.
- `deleteColumn` Deletes a column. Requires the column number.
- `insertRows` Inserts rows into the table. Requires the starting position and the number of rows to insert.
- `deleteRows` Deletes rows from a table. Requires the starting position and the number of rows to delete.
- `copyColumn` Copy a column. Requires the column number, a reference to the destination table and the new name.
- `numberOfColumns` Returns the number of columns.
- `numberOfRows` Returns the number of rows.
- `copyRows` Copies rows. Requires the row number to copy from, the row number to copy to, and the number of rows to copy.
- `insertRows` Inserts rows, containing arbitrary values. Requires the row number where to insert the rows and the number of rows to insert.
- `deleteRows` Deletes rows. Requires the row number of the first row to delete and the number of rows to delete.

18.4.7 Column

A column can be accessed and released again with the procedures

- `column` Return a handle to a column. Requires a table handle, the column name or number, the starting row of interest, the number of rows of interest, and the access mode (see ??).

- `release` Release the column. Requires the column handle.

The following example illustrates how to change the values in a column.

```
program modifyColumn
  use dal
  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: col
  integer(kind=int32), dimension(:), pointer :: x

  set = dataSet("test.dat",Modify)
  tab = table(set,"test")
  col = column(tab,"x",0,0,Modify)
  x => int32Data(col)
  x = 124
  call release(col)
  call release(set)
end program
```

The following procedures can operator on a column handle:

- `name` Returns the name.
- `number` Returns the column number.
- `dataType` Returns the data type of the column.
- `dimensions` Returns the shape of each cell in the column.
- `numberOfDimensions` Returns the number of dimensions of each cell in the column.
- `numberOfRows` Returns the number of rows in the column.

To reduce the size of the column that is accessed in one go (and thereby reducing memory usage), one can select a certain range:

- `columnSeek` Set the range of interest. Requires a column handle, the starting row and the number of rows.

Note: the current implementation of the DAL does not support the column seek function.

18.5 Reference

NAME `addArray`(dataSet, name, type, dimensions, units, comment, position)

PURPOSE

Create and add an array to a dataset.

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet
  A handle to a dataset. The newly created array will be added to this dataset.

- character(len=*), intent(in) :: name
  The name of the array; there must not be a block with this name already in the dataset.
- integer, intent(in) :: type
  The data type of the array. It must be one of Integer8, Integer16, Integer32, Real32, Real64
- integer(KIND=INT32), dimension(:) :: dimensions
  A vector of integer values which desribes the dimensions of the array.
- character(len=*), intent(in), optional :: units
  The units of the array. This is a passive description of the units, which has no effect on the array's data.
- character(len=*), intent(in), optional :: comment
  A textual comment which is used to describe the array.
- integer, intent(in), optional :: position
  This is the ordinal position which the array is to occupy in the given dataset dataSet. The first block in a dataset has position zero.

RETURNS

- type(ArrayT)
  The value returned is a handle to the newly created array.

DESCRIPTION

An array is a block. Create and add an array to the specified dataset. The name must be unique. If no position is specified the array is placed at the end of the dataset. The number of blocks will be increaseed by one. All the data elements of the array will have the specified type. The total number of elements is given by the product of the given dimensions. If a position is specified, the array will have the given position e.g. blockNumber(arrayName) returns the given position. The remaining blocks, if any, will be moved (notionally) to accommodate the new array.

Whilst the DAL data model is independent of any underlying representation, there is an exceptional facility for handling the notion of the FITS Primary Image. An array with block number zero, and name "PRIMARY", will be treated by the FITS file reader/writer as the FITS primary image. This does not spoil the purety of the DAL data model, with FITS specific notions, since the interpretation, is made by the FITS File reader/writer, which is a separate piece of software, which is implementated, in terms of the abstract DAL interface. The restriction of setting the block number to zero, is needed to ensure consistancy of block numbers, between successive create and reads.

The name, units and comment may be changed. The type, dimensions and position may not be changed. In the event that the ordinal position is not at the end, the newly created array is inserted, moving subsequent blocks as necessary.

ERRORS

blockExists invalidBlockPosition

EXAMPLES

```
! In this example add dataset is created (opened) containing
! two 3-dimensional arrays.
!
! It illustrates the use of the derived types DataSetT and ArrayT.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
```

```
!
! The first array is filled with unique data before the
! dataset is released (closed).
program example_addarray

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_addarray
```

SEE ALSO

`array` `block` `blockNumber` `blockType` `deleteBlock` `hasBlock` `numberOfBlocks`

BUGS AND LIMITATIONS

Boolean and String Data types are not supported.

NAME `addAttributes`(destination, source)

PURPOSE

Add the attributes from the source attributable object to the destination attributable object.

ARGUMENTS

- type(AttributableT), intent(in) :: destination
  The destination attributble object.

- type(AttributableT), intent(in) :: source
  The source attributble object.

RETURNS
None

DESCRIPTION
The attributes in source are copied to destination. Attributes in the destination object, which have the same name are overwritten.

ERRORS
None

EXAMPLES

```
program example_addattributes

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.false.,"dataset bool comment")

  tab = addTable(set,"table",10);
  call addAttributes(attributable(tab),attributable(set))
  call release(tab)
  call release(set)

end program example_addattributes
```

SEE ALSO
`setAttributes` `addAttributes`

BUGS AND LIMITATIONS
None known.

NAME `addColumn`(table, columnName, dataKind, units, dimensions, comment, position)

PURPOSE
Create and add a column to a table.

ARGUMENTS

- type(TableT), intent(in) :: table
  A handle to the table to which a column is to be added.
- character(len=*), intent(in) :: columnName
  The name of the column. There must not be a column with the same name already in the table.
- integer, intent(in) :: dataKind
  The type of the column's data. Must be one of Boolean, String, Integer8, Integer16, Interger32, Real32, Real64.

- character(len=*), intent(in), optional :: units
  The units of the column's data. This is a passive description of the units and has no effect on the column's data.
- character(len=*), intent(in), optional :: comment
  A short textual description to be attached to the column.
- integer, dimension(:), optional :: dimensions
  The dimensions of the column's data.
- integer, intent(in), optional :: position
  This is the ordinal position which the column is to occupy in the table. The first column in a table has ordinal position zero.

RETURNS

- type(ColumnT)
  The newly created column is returned as a handle.

DESCRIPTION

Create and add a column to the given table. It is not possible to change the data type, position and dimensions of a column. The name, label and units of a column may be changed. The column handle which is returned is opaque, in that, its contents are hidden from the user, and is an abstract representation of the column.

ERRORS

columnAlreadyExists invalidColumnPosition

EXAMPLES

```
program example_addcolumn

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  logical(kind=bool), dimension(:), pointer :: b
  integer(kind=int8), dimension(:), pointer :: i8
  integer(kind=int16), dimension(:), pointer :: i16
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=single), dimension(:), pointer :: r32
  real(kind=double), dimension(:), pointer :: r64
  character(len=1024) :: s
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  b => boolData(col)
  do i=0,numberOfRows(tab)-1
    b(i) = ( modulo (i,2) .eq. 0 )
  end do

  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
```

```
  i8 => int8Data(col)
  write(*,*) shape(i8)
  do i=0,numberOfRows(tab)-1
    i8(i) = i
  end do

  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  i16 => int16Data(col)
  do i=0,numberOfRows(tab)-1
    i16(i) = 2*i
  end do

  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  i32 => int32Data(col)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do

  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  r32 => real32Data(col)
  do i=0,numberOfRows(tab)-1
    r32(i) = 0.5*i
  end do

  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))
  do i=0,numberOfRows(tab)-1
    write(s,*) "string",i
    call setStringCell(col,i,s)
  end do

  call release(set)

end program example_addcolumn
```

SEE ALSO

`columns` `deleteColumn` `rename` `relabel`

BUGS AND LIMITATIONS

None known.

NAME

`addComment`

PURPOSE

Add a comment string to an attributable object.

INTERFACE

subroutine addCommentToArray(array, comment)
subroutine addCommentToAttributable(attributable, comment)
subroutine addCommentToBlock(block, comment)

subroutine addCommentToDataSet(dataSet, comment)
subroutine addCommentToTable(table, comment)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(BlockT), intent(in) :: block
- character(len=*), intent(in) :: comment
- type(DataSetT), intent(in) :: dataSet
- type(TableT), intent(in) :: table

RETURNS

None

DESCRIPTION

ERRORS

None

EXAMPLES

```
program example_addcomment

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab

  set = dataSet("test.dat",CREATE)
  call addComment(set,"this comment is a dataset comment" )
  call addComment(set,"and so is this one." )

  tab = addTable(set,"some table",100)
  call addComment(tab,"this comment is a table comment" )
  call addComment(tab,"and so is this one." )

  call addComment(block(set,0,MODIFY),"Another table comment")
  call release(set)

end program example_addcomment
```

SEE ALSO

addHistory

BUGS AND LIMITATIONS

Whilst columns are attributable, column comments are not supported when using fits files. This is a result of an underlying limitation of the fits file standard.

NAME

`addHistory`

PURPOSE

Add a history string to an attributable object.

INTERFACE

subroutine addHistoryToArray(array, history)
subroutine addHistoryToAttributable(attributable, history)
subroutine addHistoryToBlock(block, history)
subroutine addHistoryToDataSet(dataSet, history)
subroutine addHistoryToTable(table, history)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableTDataSetT), intent(in) :: attributable
- type(BlockT), intent(in) :: block
- type(DataSetT), intent(in) :: dataSet
- character(len=*), intent(in) :: history
- type(TableT), intent(in) :: table

RETURNS

None

DESCRIPTION

ERRORS

EXAMPLES

```
program example_addhistory

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab

  set = dataSet("test.dat",CREATE)
  call addHistory(set,"this history is a dataset history" )
  call addHistory(set,"and so is this one." )

  tab = addTable(set,"some table",100)
  call addHistory(tab,"this history is a table history" )
  call addHistory(tab,"and so is this one." )
```

```
  call addHistory(block(set,0,MODIFY),"Another table history")
  call release(set)

end program example_addhistory
```

SEE ALSO

addComment

BUGS AND LIMITATIONS

Whilst columns are attributable, column history is not supported. This is a result of an underlying limitation of the fits file standard.

NAME

addTable(dataSet, name, numberOfRows, comment, position)

PURPOSE

Adds a table to a dataset.

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet
  A handle to a dataset to which a table is to be added.
- character(len=*), intent(in) :: name
  The name of the table. There must not be a block with this name already in the dataset.
- integer, intent(in) :: numberOfRows
  The number of rows of the table.
- character(len=*), intent(in), optional :: comment
  A short textual description of the table.
- integer, intent(in), optional :: position
  The ordinal position of the table within the dataset. The first block within a dataset has ordinal position zero.

RETURNS

None

- type(TableT)
  The newly created table is returned as a handle.

DESCRIPTION

The table name must be unique. If no position is specified, the table is placed at the end of the dataset. The number of rows is defined on the table, rather than on the columns, as this ensures that all columns have the same number of rows. The handle returned to the newly created table is opaque, in that the contents are hidden. In the event that the ordinal position is not at the end, subsequent blocks are moved as necessary.

ERRORS

tableAlreadyExists badTablePosition

EXAMPLES

```
! This example shows how the addtable()
! function is used.
program example_addtable

  use dal
```

```
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(BlockT) blk
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table1",10)
  tab = addTable(set,"table2",100)
  tab = addTable(set,"table3",1000)

  do i=0,numberOfBlocks( set ) - 1
    blk = block( set, i, MODIFY )
    write(*,*) name( blk )
    call addComment( blk, "A table comment" )
  end do

  call release(set)

end program example_addtable
```

SEE ALSO

`deleteBlock`

BUGS AND LIMITATIONS

None known.

NAME

`array`

PURPOSE

Get an array from a dataset.

INTERFACE

function arrayWithNumber(dataSet, blockNumber, mode)
function arrayWithName(dataSet, blockName, mode)

ARGUMENTS

- character(len=*), intent(in) :: blockName
  The name of the array to get.
- integer, intent(in) :: blockNumber
  The ordinal position of the array to get.
- type(DataSetT), intent(in) :: dataSet
  A handle to the dataset from which the array will be retrieved.
- integer, intent(in) :: mode
  The access mode which the retrieved array should have. It must be one of the enumeration values: READ, WRITE or MODIFY

RETURNS

- type(ArrayT)
  A handle, to the retrieved array, is returned. All subsequent operations on this handle will operate on the actuall array stored within the dataset.

DESCRIPTION

Retrieve a particular array from a given dataset. The array may be specified either by name or by number (ordinal position within the dataset).

ERRORS

arrayNotFound

EXAMPLES

```
! In this example, a dataset is created containing
! a 3-dimensional array. The array is filled with unique
! numbers, before the dataset is released (closed).
!
! The dataset is then reopened (with READ access),
! and the array's data is displayed.
program example_array

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  integer(kind=int32), dimension(:,:,:), pointer :: a
  integer(kind=int32), dimension(:), pointer :: ad
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr = addArray(set, "some array", INTEGER32, dimensions=s )

  ! fill with unique numbers
  a => int32Array3Data(arr)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a(i,j,k) = n
        n = n + 1
      end do
    end do
  end do

  call release(arr)
  call release(set)

  ! create a set
  set = dataSet("test.dat",READ)
  arr = array(set, "some array", READ)
  ad => int32Data( arr )

  do n = 0, numberOfElements( arr ) - 1
```

```
    write(*,*) ad( n )
  end do

  call release(set)

end program example_array
```

SEE ALSO

addArray block blockType hasBlock

BUGS AND LIMITATIONS

None known.

NAME

ARRAY_BLOCK

PURPOSE

Used to indicate a block of type array.

DESCRIPTION

A block is either an array or a table. The function blockType may be used to determine the type of a given block. In the event that a block is an array the call blockType(someBlock) will return the value ARRAY_BLOCK.

EXAMPLES

```
! In this example add dataset is created containing
! 2 arrays and 2 tables.
!
! A simple loop then iterates over the dataset's
! blocks printing appropriate messages.
! The first two blocks will have ARRAY_BLOCK block type.
! The second two blocks will have TABLE_BLOCK block type.
program example_blocktype

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(BlockT) blk
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr = addArray(set, "block0", INTEGER32, dimensions=s )
  arr = addArray(set, "block1", INTEGER32, dimensions=s )
  tab = addTable(set, "block2", 5 )
  tab = addTable(set, "block3", 5 )

  do i = 0, numberOfBlocks( set ) - 1
```

```
    blk = block( set, i, READ )

    if( blockType( block( set, i, READ ) ) .eq. ARRAY_BLOCK ) then
      write(*,*) "The block with name ", name( blk ), " is an array."
      arr = array( set, i, READ )
      write(*,*) "It has ", numberOfDimensions( arr ), " dimensions."
    end if

    if( blockType( block( set, i, READ ) ) .eq. TABLE_BLOCK ) then
      write(*,*) "The block with name ", name( blk ), " is a table."
      tab = table( set, i )
      write(*,*) "It has ", numberOfRows( tab ), " rows."
    end if
  end do

  call release(arr)
  call release(set)

end program example_blocktype
```

SEE ALSO

addArray block blockType hasBlock

BUGS AND LIMITATIONS

NAME

ArrayT

PURPOSE

A derived type which is used to declare array handles.

DESCRIPTION

The Array type is derived from the Attributable type, which means that an array is attributable, and hence may contain attributes. An object of type ArrayT may be converted to an object of type AttributableT, BlockT or LabelledT.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! two 3-dimensional arrays.
!
! It illustrates the use of the derived types DataSetT and ArrayT.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The first array is filled with unique data before the
! dataset is released (closed).
program example_addarray

  use dal
```

```
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_addarray
```

SEE ALSO

`addArray` `array` `attributable` AttributeT `attribute` AttributableT

BUGS AND LIMITATIONS

None known.

NAME

`attributable`

PURPOSE

Convert a handle to an Attributable handle (AttributeT).

INTERFACE

function arrayAttributable(array)
function blockAttributable(block)
function columnAttributable(column)
function datasetAttributable(dataset)
function tableAttributable(table)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  Convert an array to base type AttributableT
- type(BlockT), intent(in) :: block
  Convert a block to base type AttributableT
- type(ColumnT), intent(in) :: column
  Convert a column to base type AttributableT
- type(DataSetT), intent(in) :: dataset
  Convert a dataset to base type AttributableT
- type(TableT), intent(in) :: table
  Convert a table to base type AttributableT

RETURNS

- type(AttributableT)
  A handle to the Attributable base object.

DESCRIPTION

Convert a handle of a derived type of base type Attributable, to AttributableT. This function allows the programmer to create generic routines, based on the Attributble type, which is the base class for ArrayT, BlockT, ColumnT, DataSetT, and TableT.

ERRORS

None

EXAMPLES

```
! In this example, a dataset is created with one table and one
! array.
! Two attributes are added to each of the dataset, table and array.
!
! The generic subroutine displayAttributes, which operates on the
! AttributableT base type, displays the attributes contained in
! each of the dataset, table and array.
subroutine displayAttributes( thisAttributable )
  use dal

  implicit none

  type(AttributableT) thisAttributable
  type(AttributeT) att
  integer i

  do i = 0, numberOfAttributes( thisAttributable ) - 1
    att = attribute( thisAttributable, i )
    write(*,*) name( att ), stringAttribute( att ), units( att ), label( att )
  end do

end subroutine displayAttributes

program example_attributable

  use dal

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset first bool comment")
  call setAttribute(set,"sbool2",.true.,"dataset second comment")

  tab = addTable(set,"table",10);
  call setAttribute(set,"int1",1,"table first integer comment","kg")
  call setAttribute(set,"int2",2,"table second integer comment","mm")

  arr = addArray(set, "array", INTEGER32, dimensions=s )
  call setAttribute(set,"real1",1.1,"array first real comment","kN")
  call setAttribute(set,"real2",2.3,"array second real comment","rad")

  call displayAttributes( attributable( set ) )
  call displayAttributes( attributable( tab ) )
  call displayAttributes( attributable( arr ) )

  call release(set)

end program example_attributable
```

SEE ALSO

AttributableT

BUGS AND LIMITATIONS

None known.

NAME

AttributableT

PURPOSE

A derived type which is used to declare attributable handles.

DESCRIPTION

The derived type AttributableT is a base type for ArrayT, BlockT, ColumnT, DataSetT and TableT.

ERRORS

EXAMPLES

See attributable.

SEE ALSO

`attributable` ArrayT BlockT ColumnT DataSetT TableT

BUGS AND LIMITATIONS

None known.

NAME

`attribute`

PURPOSE

Get an attribute from an attributable object.

INTERFACE

function arrayAttributeWithName(array, name)
function arrayAttributeWithNumber(array, number)
function attributableAttributeWithName(attributable, name)
function attributableAttributeWithNumber(attributable, number)
function blockAttributeWithName(block, name)
function blockAttributeWithNumber(block, number)
function columnAttributeWithName(column, name)
function columnAttributeWithNumber(column, number)
function dataSetAttributeWithName(dataSet, name)
function dataSetAttributeWithNumber(dataSet, number)
function tableAttributeWithName(table, name)
function tableAttributeWithNumber(table, number)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of an array object from which to get the attribute.
- type(AttributableT), intent(in) :: attributable
  A handle of an attributable object from which to get the attribute.
- type(BlockT), intent(in) :: block
  A handle of a block object from which to get the attribute.
- type(ColumnT), intent(in) :: column
  A handle of a column object from which to get the attribute.
- type(DataSetT), intent(in) :: dataSet
  A handle of an attribute object from which to get the attribute.
- character(len=*), intent(in) :: name
  The name of the attribute to get.
- integer, intent(in) :: number
  The ordinal number of the attribute to get.
- type(TableT), intent(in) :: table
  A handle of a table object from which to get the attribute.

RETURNS

- type(AttributeT)
  The attribute handle of the retrieved attribute.

DESCRIPTION

Get an attribute, either by name or by number (ordinal position within the attributable object) from an attributable object.

ERRORS

attributeNotFound invalidAttributeNumber

EXAMPLES

```
! In this example a dataset with a table is ! created.
! Two attributes are added to each of the dataset ! and table.
! The attribute names of the dataset are displayed using
! access-by-number, and the table attribute names are displayed
! using access-by-name.
program example_attribute

  use dal
```

```
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.true.,"dataset bool comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"sbool1",.false.,"table bool comment")
  call setAttribute(tab,"sbool2",.true.,"table bool comment")

  write(*,*) name( attribute( set, 0 ))
  write(*,*) name( attribute( set, 1 ))
  write(*,*) name( attribute( tab, "sbool1" ))
  write(*,*) name( attribute( tab, "sbool2" ))

  call release(set)

end program example_attribute
```

SEE ALSO

AttributableT AttributeT

BUGS AND LIMITATIONS

None known.

NAME

AttributeT

PURPOSE

A derived type used to declare Attribute handles.

DESCRIPTION

An attribute has a name, value, comment and units. The value of an attribute is not strongly typed; type conversions are carried out when the attributes's value is accessed. Objects of type AttributeT are not attributes but attribute-handles. An attribute handle provides an abstract access layer to the internal structure of an attribute. An attribute belongs (is owned by) to an attributable (or any of its derived types) object. The parent (owner) of an attribute is an attributable object. An attribute is created with a call to addAttribute, which creates and adds a new attribute to an attributable object. An attribute is retrieved from an attributable object with the attribute function.

ERRORS

EXAMPLES

See addAtribute.

SEE ALSO

`addAttribute` `attribute` `parent` AttributableT

BUGS AND LIMITATIONS

None known.

NAME

`block`

PURPOSE

Convert a handle, of derived type of base type Block, to BlockT

INTERFACE

function arrayBlock(array)
function tableBlock(table)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  Handle of array which is to be converted into a handle of BlockT.
- type(TableT), intent(in) :: table
  Handle of table which is to be converted into a handle of BlockT.

RETURNS

- type(BlockT)
  A Block handle is returned.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example, a dataset is created with one table and one
! array.
! The generic subroutine displayBlock, which operates on the
! BlockT base type. The blockType() function operates on objects
! of type BlockT.
! The example also sjows blocks being retrieved from the dataset
! both by name and by number.
subroutine displayBlock( thisBlock )
  use dal

  implicit none

  type(BlockT) thisBlock

  write(*,*) "The block with name ", name( thisBlock )

  if( blockType( thisBlock ) .eq. ARRAY_BLOCK ) then
    write(*,*) " is an array."
  end if

  if( blockType( thisBlock ) .eq. TABLE_BLOCK ) then
    write(*,*) " is a table."
  end if

end subroutine displayBlock

subroutine displayBlocks( thisSet )
```

```
  use dal

  implicit none

  type(DataSetT) thisSet
  integer i

  do i = 0, numberOfBlocks( thisSet ) - 1
    call displayBlock( block( thisSet, i, READ ) )
  end do

end subroutine displayBlocks

program example_block

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table",10);
  arr = addArray(set, "array", INTEGER32, dimensions=s )

  call displayBlock( block( tab ) )
  call displayBlock( block( arr ) )
  call displayBlock( block( set, "table", READ ) )
  call displayBlock( block( set, "array", READ ) )
  call displayBlocks( set )

  call release(set)

end program example_block
```

SEE ALSO

BlockT

BUGS AND LIMITATIONS

None known.

NAME

`block`

PURPOSE

Get a block from a dataset.

INTERFACE

function blockWithNumber(dataSet, blockNumber, mode)
function blockWithName(dataSet, blockName, mode)

ARGUMENTS

- character(len=*), intent(in) :: blockName
  Name of block which is to be retrieved from a dataset.
- integer, intent(in) :: blockNumber
  Number of block which is to be retrieved from a dataset.
- type(DataSetT), intent(in) :: dataSet
  Handle of a dataset from which a block is to be retrieved.
- integer, intent(in) :: mode
  The access mode which the retrieved block should have. It must be one of the enumeration values: READ, WRITE or MODIFY.

RETURNS

- type(BlockT)
  The Block handle is returned.

DESCRIPTION

The block may be retrieved either by name or by number (i.e. ordinal position) from the dataset.

ERRORS

blockNotFound

EXAMPLES

```
! In this example, a dataset is created with one table and one
! array.
! The generic subroutine displayBlock, which operates on the
! BlockT base type. The blockType() function operates on objects
! of type BlockT.
! The example also sjows blocks being retrieved from the dataset
! both by name and by number.
subroutine displayBlock( thisBlock )
  use dal

  implicit none

  type(BlockT) thisBlock

  write(*,*) "The block with name ", name( thisBlock )

  if( blockType( thisBlock ) .eq. ARRAY_BLOCK ) then
    write(*,*) " is an array."
  end if

  if( blockType( thisBlock ) .eq. TABLE_BLOCK ) then
    write(*,*) " is a table."
  end if

end subroutine displayBlock

subroutine displayBlocks( thisSet )
  use dal

  implicit none
```

```
  type(DataSetT) thisSet
  integer i

  do i = 0, numberOfBlocks( thisSet ) - 1
    call displayBlock( block( thisSet, i, READ ) )
  end do

end subroutine displayBlocks

program example_block

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table",10);
  arr = addArray(set, "array", INTEGER32, dimensions=s )

  call displayBlock( block( tab ) )
  call displayBlock( block( arr ) )
  call displayBlock( block( set, "table", READ ) )
  call displayBlock( block( set, "array", READ ) )
  call displayBlocks( set )

  call release(set)

end program example_block
```

SEE ALSO

BlockT

BUGS AND LIMITATIONS

None known.

NAME

`blockNumber`(dataSet, blockName)

PURPOSE

Get the number (ordinal position) of a block.

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet
  Handle of a dataset which contains the desired block.
- character(len=*), intent(in) :: blockName
  The name of the block for which the number is required.

RETURNS

- integer
  The ordinal number (position of the block within the dataset).

DESCRIPTION

A dataset contains zero or more blocks. Each block has an ordinal number (or position) within its parent dataset. This function returns the ordinal position of a block.

ERRORS

blockNotFound

EXAMPLES

```
! This example creates a dataset with one
! table and one array.
! The table will have block number 0,
! and the array will have block number 1
subroutine displayBlockNumber( thisBlock )
  use dal

  implicit none

  type(BlockT) thisBlock

  write(*,*) "The block with name ", name( thisBlock ), "has number "
  write(*,*) blockNumber( parent( thisBlock ), name( thisBlock ) )

end subroutine displayBlockNumber

program example_blocknumber
  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table",10);
  arr = addArray(set, "array", INTEGER32, dimensions=s )

  call displayBlockNumber( block( tab ) )
  call displayBlockNumber( block( arr ) )

  call release(set)

end program example_blockNumber
```

SEE ALSO

BlockT DataSetT

BUGS AND LIMITATIONS

None known.

NAME

BlockT

PURPOSE

A derived type used to declare block handle objects.

DESCRIPTION

This type allows generic routines to be writeen, which do not need to worry whether a particular block is an array or a table. The Block type is derived from the Attributable type, which means that a block is attributable, and may thus contain attributes.

EXAMPLES

See block, blockNumber

SEE ALSO

`addAttribute` `attributable` AttributbleT `attribute` AttributeT DataSetT

BUGS AND LIMITATIONS

None known.

NAME

`blockType`

PURPOSE

Get the type of a block.

INTERFACE

function blockType(block)
function blockTypeOfBlockWithName(dataSet, blockName)
function blockTypeOfBlockWithNumber(dataSet, blockNumber)

ARGUMENTS

- type(BlockT), intent(in) :: block
  A handle to the block.
- character(len=*), intent(in) :: blockName
  The name of the block.
- integer, intent(in) :: blockNumber
  The number of the block.
- type(DataSetT), intent(in) :: dataSet
  A handle to the dataset containing the desired block.

RETURNS

- integer
  Returns ARRAY_BLOCK if the block is an array and returns TABLE_BLOCK if the block is a table.

DESCRIPTION

Determines the type of the given block.

ERRORS

blockNotFound

EXAMPLES

```
! In this example add dataset is created containing
! 2 arrays and 2 tables.
!
! A simple loop then iterates over the dataset's
! blocks printing appropriate messages.
```

```
! The first two blocks will have ARRAY_BLOCK block type.
! The second two blocks will have TABLE_BLOCK block type.
program example_blocktype

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(BlockT) blk
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr = addArray(set, "block0", INTEGER32, dimensions=s )
  arr = addArray(set, "block1", INTEGER32, dimensions=s )
  tab = addTable(set, "block2", 5 )
  tab = addTable(set, "block3", 5 )

  do i = 0, numberOfBlocks( set ) - 1
    blk = block( set, i, READ )

    if( blockType( block( set, i, READ ) ) .eq. ARRAY_BLOCK ) then
      write(*,*) "The block with name ", name( blk ), " is an array."
      arr = array( set, i, READ )
      write(*,*) "It has ", numberOfDimensions( arr ), " dimensions."
    end if

    if( blockType( block( set, i, READ ) ) .eq. TABLE_BLOCK ) then
      write(*,*) "The block with name ", name( blk ), " is a table."
      tab = table( set, i )
      write(*,*) "It has ", numberOfRows( tab ), " rows."
    end if
  end do

  call release(arr)
  call release(set)

end program example_blocktype
```

SEE ALSO

`addBlock` `block` BlockT

BUGS AND LIMITATIONS

None known.

NAME

BOOL

PURPOSE

Native fortran enumeration which is used to indicate boolean data.

DESCRIPTION

ERRORS

EXAMPLES

N/A

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

BOOLEAN

PURPOSE

Enumeration value which is used to indicate DAL boolean data.

DESCRIPTION

This value should not be confused with the Native fortran value BOOL. Boolean data values are 4-byte logicals.

EXAMPLES

N/A

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`booleanAttribute`

PURPOSE

Get the value of an attribute as a boolean.

INTERFACE

function booleanArrayAttribute(array, name)
function booleanAttributableAttribute(attributable, name)
function booleanAttribute(attribute)
function booleanBlockAttribute(block, name)
function booleanColumnAttribute(column, name)
function booleanDataSetAttribute(dataSet, name)
function booleanTableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute

- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- character(len=*), intent(in) :: name
- type(TableT), intent(in) :: table

RETURNS

- logical

DESCRIPTION
Gets the value of an attribute as a boolean from the specified object. In the event that the data conversion is not possible an error is raised.

ERRORS

EXAMPLES

```
! This example shows how boolean attributes are used.
! The program creates a dataset containing two boolean attributes,
! together with a table containing two boolean attributes.
! The attributes are then accessed, by name, with
! the booleanAttribute() function.
! Also, it is shown how to access the attributes by position.
program example_booleanattribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.true.,"dataset bool comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"sbool1",.false.,"dataset bool comment")
  call setAttribute(tab,"sbool2",.true.,"dataset bool comment")

  write(*,*) booleanAttribute( set, "sbool1" ) ! output 'F'
  write(*,*) booleanAttribute( set, "sbool2" ) ! output 'T'
  write(*,*) booleanAttribute( tab, "sbool1" ) ! output 'F'
  write(*,*) booleanAttribute( tab, "sbool2" ) ! output 'T'
```

```
  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) booleanAttribute( att ) ! output the sequence 'F','T'
  end do

  call release(set)

end program example_booleanattribute
```

SEE ALSO

AttributeT

BUGS AND LIMITATIONS

NAME

`boolArray2Data`

PURPOSE

Get the boolean data from an array or column cell containing 2-dimensional array data.

INTERFACE

function boolArrayArray2Data(array)
function boolColumnArray2DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- logical(kind=bool), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling
```

```
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
    c1 => boolArray2Data(col1,k)
    c2 => boolArray2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = .false.
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

boolArrayArray2Data is not implemented.

NAME

`boolArray2Data`

PURPOSE

Get the boolean data from a column containing 2-dimensional array data.

INTERFACE

function boolColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- logical(kind=bool), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => boolArray2Data(col1)
  c2 => boolArray2Data(col1)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = .false.
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
```

```
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolArray3Data`

PURPOSE

Get the boolean data from an array or column cell containing 3-dimensional array data.

INTERFACE

function boolArrayArray3Data(array)
function boolColumnArray3DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- logical(kind=bool), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
```

```
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => boolArray3Data(col1,l)
    c2 => boolArray3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = .false.
          c2(i,j,k) = c1(i,j,k)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolArray3Data`

PURPOSE

Get the boolean data from a column containing 3-dimensional array data.

INTERFACE

function boolColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- logical(kind=bool), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => boolArray3Data(col1)
  c2 => boolArray3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = .false.
          c2(i,j,k,l) = c1(i,j,k,l)
```

```
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolArray4Data`

PURPOSE

Get the boolean data from a column cell containing 4-dimensional array data.

INTERFACE

function boolColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- logical(kind=bool), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
```

```
program example_cellarray4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => boolArray4Data(col1,m)
    c2 => boolArray4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = .false.
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

boolArray4Data

PURPOSE

Get the boolean data from a column containing 4-dimensional array data.

INTERFACE

function boolColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- logical(kind=bool), dimension(:,:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => boolArray3Data(col1)
  c2 => boolArray3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
```

```
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = .false.
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolData`

PURPOSE

Get the boolean data from an array, column or column cell.

INTERFACE

function boolArrayData(array)
function boolColumnData(column)
function boolColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array which contains the data to be accessed.
- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.
- integer, intent(in) :: row
  The number of the column cell which contains the data to be accessed.

RETURNS

- logical(kind=bool), dimension(:), pointer
  The data is returned as a flat vector regardless of the dimensionality of the data.

DESCRIPTION

The data is returned in a vector regardles of the dimensionality of the data. In particular, when accessing a scalar column cell, a vector of length 1 is returned, which contains the single scalar value.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_booldata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:,:,:,:), pointer :: c1, c2
  logical(kind=BOOL), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => boolArray4Data(col1)
  c2 => boolArray4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = .false.
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
  cd => boolData( col2 ) ! Access the column's 4-dimensional data as a flat vector.
```

```
  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_booldata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolVectorData`

PURPOSE

Get the data from an array or column cell containing vector array data.

INTERFACE

function boolArrayVectorData(array)
function boolColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array which contains the data to be accessed.
- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.
- integer(kind=INT32), intent(in) :: row
  The number of the column cell which contains the data to be accessed.

RETURNS

- logical(kind=bool), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell).
! The data is output on a cell-by-cell basis and accessing
```

```
! the cell as a flat vector.
program example_boolcellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => boolVectorData(col1,m)
    c2 => boolVectorData(col2,m)
    do i=0,2
      c1(i) = .false.
      c2(i) = c1(i)
      n = n + 1
    end do
    ! release(col1)
    ! release(col2)
  end do

  ! Output col2
  do m=0,numberOfRows(tab) - 1
    c2 => boolVectorData(col2,m)
    do n=0,numberOfElements(col2) - 1
      write(*,*) c2(n)
    end do
    ! release(col2)
  end do

  call release(set)

end program example_boolcellvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`boolVectorData`

PURPOSE
Get the data from a column containing vector array data.

INTERFACE
function boolColumnVectorData(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array which contains the data to be accessed.
- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.
- integer(kind=INT32), intent(in) :: row
  The number of the column cell which contains the data to be accessed.

RETURNS

- logical(kind=bool), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  logical(kind=BOOL), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", BOOLEAN, "km", s, "column comment" )
```

```
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => boolVectorData(col1)
  c2 => boolVectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
      c1(i,m) = .false.
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`clobber()`

PURPOSE

Get the clobber setting.

ARGUMENTS

None

RETURNS

- logical
  True, if the SAS Clobber setting is set, false otherwise.

DESCRIPTION

The SAS Clobber setting is determined by the setting of the SAS_CLOBBER environment variable.

ERRORS

None.

EXAMPLES

```
! This example shows how the clobber() function is
! used.
program example_clobber

  use dal
  implicit none
```

```
  if( clobber() ) then
    write(*,*) "The SAS_CLOBBER environment variable is set"
  else
    write(*,*) "The SAS_CLOBBER environment variable is not set"
  endif

end program example_clobber
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`column`

PURPOSE

Get a column from a table.

INTERFACE

column

- functon columnWithName(table, columnName, mode)
- function columnWithNumber(table, columnNumber, mode)

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table from which to get the column.
- character(len=*), intent(in) :: columnName
  The name of the column in the table.
- integer, intent(in) :: columnNumber
  The ordinal number of the column. It must be in the range 0 to n - 1, where n is the number of columns in the table.
- integer, intent(in) :: mode The access mode which will be given to the retrieved column. The options are: READ—WRITE—MODIFY

RETURNS

- type(ColumnT)
  A handle of the retrieved column.

DESCRIPTION

In the event that the column was not found an error is raised. An error is usually raised when the column name does not exist or the ordinal number is invalid. The handle (which is essentailly a pointer to the internal column, but this detail is hidden by the API) which is returned may be (in fact this is the only way to modify a column) used to perform various operations on the column. A column is deleted from a table with deleteColumn.

ERRORS

EXAMPLES

```
! This examples show how the column() function is used.
! The column by name is used to get a column and rename it.
! The column by number is used to iterate over all
! columns in the table to output the name, type and units.
program example_column

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"col1",INTEGER32,units="m1",comment="in32 column")
  col = addColumn(tab,"col2",INTEGER32,units="m2",comment="in32 column")
  col = addColumn(tab,"col3",INTEGER32,units="m3",comment="in32 column")

  col = column( tab, "col2", MODIFY )
  call rename( col, "col4" )

  do i =0, numberOfColumns( tab ) - 1
    col = column( tab, i, READ )
    write(*,*) name( col ), columnDataType( col ), units( col )
  end do

  call release(set)

end program example_column
```

SEE ALSO

`addColumn` ColumnT

BUGS AND LIMITATIONS

None known.

NAME

`columnNumber`(table, columnName)

PURPOSE

Get the number (ordinal position) of a column.

ARGUMENTS

- type(TableT), intent(in) :: table
  The table containing the column for which the number is required.
- character(len=*), intent(in) :: columnName
  The name of the column.

RETURNS

- integer
  The value returned will be in the range 0 to n - 1, where n is the number of columns in the table.

DESCRIPTION

The first column in a table has number zero. In the event that a column with the specified name is not found in the given table, an error will be raised. A column's number will change when additional columns are added, or when columns are deleted to earlier positions.

ERRORS

columnNotFound

EXAMPLES

```
! This examples shows how the columnNumber() function
! is used.
program example_columnnumber

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")

  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column", &
          position=columnNumber( tab, "int32" ) )

  do i = 0, numberOfColumns( tab ) - 1
    col = column( tab, i, READ )
    write(*,*) name( col )
  end do

  call release(set)

end program example_columnnumber
```

SEE ALSO

addColumn

BUGS AND LIMITATIONS

None known.

NAME

ColumnT

PURPOSE

A derived type which is used to declare column handles.

DESCRIPTION

A derived type which is used to declare column handles. A column always belongs to a table. A columns is created with the addColumn() function, which requires a table as an argument. This table becomes the owning parent of the column. The Column type is derived from the Attributable type, which means that a column is attributable, and may thus contain attributes.

ERRORS

EXAMPLES

See addColumn

SEE ALSO

`addColumn` `attributable` AttributableT `attribute` AttributeT `column`

BUGS AND LIMITATIONS

None known.

NAME

`copyBlock`(set, block, newName)

PURPOSE

Copy a block.

ARGUMENTS

- type(DataSetT), intent(in) :: set
  The handle of a dataset to which the block should be copied.
- type(BlockT), intent(in) :: block
  The handle of a block which is to be copied to the dataset.
- character(len=*), intent(in), optional :: newName
  An optional name of the newly copied block.

RETURNS None

DESCRIPTION

Copy a block to a dataset. The source block may reside in a differnt dataset from set. A duplicate of the source block is created, either with the same name as the source block or with a new name speicfied with newName. If newName is not specified, the newly created block will have the same name as the source block. It will be necessary to specify the name of the new block when block resides in the dataset set. In the event that a block cannot be copied (e.g. non-unique name) an error will be raised.

ERRORS

EXAMPLES

```
! In this example a dataset is created, with
! three tables.
!
! The created datanew is then copied to a new dataset.

! A simple loop then iterates over the
! dataset's blocks (a table may be treated as a block)
! Each block is copyied to a second dataset, and
! then displays the name of each new block (which actually
! will be the same name as the source block); a comment
```

```
! is added to each new block.
program example_copyblock

  use dal

  implicit none

  type(DataSetT) set1, set2
  type(TableT) tab
  type(BlockT) blk
  integer i

  set1 = dataSet("test.dat",CREATE)
  tab = addTable(set1,"first table",100)
  tab = addTable(set1,"second table",1000)
  tab = addTable(set1,"third table",10000)

  set2 = dataSet("test1.dat",CREATE)

  do i = 0, numberOfBlocks( set1 ) - 1
    blk = block( set1, i, MODIFY )
    call copyBlock( set2, blk )
    blk = block( set2, i, MODIFY )
    call addComment( blk, "Copied from test.dat" )
  end do

  call release(set1)
  call release(set2)

end program example_copyblock
```

SEE ALSO

`addBlock`

BUGS AND LIMITATIONS

None known.

NAME

`copyColumn`(table, column, newName)

PURPOSE

Copy a column.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table to which the column should be copied.
- type(ColumnT), intent(in) :: column
  The handle of a column to be copied.
- character(len=*), intent(in), optional :: newName
  An optional name for the newly copied column should have.

RETURNS None

DESCRIPTION

The source column may reside in a different table. A duplicate of the sourcee column is created, whose name is either the same as the source column's name or a new name specified with newName.

ERRORS

EXAMPLES

```
! This examples show how to use the copyColumn function.
program example_copycolumn

  use dal

  implicit none

  type(DataSetT) set1, set2
  type(TableT) tab1, tab2
  type(ColumnT) col1, col2, col3, col4
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=single), dimension(:), pointer :: r32
  integer i

  set1 = dataSet("test.dat",CREATE)
  tab1 = addTable(set1,"some table",100)

  col1 = addColumn(tab1,"col1",INTEGER32,units="m",comment="in32 column")
  i32 => int32Data(col1)
  do i=0,numberOfRows(tab1)-1
    i32(i) = 3*i
  end do
  call release( col1)

  col2 = addColumn(tab1,"col2",REAL32,units="Dm",comment="real32 column")
  r32 => real32Data(col2)
  do i=0,numberOfRows(tab1)-1
    r32(i) = 0.5*i
  end do
  call release( col2)

  set2 = dataSet("test1.dat",CREATE)
  tab2 = addTable(set2,"some table",100)
  call copyColumn( tab2, col1 )
  call copyColumn( tab2, col2, "col3" )

  col3 = column( tab2, name( col1 ), READ )
  col4 = column( tab2, "col3", READ )
  i32 => int32Data(col3)
  r32 => real32Data(col4)
  do i = 0, numberOfRows( tab2 ) - 1
    write(*,*) i32(i), r32(i)
  end do
  call release(col3)
  call release(col4)

  call release(set1)
  call release(set2)

end program example_copycolumn
```

SEE ALSO

`addColumn`

BUGS AND LIMITATIONS

None known.

NAME

`copyDataSet`(sourceName, destName)

PURPOSE

Copy a dataset.

ARGUMENTS

- character(len=*), intent(in) :: destName
  The name of the destination dataset.
- character(len=*), intent(in) :: sourceName
  The name of the destination source set.

RETURNS None

DESCRIPTION

Copy the dataset with name sourceName to the set with name destName. The destination dataset becomes a duplicate of the source dataset. In the event that the source dataset is not found an error will be raised.

ERRORS

EXAMPLES

```
! In this example a dataset is created, with
! three tables.
! The created datanew is then copied to a new dataset.
! A simple loop then iterates over the new
! dataset's ! blocks (a table may be treated as a block)
! then displays the name of ! each table, and adds a comment
! to each block (table).
program example_copydataset

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(BlockT) blk
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"first table",100)
  tab = addTable(set,"second table",1000)
  tab = addTable(set,"third table",10000)

  call release(set)
```

```
  call copyDataSet( "test.dat", "test1.dat" )

  set = dataSet("test1.dat",READ)
  do i = 0, numberOfBlocks( set ) - 1
    blk = block( set, i, MODIFY )
    write(*,*) name( blk )
    call addComment( blk, "Copied from test.dat" )
  end do

  call release(set)

end program example_copydataset
```

SEE ALSO

`addDataSet`

BUGS AND LIMITATIONS

This subroutine has been implemented with a subprocess call to the unix rm command. The DAL has no control over the overall behaviour of this command.

NAME

`clone`(from, to, mode, memoryModel)

PURPOSE

Clone a dataset.

ARGUMENTS

- character(len=*), intent(in) :: from
  The name of the dataset to be cloned.
- character(len=*), intent(in) :: to
  The name of the clone.
- integer, intent(in) :: mode
  The access mode which the dataset should be used with. It must be one of the following values:
  - MODIFY All changes made to the clone will be written saved upon closure.
  - TEMP The clone is discarded upon closure.
- integer, intent(in), optional :: memoryModel
  This specifies a hint to which the memory model should be used. The following values are possible:
  - HIGH_MEMORY
  - HIGH_LOW_MEMORY
  - LOW
  - USE_ENVIRONMENT

RETURNS

- type(DataSetT) :: dataSet
  A handle to the clone .

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the clone
! function.
program example_clone

  use dal

  implicit none

  type(DataSetT) set
  type(DataSetT) clonedSet

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1", 10, "mm", "attribute comment" )
  call release(set)

  set = dataSet("test.dat",MODIFY)
  call setAttribute(set,"att1", 10, "mm", "attribute comment" )
  call release(set)


  clonedSet = clone("test.dat","test2.dat",MODIFY)
  call setAttribute(clonedSet,"att2", 10, "mm", "attribute comment" )
  call release(clonedSet)

  set = dataSet("test2.dat",READ)
  write(*,*) "att2 = ", int32Attribute( set, "att2" );
  call release(set)

end program example_clone
```

SEE ALSO

`dataSet` `release` `setexist` HIGH_MEMORY HIGH_LOW_MEMORY LOW_MEMORY USE_ENVIRONMENT

BUGS AND LIMITATIONS

None known.

NAME

`copyRows`(table, from, to, count)

PURPOSE

Copy a range of rows.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table within which the rows are to be copied.
- integer, intent(in) :: from
  The source row number. 'from' must be in the range 0 to n, where n is the number of rows in the table. The source row numbers will then be in the range 'from' to 'from + count'. See the description below for additional constraints.
- integer, intent(in) :: to
  The destination row number. 'to' must be in the range 0 to n, where n is the number of rows in the table. The destination row numbers will then be 'to' to 'to + count'. See the description below for additional constraints.

- integer, intent(in), optional :: count
  The number of rows which should be copied. See the description below for additional constraints.

RETURNS None

DESCRIPTION

Copy a range of rows from one location to another, within a table. The triple (from, to, count) must be logically possible e.g. from + count ¡= n and to + count ¡= n, where n is the number of rows in the table.

Memory Considerations This operation of copying rows within a table is very expensive.

ERRORS

EXAMPLES

```
! This examples show how to use the copyRows subroutine.
program example_copyrows

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=single), dimension(:), pointer :: r32
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",10)

  col1 = addColumn(tab,"col1",INTEGER32,units="m",comment="in32 column")
  i32 => int32Data(col1)

  do i=0,4
    i32(i) = 3*i
  end do

  call release( col1)

  col2 = addColumn(tab,"col2",REAL32,units="Dm",comment="real32 column")
  r32 => real32Data(col2)

  do i=0,4
    r32(i) = 0.5*i
  end do

  call release( col2)

  call copyRows( tab, 0, 5, 5 ) ! copy range [0,4] to [5,9]

  i32 => int32Data(col1)
  r32 => real32Data(col2)
```

```
  do i = 0, numberOfRows( tab ) - 1
    write(*,*) i32(i), r32(i)
  end do

  call release(col1)
  call release(col2)

  call release(set)

end program example_copyrows
```

SEE ALSO

`deleteRows` `insertRows`

BUGS AND LIMITATIONS

None known.

NAME

`count`

PURPOSE

Get the count-value from the seek range of an object.

INTERFACE

function countColumn(column) function countTable(table)

ARGUMENTS

- type(ColumnT), intent(in) :: column
- type(TableT), intent(in) :: table

RETURNS

- integer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the seek functions
! are used.
! This subroutine will dispaly the seek values of the given table and column.
subroutine whatisseek(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: col

  write(*,*) from( tab ), count( tab )

  col = column(tab,"x",MODIFY)
```

```
  write(*,*) from( col ), count( col )

end subroutine whatisseek

program example_seek

  use dal

  implicit none

  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: col

  interface
    subroutine whatisseek( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine whatisseek
  end interface

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"events",10)
  col = addColumn(tab,"x",real32,"mm")

  call forEachSubTable(tab,whatisseek)

  call release(set)

end program example_seek
```

SEE ALSO

`from`

BUGS AND LIMITATIONS

None known.

NAME

CREATE

PURPOSE

An enumeration value which is used to indicate that a new dataset should be created.

DESCRIPTION

ERRORS

EXAMPLES

Most of the examples show how to use the CREATE enumeration value.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

dataComponent

PURPOSE

Convert a subclass of DataComponent into DataComponent.

INTERFACE

function arrayDataComponent(array) function columnDataComponent(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  The handle of an array which is to be converted to a DataComponent
- type(ColumnT), intent(in) :: column
  The handle of a column which is to be converted to a DataComponent

RETURNS

- type(DataComponentT)
  The converted object is returned as a handle to a DataComponent object.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example illustrates the use of the dataComponent() function.
! The units of objects with data type BOOLEAN and STRING are meaningless
! and so are not displayed.
subroutine displayUnits( dcomponent )
  use dal

  implicit none

  type(DataComponentT) dcomponent
  integer dattype

  dattype = dataType( dcomponent )
  write(*,*) dattype
  if(dattype.eq.INTEGER8.or.dattype.eq.INTEGER16.or.dattype.eq.INTEGER32 &
    .or.dattype.eq.REAL32.or.dattype.eq.REAL64) then
    write(*,*) units( dcomponent )
  end if

end subroutine displayUnits

program example_datacomponent

  use dal

  implicit none

  type(ArrayT) arr
```

```
  type(BlockT) blk
  type(ColumnT) col
  type(DataSetT) set
  type(TableT) tab
  integer i, j
  integer, dimension(3), parameter :: s = (/ 2,3,4 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))
  arr = addArray(set, "array1", INTEGER16, dimensions=s, units="klm" )
  arr = addArray(set, "array2", INTEGER32, dimensions=s, units="kla" )

  do i = 0, numberOfBlocks( set ) - 1
    blk = block( set, i, READ )
    if( blockType( blk ).eq.ARRAY_BLOCK ) then
      arr = array( set, name( blk ), READ )
      call displayUnits( dataComponent( arr ) )
    else
      tab = table( set, name( blk ) )
      do j = 0, numberOfColumns( tab ) - 1
        col = column( tab, j, READ )
        call displayUnits( dataComponent( col ) )
      end do
    end if
  end do
  call release(set)

end program example_datacomponent
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

DataComponentT

PURPOSE

Used to declare DataComponent handles.

DESCRIPTION

DataComponent is a C++ class whose details are not available to the F90 programmer. Access to this underlying class is achieved through the DataComponentT handle.

ERRORS

EXAMPLES

See dataComponent.

SEE ALSO

The description on handles and the description of class hierarchies etc.

BUGS AND LIMITATIONS

None known.

NAME

`dataSet`(dataSetname, mode, memoryModel)

PURPOSE

Open dataset with the given name.

ARGUMENTS

- character(len=*), intent(in) :: dataSetName
  The name of the dataset.
- integer, intent(in) :: mode
  The access mode which the dataset should be used with. It must be one of the following values:
  - READ Read an existing dataset with the given name. An error is raised if the dataset is not found, or cannot be opened.
  - CREATE Create a new dataset with the given name. If an dataset already exisits with the given name, the behaviour is dependent on the setting of the environment variable SAS_FORMAT. Any changes made to the dataset will be discarded upon closure,
  - MODIFY Open an existing dataset with the given name. All changes made to the dataset will be written saved upon closure.
  - TEMP Open a new dataset. The dataset is discarded upon closure.
- integer, intent(in), optional :: memoryModel
  This specifies a hint to which the memory model should be used. The following values are possible:
  - HIGH_MEMORY
  - HIGH_LOW_MEMORY
  - LOW_MEMORY
  - USE_ENVIRONMENT

RETURNS

- type(DataSetT) :: dataSet
  A handle to the new dataset.

DESCRIPTION

This is a fundamental routine within the DAL API. Virtually all programs requiring the DAL will need to call this function to gain access to a dataset.

ERRORS

EXAMPLES

```
! This examp,e shows how to use the dataset
! function.
program example_dataset

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1", 10, "mm", "attribute comment" )
  call release(set)

  set = dataSet("test.dat",MODIFY)
  call setAttribute(set,"att1", 10, "mm", "attribute comment" )
  call release(set)

  set = dataSet("test.dat",READ)
  write(*,*) "att1 = ", int32Attribute( set, "att1" )
  call release(set)

end program example_dataset
```

SEE ALSO

`clone` `release` `setexists` HIGH_MEMORY HIGH_LOW_MEMORY LOW_MEMORY USE_ENVIRONMENT

BUGS AND LIMITATIONS

None known.

NAME

`dataType`

PURPOSE

Get the data type of an object.

INTERFACE

function arrayDataType(array)
function columnData(column)
function dataComponent(dataComponent)
function attributeDataType(attribute)
function arrayAttributeDataTypeN(array, name)
function arrayAttributeDataTypeR(array, number)
function attributableAttributeDataTypeN(attributable, name)
function attributableAttributeDataTypeR(attributable, number)
function blockAttributeDataTypeN(block, name)
function blockAttributeDataTypeR(block, number)
function columnAttributeDataTypeN(column, name)
function columnAttributeDataTypeR(column, number)
function dataSetAttributeDataTypeN(dataSet, name)
function dataSetAttributeDataTypeR(dataSet, number)
function tableAttributeDataTypeN(table, name)
function tableAttributeDataTypeR(table, number)

ARGUMENTS

- type(ArrayT), intent(in) :: array A handle of the array whose data type is required, or a handle of the array object containing the attribute whose data type is required.
- type(AttributableT), intent(in) :: attributable A handle of the attributable object containing the attribute whose data type is required.
- type(AttributeT), intent(in) :: attribute A handle of the attribute whose data type is required.
- type(BlockT), intent(in) :: block A handle of the block object containing the attribute whose data type is required.
- type(ColumnT), intent(in) :: column A handle of the column whose data type is required, or a handle of the column object containing the attribute whose data type is required.
- type(DataComponentT), intent(in) :: dataComponent A handle of the dataComponent whose data type is required.
- type(DataSetT), intent(in) :: dataSet A handle of the dataset object containing the attribute whose data type is required.
- character(len=*), intent(in) :: name The name of the attribute whose data type is required.
- integer, intent(in) :: number The number of the attribute whose data type is required.
- type(TableT), intent(in) :: table A handle of the table object containing the attribute whose data type is required.

RETURNS

- integer Data type of a column, array, or dataComponent; one of the following enumeration values will be returned: BOOLEAN, INTEGER8, INTEGER16, INTEHER32, REAL32, REAL64, STRING Data type of an attribute: one of the following enumeration values will be returned: INTEGER_ATTRIBUTE, REAL_ATTRIBUTE, STRING_ATTRIBUTE, BOOLEAN_ATTRIBUTE.

DESCRIPTION

This interface is used to get the data type of columns, arrays, dataComponents and attributes. For attributes, the interface allows an attribute object to be used directly, or to specified an attribute, by giving its name or number within an attributable object.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! two 3-dimensional arrays.
!
! It illustrates the use of the derived types DataSetT and ArrayT.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The first array is filled with unique data before the
! dataset is released (closed).
program example_addarray

  use dal
  use errorhandling
```

```
  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_addarray
! This examples show how the column() function is used.
! The column by name is used to get a column and rename it.
! The column by number is used to iterate over all
! columns in the table to output the name, type and units.
program example_column

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"col1",INTEGER32,units="m1",comment="in32 column")
  col = addColumn(tab,"col2",INTEGER32,units="m2",comment="in32 column")
  col = addColumn(tab,"col3",INTEGER32,units="m3",comment="in32 column")
```

```
  col = column( tab, "col2", MODIFY )
  call rename( col, "col4" )

  do i =0, numberOfColumns( tab ) - 1
    col = column( tab, i, READ )
    write(*,*) name( col ), columnDataType( col ), units( col )
  end do

  call release(set)

end program example_column
! This example illustrates the use of the dataComponent() function.
! The units of objects with data type BOOLEAN and STRING are meaningless
! and so are not displayed.
subroutine displayUnits( dcomponent )
  use dal

  implicit none

  type(DataComponentT) dcomponent
  integer dattype

  dattype = dataType( dcomponent )
  write(*,*) dattype
  if(dattype.eq.INTEGER8.or.dattype.eq.INTEGER16.or.dattype.eq.INTEGER32 &
    .or.dattype.eq.REAL32.or.dattype.eq.REAL64) then
    write(*,*) units( dcomponent )
  end if

end subroutine displayUnits

program example_datacomponent

  use dal

  implicit none

  type(ArrayT) arr
  type(BlockT) blk
  type(ColumnT) col
  type(DataSetT) set
  type(TableT) tab
  integer i, j
  integer, dimension(3), parameter :: s = (/ 2,3,4 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))
```

```
  arr = addArray(set, "array1", INTEGER16, dimensions=s, units="klm" )
  arr = addArray(set, "array2", INTEGER32, dimensions=s, units="kla" )

  do i = 0, numberOfBlocks( set ) - 1
    blk = block( set, i, READ )
    if( blockType( blk ).eq.ARRAY_BLOCK ) then
      arr = array( set, name( blk ), READ )
      call displayUnits( dataComponent( arr ) )
    else
      tab = table( set, name( blk ) )
      do j = 0, numberOfColumns( tab ) - 1
        col = column( tab, j, READ )
        call displayUnits( dataComponent( col ) )
      end do
    end if
  end do
  call release(set)

end program example_datacomponent
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

DataSetT

PURPOSE

A derived type which is used to declare DataSet handles.

DESCRIPTION

EXAMPLES

See dataSet.

SEE ALSO

`dataSet`

BUGS AND LIMITATIONS

None known.

NAME

`deleteAttribute`

PURPOSE

Delete an attribute.

INTERFACE

subroutine deleteAttribute(attribute)
subroutine deleteArrayAttributeWithName(array, name)
subroutine deleteArrayAttributeWithNumber(array, attributeNumber)
subroutine deleteAttribAttributeWithName(attributable, name)
subroutine deleteAttribAttributeWithNumber(attributable, attributeNumber)
subroutine deleteBlockAttributeWithName(block, name)

subroutine deleteBlockAttributeWithNumber(block, attributeNumber)
subroutine deleteColumnAttributeWithName(column, name)
subroutine deleteColumnAttributeWithNumber(column, attributeNumber)
subroutine deleteDataSetAttributeWithName(dataSet, name)
subroutine deleteDatSetAttributeWithNumber(dataSet, attributeNumber)
subroutine deleteTableAttributeWithName(table, name) subroutine deleteTableAttributeWithNumber(table, attributeNumber)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  The handle of an array from which the specified attribute should be deleted.
- type(AttributableT), intent(in) :: attributable
  The handle of an attributable from which the specified attribute should be deleted.
- type(AttributeT), intent(in) :: attribute
  The handle of an attribute to be deleted.
- type(AttributableT), intent(in) :: attributeNumber
  The ordinal position ofthe attribute to delete.
- type(BlockT), intent(in) :: block
  The handle of a block from which the specified attribute should be deleted.
- type(ColumnT), intent(in) :: column
  The handle of a column from which the specified attribute should be deleted.
- type(DataSetT), intent(in) :: dataSet
  The handle of a dataset from which the aspecified ttribute should be deleted.
- character(len=*), intent(in) :: name
  The name of the attribute to be deleted.
- type(TableT), intent(in) :: table
  The handle of a table from which the specified attribute should be deleted.

RETURNS None

DESCRIPTION

Delete the given attribute or delete an attribute, with the given name or number, from the specified attributable (or a subclass of attributable) object. In the event that the attribute cannot be deleted an error will be raised. The DataSet, Table, Array, Block, Column and Table types are derived from the Attributble type and hence may contain attributes.

ERRORS

EXAMPLES

```
! This example shows how the deleteAttribute inteface is
! used.
subroutine deleteAllAttributes( attrib )
  use dal
  implicit none

  type(AttributableT) attrib
  type(AttributeT) att
  integer i

  do i = 0, numberOfAttributes( attrib ) - 1
    att = attribute( attrib, 0 )
```

```
    write(*,*) "deleting attribute with name ", name( att )
    call deleteAttribute( att )
  end do

end subroutine deleteAllAttributes

program example_deleteattribute

  use dal
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.true.,"dataset bool comment")
  call setAttribute(set,"sbool3",.false.,"table bool comment")
  call setAttribute(set,"sbool4",.true.,"table bool comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"sbool1",.false.,"table bool comment")
  call setAttribute(tab,"sbool2",.true.,"table bool comment")
  call setAttribute(tab,"sbool3",.false.,"table bool comment")
  call setAttribute(tab,"sbool4",.true.,"table bool comment")

  write(*,*) numberOfAttributes( set )
  call deleteAllAttributes( attributable( set ) )
  write(*,*) numberOfAttributes( set )
  write(*,*) numberOfAttributes( tab )
  call deleteAllAttributes( attributable( tab ) )
  write(*,*) numberOfAttributes( tab )

  call release(set)

end program example_deleteattribute
```

SEE ALSO

`attributble` AttributableT `attribute` AttributeT

BUGS AND LIMITATIONS

None known.

NAME

`deleteBlock`

PURPOSE

Delete a block from a dataset.

INTERFACE

subroutine deleteBlockWithName(dataSet, blockName)
subroutine deleteBlockWithNumber(dataSet, blockNumber)

ARGUMENTS

- character(len=*), intent(in) :: blockName
  The name of the block to be deleted.
- integer, intent(in) :: blockNumber
  The ordinal position of the block to be deleted.
- type(DataSetT), intent(in) :: dataSet
  The handle of a dataset from which the block should be deleted.

RETURNS None

DESCRIPTION
Delete a block wth the given name or number from the specified dataset. In the event that the block cannot be deleted an error will be raised.

ERRORS

EXAMPLES

```
! This example shows how the deleteBlock interface
! is used.
program example_deleteblock

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(BlockT) blk
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table1",10)
  tab = addTable(set,"table2",100)
  tab = addTable(set,"table3",1000)

  write(*,*) numberOfBlocks( set )

  call deleteBlock( set, "table2" );

  do i=0,numberOfBlocks( set ) - 1
    blk = block( set, 0, READ )
    write(*,*) "deleting block with name ", name( blk )
    call deleteBlock( set, 0 )
  end do

  write(*,*) numberOfBlocks( set )

  call release(set)

end program example_deleteblock
```

SEE ALSO
addBlock BlockT

BUGS AND LIMITATIONS

None known.

NAME

`deleteColumn`

PURPOSE

Delete a column from a table.

INTERFACE

subroutine deleteColumnWithName(table, columnName)
subroutine deleteColumnWithNumber(table, columnNumber)

ARGUMENTS

- character(len=*), intent(in) :: columnName
  The name of the column to be deleted.
- integer, intent(in) :: columnNumber
  The ordinal position of the column to be deleted.
- type(TableT), intent(in) :: table
  The handle of a table from which the column should be deleted.

RETURNS None

DESCRIPTION

Delete a column wth the given name or number from the specified table. In the event that the column could not be deleted an error is raised.

ERRORS

EXAMPLES

```
program example_addcolumn

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))

  call deleteColumn( tab, "int32" )
```

```
  call deleteColumn( tab, 3 ) ! "real32"

  do i = 0, numberOfColumns( tab ) - 1
    write(*,*) name( column( tab, i, READ ) )
  end do

  call release(set)

end program example_addcolumn
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`deleteRows`(table, from, count)

PURPOSE

Delete a range of rows from a table.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table within which the specified range of rows should be deleted.
- integer, intent(in) :: from
  The first row number of the range. 0 ¡= from ¡ n, where n is the number of rows in the table.
- integer, intent(in), optional :: count
  The number of rows in the range. 0 ¡= count ¡ n ,where n is the number of rows in the table.

RETURNS None

DESCRIPTION

This operation is very expensive. The range is specified with couple [from,count], where from + count ¡= n, where n is the number of rows in the table. It should be carefully noted that any data pointers (to columns in this table) which are currently active will become stale after deleteRows() has been called.

ERRORS

EXAMPLES

```
! This examples show how to use the deleteRows() subroutine.
program example_deleterows

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
```

```
integer(kind=int32), dimension(:), pointer :: i32
real(kind=single), dimension(:), pointer :: r32
integer i, r

set = dataSet("test.dat",CREATE)
tab = addTable(set,"some table",10)

col1 = addColumn(tab,"col1",INTEGER32,units="m",comment="in32 column")
i32 => int32Data(col1)

do i=0,4
  i32(i) = 3*i
end do

call release( col1)

col2 = addColumn(tab,"col2",REAL32,units="Dm",comment="real32 column")
r32 => real32Data(col2)

do i=0,4
  r32(i) = 0.5*i
end do

call release( col2)

call copyRows( tab, 0, 5, 5 ) ! copy range [0,4] to [5,9]

i32 => int32Data(col1)
r32 => real32Data(col2)

do i = 0, numberOfRows( tab ) - 1
  write(*,*) i32(i), r32(i)
end do

call release(col1)
call release(col2)

r = 0
do i = 0, 9
  i32 => int32Data(col1)
  if( i32(r) .eq. 6 ) then
    write(*,*) "deleting row number ", i
    call deleteRows( tab, r, 1 )
  else
    r = r + 1
  end if
  call release( col1 )
end do

i32 => int32Data(col1)
r32 => real32Data(col2)
do i = 0, numberOfRows( tab ) - 1
  write(*,*) i32(i), r32(i)
end do
```

```
  call release(set)

end program example_deleterows
```

SEE ALSO

`copyRows` `insertRows`

BUGS AND LIMITATIONS

None known.

NAME

`discardDataSet`

PURPOSE

Tells the data set server object to discard the named data set.

ARGUMENTS

- character(len=*), intent(in) :: dataSetName
  The name of the dataset.

RETURNS

None

DESCRIPTION

The named data set is released from memory.

This subroutine must only be called by Meta Tasks.

ERRORS

EXAMPLES

```
! This example shows how to use the keepDataSet
! subroutine
program example_keepdiscarddataset

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet("test.dat",CREATE)
  call release(set)   ! The dataset will be released from memory

  call keepDataSet("test.dat")   ! Tell the dataset server not to discard
     ! the dataset with name "test.dat"

  set = dataSet("test.dat",READ)
  call release(set)   ! The dataset will not be released from memory

  set = dataSet("test.dat",READ)  ! The dataset is already in memory, so this
     ! operation has virtually no overhead.
```

```
  call release(set)   ! The dataset will not be released from memory
  call discardDataSet("test.dat") ! Tell the dataset server to discard and
     ! release the dataset with name "test.dat"

end program example_keepdiscarddataset
```

SEE ALSO

`keepDataSet`

BUGS AND LIMITATIONS

None known.

NAME

`dimensions`

PURPOSE

Get the dimensions of an array or a column.

INTERFACE

function dimensionsOfArray(array)
function dimensionsOfColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  The handle of an array from which the dimensions are to be retrieved.
- type(ColumnT), intent(in) :: column
  The handle of a column from which the dimensions are to be retrieved.

RETURNS

- integer, dimension(:), pointer
  The number of elements in the returned vector gives the rank of the objects data. Each element of the returned vector gives the length of each dimension.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example demonstrates the dimensions inteferface.

subroutine fillWithData( dataSetName )

  use dal

  implicit none

  character(len=*), intent(in) :: dataSetName
  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer(kind=INT32), dimension(:,:,:,:,:), pointer :: c
  integer, dimension(:), pointer :: s
```

```
  integer :: i,j,k,l,m,n

  ! Reopen dataset and fill with data.
  set = dataSet( dataSetName, MODIFY )
  tab = table( set, "table" )
  col = column( tab, "column", MODIFY )
  s => dimensions( col )
  c => int32Array4Data( col )

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0, s(3) - 1
      do k=0, s(2) - 1
        do j=0, s(1) - 1
          do i=0, s(0) - 1
            c(i,j,k,l,m) = n
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col)
  call release(set)

end subroutine fillWithData

program example_dimensions

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col = addColumn( tab, "column", INTEGER32, "km", s, "column comment" )
  call release( set )

  call fillWithData( "test.dat" )

end program example_dimensions
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

DOUBLE

PURPOSE

An enumeration value which is used to indicate real data of double precision.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`forEachBlock`(dataSet, callThisFunction)

PURPOSE

Block iteration.

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet
  The handle of a dataset for which block iteration is to be carried out.
- interface subroutine callThisFunction(block) type(BlockT), intent(in) :: block end subroutine callThisFunction end interface
  The iterating function to be called for each block in the dataset. The block is passed by handle to the iterating function.

RETURNS None

DESCRIPTION

Call the specified subroutine for each block, in turn, in the specified dataset. If the dataset has no blocks, no iteration will be attempted. Each block is passed to the iterating subroutine as a block-handle.

ERRORS

EXAMPLES

```
! In this example, a dataset is created with one table and one
! array.
! The generic subroutine displayBlock, which operates on the
! BlockT base type. The blockType() function operates on objects
! of type BlockT.
! The example also shows blocks being retrieved from the dataset
! both by name and by number.
subroutine displayBlock( thisBlock )
  use dal

  implicit none

  type(BlockT), intent(in) ::  thisBlock
```

```
  write(*,*) "The block with name ", name( thisBlock )

  if( blockType( thisBlock ) .eq. ARRAY_BLOCK ) then
    write(*,*) " is an array."
  end if

  if( blockType( thisBlock ) .eq. TABLE_BLOCK ) then
    write(*,*) " is a table."
  end if

end subroutine displayBlock

subroutine displayBlocks( thisSet )
  use dal

  implicit none

  type(DataSetT) thisSet
  integer i

  interface
    subroutine displayBlock( blk )
      use dal
      implicit none
      type(BlockT), intent(in) :: blk

    end subroutine displayBlock
  end interface
  call foreachblock( thisSet, displayBlock )

end subroutine displayBlocks

program example_foreachblock

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table",10);
  arr = addArray(set, "array", INTEGER32, dimensions=s )

  call displayBlock( block( tab ) )
  call displayBlock( block( arr ) )
  call displayBlock( block( set, "table", READ ) )
  call displayBlock( block( set, "array", READ ) )
  call displayBlocks( set )

  call release(set)
```

```
end program example_foreachblock
```

SEE ALSO
BlockT

BUGS AND LIMITATIONS

None known.

NAME
`forEachColumn`(table, callThisFunction)

PURPOSE
Column iteration.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table fro which column iteration is to be carried out.
- interface subroutine callThisFunction(column) type(TableT), intent(in) :: column end subroutine callThisFunction end interface
  The iterating function to be called for each column in the table. Each column is passed to the iterating subroutine as a column-handle.

RETURNS None

DESCRIPTION

ERRORS

EXAMPLES

```
! This examples show how the forEachColumn() function is used.
! The column by name is used to get a column and rename it.
! The column by number is used to iterate over all
! columns in the table to output the name, type and units.
subroutine displayColumn( col )

  use dal

  implicit none

  type(ColumnT), intent(in) :: col

  write(*,*) name( col ), columnDataType( col ), units( col )

end subroutine displayColumn

program example_foreachcolumn

  use dal

  implicit none

  type(DataSetT) set
```

```
  type(TableT) tab
  type(ColumnT) col
  integer i

  interface
    subroutine displayColumn( col )
      use dal
      implicit none

      type(ColumnT), intent(in) :: col
    end subroutine displayColumn
  end interface

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"col1",INTEGER32,units="m1",comment="in32 column")
  col = addColumn(tab,"col2",INTEGER32,units="m2",comment="in32 column")
  col = addColumn(tab,"col3",INTEGER32,units="m3",comment="in32 column")

  col = column( tab, "col2", MODIFY )
  call rename( col, "col4" )

  call forEachColumn( tab, displayColumn )
  call release(set)

end program example_foreachcolumn
```

SEE ALSO

ColumnT

BUGS AND LIMITATIONS

None known.

NAME

`forEachSubTable`(table, callThisFunction)

PURPOSE

Subtable iteration.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table for which subtable iteration is to be carried out.
- interface subroutine callThisFunction(subTable) type(TableT), intent(in) :: subTable end subroutine callThisFunction end interface
  The iterating subroutine to be called for each subtable of the table. The iterating subroutine is passed the subtable as a subtable-hnadle. The table size is dependent on the model. In, the High Memory Mode, the entire table is passed to the iterating function, which therefore is called only once. In the Low Memory Mode, the table size defaults to 1 row, but may be changed by setting the environment variable SAS_ROWS to the required table size. The Memory Model is set with the environment variable SAS_MEMORY_MODEL.

RETURNS None

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the forEachSubTable() function
! is used.
! This subroutine will fill the subtable with dummy data.
subroutine fill(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: xCol, yCol, tCol
  real(kind=SINGLE), dimension(:), pointer :: x,y,t

  write(*,*) from( tab ), count( tab )

  xCol = column(tab,"x",MODIFY)
  yCol = column(tab,"y",MODIFY)
  tCol = column(tab,"t",MODIFY)
  x => real32Data(xCol)
  x = 1.23
  write(*,*) x
  y => real32Data(yCol)
  y = 2.34
  write(*,*) y
  t => real32Data(tCol)
  t = 3.45
  write(*,*) t

end subroutine fill

! This subroutine will write the contents of the subtable to standard output.
subroutine check(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: xCol, yCol, tCol
  real(kind=single), dimension(:), pointer :: x,y,t

  write(*,*) from( tab ), count( tab )

  xCol = column(tab,"x",READ)
  yCol = column(tab,"y",READ)
  tCol = column(tab,"t",READ)
  x => real32Data(xCol)
  y => real32Data(yCol)
  t => real32Data(tCol)
  write(*,*) "DATA:", x, y, t

end subroutine check

program example_foreachsubtable
```

```
  use dal

  implicit none

  ! This part of the program will apply reportX to a table.
  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: xCol, yCol, tCol
  real(kind=SINGLE), dimension(:), pointer :: x,y,t

  interface
    subroutine fill( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine fill
    subroutine check( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine check
  end interface

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"events",10)
  xCol = addColumn(tab,"x",real32,"mm")
  yCol = addColumn(tab,"y",real32,"mm")
  tCol = addColumn(tab,"t",real32,"s")

  call forEachSubTable(tab,fill)
  call forEachSubTable(tab,check)

  call release(set)

end program example_foreachsubtable
```

SEE ALSO

`foreachBlock` `forEachColumn` `forEachRow` SubtableT

BUGS AND LIMITATIONS

None known.

NAME

`forEachRow`(table, fn)

PURPOSE

Row iteration.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table for which subtable iteration is required.
- interface subroutine fn(r) type(RowT), intent(in) :: r end subroutine end interface
  The iterating subroutine which be will be called for each row in the table; the row being passed as a row-handle.

RETURNS None

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the forEachSubTable() function
! is used.
! This subroutine will fill the subtable with dummy data.
subroutine fill(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: xCol, yCol, tCol
  real(kind=SINGLE), dimension(:), pointer :: x,y,t

  xCol = column(tab,"x",MODIFY)
  yCol = column(tab,"y",MODIFY)
  tCol = column(tab,"t",MODIFY)
  x => real32Data(xCol)
  x = 1.23
  y => real32Data(yCol)
  y = 2.34
  t => real32Data(tCol)
  t = 0

end subroutine fill

! This subroutine will write the contents of the subtable to standard output.
subroutine check(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: xCol, yCol, tCol
  real(kind=single), dimension(:), pointer :: x,y,t

  xCol = column(tab,"x",READ)
  yCol = column(tab,"y",READ)
  tCol = column(tab,"t",READ)
  x => real32Data(xCol)
  y => real32Data(yCol)
  t => real32Data(tCol)
  write(*,*) x, y, t

end subroutine check

program example_foreachsubtable

  use dal
```

```
  implicit none

  ! This part of the program will apply reportX to a table.
  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: xCol, yCol, tCol
  real(kind=SINGLE), dimension(:), pointer :: x,y,t

  interface
    subroutine fill( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine fill
    subroutine check( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine check
  end interface

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"events",10)
  xCol = addColumn(tab,"x",real32,"mm")
  yCol = addColumn(tab,"y",real32,"mm")
  tCol = addColumn(tab,"t",real32,"s")

  call forEachRow(tab,fill)
  call forEachRow(tab,check)

  call release(set)

end program example_foreachsubtable
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`from`

PURPOSE

Get the from-value from the seek range of an object.

INTERFACE

function fromColumn(column) function fromTable(table)

ARGUMENTS

- type(ColumnT), intent(in) :: column
- type(TableT), intent(in) :: table

RETURNS

- integer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the seek functions
! are used.
! This subroutine will dispaly the seek values of the given table and column.
subroutine whatisseek(tab)
  use dal

  type(TableT), intent(in) :: tab

  type(ColumnT) :: col

  write(*,*) from( tab ), count( tab )

  col = column(tab,"x",MODIFY)
  write(*,*) from( col ), count( col )

end subroutine whatisseek

program example_seek

  use dal

  implicit none

  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: col

  interface
    subroutine whatisseek( subtab )
      use dal
      implicit none
      type(TableT), intent(in) :: subtab
    end subroutine whatisseek
  end interface

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"events",10)
  col = addColumn(tab,"x",real32,"mm")

  call forEachSubTable(tab,whatisseek)

  call release(set)

end program example_seek
```

SEE ALSO

count

BUGS AND LIMITATIONS

None known.

NAME

`hasAttribute`

PURPOSE

Determine if an attribute with a given name exists.

INTERFACE

function arrayHasAttribute(array, name)
function attributableHasAttribute(attributable, name)
function blockHasAttribute(block, name)
function columnHasAttribute(column, name)
function dataSetHasAttribute(dataSet, name)
function tableHasAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  The handle of an array which is to be tested for the existence of the attribute.
- type(AttributableT), intent(in) :: attributable
  The handle of an attributable which is to be tested for the existence of the attribute.
- type(BlockT), intent(in) :: block
  The handle of a block which is to be tested for the existence of the attribute.
- type(ColumnT), intent(in) :: column
  The handle of a column which is to be tested for the existence of the attribute.
- type(DataSetT), intent(in) :: dataSet
  The handle of a dataset which is to be tested for the existence of the attribute.
- character(len=*), intent(in) :: name
  The name of the attribute.
- type(TableT), intent(in) :: table
  The handle of a table which is to be tested for the existence of the attribute.

RETURNS

- logical

DESCRIPTION

Determine if an attribute with the given name exists within the given attributable set.

ERRORS

EXAMPLES

```
! This example shoes how the hasAttribute interface is used.
program example_hasattribute

  use dal
  implicit none

  type(DataSetT) set
  type(AttributeT) att
```

```
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")

  if( hasAttribute( set, "sbool2" ) ) then
    write(*,*) 'That is not possible'
  end if

  if( hasAttribute( set, "sbool1" ) ) then
    att = attribute( set, "sbool1" )
    write(*,*) name( att ), " = ", booleanAttribute( att )
  end if

  call release(set)

end program example_hasattribute
```

SEE ALSO

AttributableT AttributeT

BUGS AND LIMITATIONS

None known.

NAME

`hasBlock`(set, name)

PURPOSE

Determine if a block with a given name exists.

ARGUMENTS

- type(DataSetT), intent(in) :: set
  The handle of the dataset which is to be examined for the existence of the named block.
- character(len=*), intent(in) :: name
  The name of the block.

RETURNS

- logical

DESCRIPTION

Block names are unique within a dataset, so there can never be more than one block with the given name. If a block with the given name is not found, false is returned, otherwise true is returned.

ERRORS

EXAMPLES

```
! This example showas how the hasBlock() function is used.
! In the example, a dataset is created with one table and one
! array.
! The generic subroutine displayBlock, which operates on the
! BlockT base type. The blockType() function operates on objects
! of type BlockT.
```

```
! The dataset is testes for the existence of the table and the array, and in
! each case, the block is displayed.
subroutine displayBlock( thisBlock )
  use dal

  implicit none

  type(BlockT) thisBlock

  write(*,*) "The block with name ", name( thisBlock )

  if( blockType( thisBlock ) .eq. ARRAY_BLOCK ) then
    write(*,*) " is an array."
  end if

  if( blockType( thisBlock ) .eq. TABLE_BLOCK ) then
    write(*,*) " is a table."
  end if

end subroutine displayBlock

program example_hasblock

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table",10);
  arr = addArray(set, "array", INTEGER32, dimensions=s )

  call release(set)

  set = dataSet("test.dat",READ)

  if( hasBlock( set, "table" ) ) then
    call displayBlock( block( set, "table",READ ) )
  end if

  if( hasBlock( set, "array" ) ) then
    call displayBlock( block( set, "array",READ ) )
  end if

  call release(set)

end program example_hasblock
```

SEE ALSO

BlockT DataSetT

BUGS AND LIMITATIONS

None known.

NAME

`hasColumn`(table, name)

PURPOSE

Determine if a column with a given name exists.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of the table which is to be examined for the existence of the named column.
- character(len=*), intent(in) :: name
  The name of the column.

RETURNS

- logical

DESCRIPTION

Determine if a column with a given name exists within the given table. Column names are unique, within a table, so there can never be more than one column with the given name. False is returned if a column the given name is not found, otherwise true is returned.

ERRORS

EXAMPLES

```
! This examples show how the hasColumn() function is used.
! The column by name is used to get a column and rename it.
! The column by number is used to iterate over all
! columns in the table to output the name, type and units.
program example_hascolumn

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"col1",INTEGER32,units="m1",comment="in32 column")
  col = addColumn(tab,"col2",INTEGER32,units="m2",comment="in32 column")
  col = addColumn(tab,"col3",INTEGER32,units="m3",comment="in32 column")

  col = column( tab, "col2", MODIFY )
  call rename( col, "col4" )

  if( hasColumn( tab, "col2" ) ) then
    write(*,*) 'This is not possible, since col4 was renamed to col4'
```

```
  end if

  do i =0, numberOfColumns( tab ) - 1
    col = column( tab, i, READ )
    write(*,*) name( col ), columnDataType( col ), units( col )
  end do

  call release(set)

end program example_hascolumn
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`hasNulls`

PURPOSE

Determines if an object contains any null values.

INTERFACE

function hasNullArray(array)
function hasNullColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the values to be checked.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the values to be checked.
- integer(kind=INT32), intent(in) :: position

RETURNS

- logical
  True, if a null value was found, false otherwise.

DESCRIPTION

This routine searches for null values in the specified object (a column or an array). Note that if the LMM is being used, the object's data is first loaded into memory before the cjheck is carried out, and is then released again.

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )
```

```
end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
       ! Would have given an error, if the null
       ! value of array arr1 had not been set.

  if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
    write(*,*) "Using null value of arr1, in arr2"
    call setNullValue( arr2, intNullValue( arr1 ))
  else
    call setNullValue( arr2, 999999 )
  end if

  call check( nullable( arr2 ) )
```

```
  call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

  call release(arr1)
  call release(arr2)

  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`intNullValue` `isNotNull` `isNull` `nullable` `nullDefined` `nullType` `setNullValue` `setToNull`

BUGS AND LIMITATIONS

None known.

NAME

hasScaling

PURPOSE

THIS INTERFACE IS NOT IMPLEMENTED. Determine if scaling factors have been set for an array or a column.

INTERFACE

function hasScalingOfArray(array)
function hasScalingOfColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column

RETURNS

- logical

DESCRIPTION

N/A

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

HIGH_MEMORY

PURPOSE

An enumeration value used to specify that the High Memory Model should be used to open a dataset.

DESCRIPTION

This is a hint only, and may be overridden using an environment variable setting.

EXAMPLES

```
! This examp,e shows how to open a dataset
! with a specific memory model.
program example_memorymodel

  use dal

  implicit none

  type(DataSetT) set
```

```
  set = dataSet( "test.dat",CREATE,HIGH_MEMORY )
  call release( set )

  set = dataSet( "test.dat",HIGH_LOW_MEMORY )
  call release( set )

end program example_memorymodel
```

SEE ALSO

HIGH_LOW_MEMORY LOW_MEMORY

BUGS AND LIMITATIONS

None known.

NAME

HIGH_LOW_MEMORY

PURPOSE

An enumeration value used to specify that the highlow memory model should be used to open a dataset.

DESCRIPTION

This is a hint only, and may be overridden using an environment variable setting.

EXAMPLES

```
! This examp,e shows how to open a dataset
! with a specific memory model.
program example_memorymodel

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet( "test.dat",CREATE,HIGH_MEMORY )
  call release( set )

  set = dataSet( "test.dat",HIGH_LOW_MEMORY )
  call release( set )

end program example_memorymodel
```

SEE ALSO

HIGH_MEMORY

BUGS AND LIMITATIONS

None known.

NAME

`insertRows`(table, position, count)

PURPOSE

Insert a range of rows in a table.

ARGUMENTS

- type(TableT), intent(in) :: table
  The handle of a table within which the specified range of rows should be inserted.
- integer, intent(in) :: position
  This specifies at which row to insert the range of rows, which must be in the range 0 to n, where n is the number of rows in the table.
- integer, intent(in), optional :: count
  The nunber of rows to be inserted.

RETURNS None

DESCRIPTION
This operation is very expensive and should be used minimally.

ERRORS

EXAMPLES

```
! This examples show how to use the insertRows() subroutine.
program example_insertrows

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=single), dimension(:), pointer :: r32
  integer i, r

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",5)

  col1 = addColumn(tab,"col1",INTEGER32,units="m",comment="in32 column")
  i32 => int32Data(col1)

  do i=0,4
    i32(i) = 3*i
  end do

  call release( col1)

  col2 = addColumn(tab,"col2",REAL32,units="Dm",comment="real32 column")
  r32 => real32Data(col2)

  do i=0,4
    r32(i) = 0.5*i
  end do

  call release( col2)

  ! insert 5 additional rows, at the end of the table
```

```
  call insertRows( tab, 5, 5 )

  ! copy the first 5 rows to the new rows.
  call copyRows( tab, 0, 5, 5 ) ! copy range [0,4] to [5,9]

  i32 => int32Data(col1)
  r32 => real32Data(col2)

  do i = 0, numberOfRows( tab ) - 1
    write(*,*) i32(i), r32(i)
  end do

  call release(col1)
  call release(col2)

  r = 0
  do i = 0, 9
    i32 => int32Data(col1)
    if( i32(r) .eq. 6 ) then
      write(*,*) "deleting row number ", i
      call deleteRows( tab, r, 1 )
    else
      r = r + 1
    end if
    call release( col1 )
  end do

  i32 => int32Data(col1)
  r32 => real32Data(col2)

  do i = 0, numberOfRows( tab ) - 1
    write(*,*) i32(i), r32(i)
  end do


  call release(set)

end program example_insertrows
```

SEE ALSO

copyRows deleteRows

BUGS AND LIMITATIONS

None known.

NAME

INT8

PURPOSE

An enumeration value which is used to indicate int8 data.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array2Data`

PURPOSE

Get the int8 data from an array or column cell containing 2-dimensional array data.

INTERFACE

function int8ColumnArray2DataElement(column, row)
function int8ArrayArray2Data(array)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array for which the data is to be retrieved.
- type(ColumnT), intent(in) :: column
  A handle of the column for which the data is to be retrieved.
- integer, intent(in) :: row
  The column row number (cell number) for which the data is to be retrieved.

RETURNS

- integer(kind=INT8), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
```

```
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
    c1 => int8Array2Data(col1,k)
    c2 => int8Array2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = n
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray2data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT8), dimension(:,:), pointer :: a1, a2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
```

```
  integer :: i,j,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER8, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER8, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int8Array2Data(arr1)
  a2 => int8Array2Data(arr2)
  do j=0,3
    do i=0,2
      a1(i,j) = n
      a2(i,j) = a1(i,j)
      n = n + 1
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array2Data`

PURPOSE

Get the int8 data from a column containing 2-dimensional array data.

INTERFACE

function int8ColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.

RETURNS

- integer(kind=INT8), dimension(:,:,:), pointer
  The 2-dimensional data is returned as a 3-dimensional array.

DESCRIPTION

The data is returned as a 3-dimensional array, since the column's data is arranged as a vector of 2-dimensional elements. The column should be released after the data is no longer required.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int8Array2Data(col1)
  c2 => int8Array2Data(col2)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = n
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array3Data`

PURPOSE

Get the int8 data from an array or column cell containing 3-dimensional array data.

INTERFACE

function int8ColumnArray3DataElement(column, row)
function int8ArrayArray3Data(array)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array which contains the data to be retrieved.
- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be retrieved.
- integer, intent(in) :: row
  The column row number (cell number) for which the data is to be retrieved.

RETURNS

- integer(kind=INT8), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:), pointer :: c1, c2
```

```
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => int8Array3Data(col1,l)
    c2 => int8Array3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = n
          c2(i,j,k) = c1(i,j,k)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT8), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,n
```

```
  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER8, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER8, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int8Array3Data(arr1)
  a2 => int8Array3Data(arr2)
  do k=0,4
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array3Data`

PURPOSE

Get the int8 data from a column containing 3-dimensional array data.

INTERFACE

function int8ColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be retrieved.

RETURNS

- integer(kind=INT8), dimension(:,:,:,:), pointer
  The 2-dimensional data is returned as a 4-dimensional array.

DESCRIPTION

The data is returned as a 4-dimensional array, since the column's data is arranged as a vector of 3-dimensional elements.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int8Array3Data(col1)
  c2 => int8Array3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = n
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array4Data`

PURPOSE

Get the int8 data from a column cell containing 4-dimensional array data.

INTERFACE

function int8ColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be retrieved.
- integer, intent(in) :: row
  The column row number (cell number) which contains the data to be retrieved.

RETURNS

- integer(kind=INT8), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n
```

```
  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int8Array4Data(col1,m)
    c2 => int8Array4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = n
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Array4Data`

PURPOSE

Get the int8 data from a column containing 4-dimensional array data.

INTERFACE

function int8ColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the data is to be retrieved.

RETURNS

- integer(kind=INT8), dimension(:,:,:,:,:), pointer
  The 5-dimensional data is returned as a 4-dimensional array.

DESCRIPTION

The data is returned as a 5-dimensional array, since the column's data is arranged as a vector of 4-dimensional elements.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int8Array4Data(col1)
  c2 => int8Array4Data(col1)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
```

```
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8Attribute`

PURPOSE

Get the value of an attribute as an int8.

INTERFACE

function int8ArrayAttribute(array, name)
function int8AttributableAttribute(attributable, name)
function int8Attribute(attribute)
function int8BlockAttribute(Block, name)
function int8ColumnAttribute(column, name)
function int8DataSetAttribute(dataSet, name)
function int8TableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required attribute.
- type(AttributableT), intent(in) :: attributable
  A handle of the attributable containing the required attribute.
- type(AttributeT), intent(in) :: attribute
  A handle of the attribute.
- type(BlockT), intent(in) :: block
  A handle of the block containing the required attribute.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required attribute.
- type(DataSetT), intent(in) :: dataSet
  A handle of the column containing the required attribute.
- character(len=*), intent(in) :: name
  The name of the required attribute.
- type(TableT), intent(in) :: table
  A handle of the table containing the required attribute.

RETURNS

- integer(kind=INT8)
  The attribute's internal value is returned as an int8-integer (type conversion taking place, if possible, as necessary).

DESCRIPTION

In the event that the attribute's value cannot be type converted an error is raised.

ERRORS

EXAMPLES

```
! This example shows how int8 attributes are used.
! The program creates a dataset containing two int8 attributes,
! together with a table containing two int8 attributes.
! The attributes are then accessed, by name, with
! the int8Attribute() function.
! Also, it is shown how to access the attributes by position.
program example_int8attribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"int1",1,"int comment")
  call setAttribute(set,"int2",2,"int comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"int1",3,"int comment")
  call setAttribute(tab,"int2",4,"int comment")

  write(*,*) int8Attribute( set, "int1" ) ! output '1'
  write(*,*) int8Attribute( set, "int2" ) ! output '2'
  write(*,*) int8Attribute( tab, "int1" ) ! output '3'
  write(*,*) int8Attribute( tab, "int2" ) ! output '4'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) int8Attribute( att ) ! output the sequence 1, 2
  end do

  call release(set)

end program example_int8attribute
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

int8Data

PURPOSE

Get the int8 data from an array, column or column cell.

INTERFACE

function int8ArrayData(array)
function int8ColumnData(column)
function int8ColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT8), dimension(:), pointer
  The data is returned as a flat vector regardless of the dimensionality of the data.

DESCRIPTION

The data is returned in a vector regardles of the dimensionality of the data. In particular, when accessing a scalar column cell, a vector of length 1 is returned, which contains the single scalar value.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_int8data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:,:,:,:), pointer :: c1, c2
  integer(kind=INT8), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
```

```
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int8Array4Data(col1)
  c2 => int8Array4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
  cd => int8Data( col2 ) ! Access the column's 4-dimensional data as a flat vector.

  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_int8data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8VectorData`

PURPOSE

Get the int8 data from an array or column cell containing vector data.

INTERFACE

function int8ArrayVectorData(array)
function int8ColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- integer(kind=INT8), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int8VectorData(col1,m)
    c2 => int8VectorData(col2,m)
```

```
    do i=0,2
      c1(i) = n
      c2(i) = c1(i)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellvectordata
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The array is then initialised,
program example_arrayvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT8), dimension(:), pointer :: a1, a2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER8, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER8, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int8VectorData(arr1)
  a2 => int8VectorData(arr2)
  do i=0,2
    a1(i) = n
    a2(i) = a1(i)
    n = n + 1
  end do

  call release(arr1)
  call release(arr2)
  call release(set)
```

```
end program example_arrayvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int8VectorData`

PURPOSE

Get the int8 data from a column containing vector data.

INTERFACE

function int8ColumnVectorData(column) result(ptr)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

RETURNS

- integer(kind=INT8), dimension(:,:), pointer

DESCRIPTION

The column must contain vector data.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second column has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT8), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n
```

```
  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 10, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER8, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int8VectorData(col1)
  c2 => int8VectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
      c1(i,m) = n
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

INT16

PURPOSE

An enumeration value which is used to indicate that integer16 data is being used.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Array2Data`

PURPOSE

Get the int16 data from an array or column cell containing 2-dimensional array data.

INTERFACE

function int16ArrayArray2Data(array)
function int16ColumnArray2DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array which contains the data to be accessed.
- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.
- integer, intent(in) :: row
  The number of the column cell which contains the data to be accessed.

RETURNS

- integer(kind=INT16), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the int16Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
```

```
    c1 => int16Array2Data(col1,k)
    c2 => int16Array2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = n
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray2data
! This example shows how to use the int16Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT16), dimension(:,:), pointer :: a1, a2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER16, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER16, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int16Array2Data(arr1)
  a2 => int16Array2Data(arr2)
  do j=0,3
    do i=0,2
      a1(i,j) = n
      a2(i,j) = a1(i,j)
```

```
      n = n + 1
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

int16Array2Data

PURPOSE

Get the int16 data from a column containing 2-dimensional array data.

INTERFACE

function int16ColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column which contains the data to be accessed.

RETURNS

- integer(kind=INT16), dimension(:,:,:), pointer
  The 2-dimensional data is returned as a 3-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int16Array2Data(col1)
  c2 => int16Array2Data(col2)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = n
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Array3Data`

PURPOSE

Get the int16 data from an array or column cell containing 3-dimensional array data.

INTERFACE

function int16ArrayArray3Data(array)
function int16ColumnArray3DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT16), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => int16Array3Data(col1,l)
    c2 => int16Array3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = n
          c2(i,j,k) = c1(i,j,k)
```

```
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT16), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER16, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER16, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int16Array3Data(arr1)
  a2 => int16Array3Data(arr2)
  do k=0,4
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k)
        n = n + 1
      end do
    end do
  end do
```

```
  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Array3Data`

PURPOSE

Get the int16 data from a column containing 3-dimensional array data.

INTERFACE

function int16ColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- integer(kind=INT16), dimension(:,:,:,:), pointer
  The 3-dimensional data is returned as a 4-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
```

```
  integer(kind=INT16), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int16Array3Data(col1)
  c2 => int16Array3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = n
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Array4Data`

PURPOSE

Get the int16 data from a column cell containing 4-dimensional array data.

INTERFACE

function int16ColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT16), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int16Array4Data(col1,m)
    c2 => int16Array4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = n
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
```

```
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Array4Data`

PURPOSE

Get the int16 data from a column containing 4-dimensional array data.

INTERFACE

function int16ColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

RETURNS

- integer(kind=INT16), dimension(:,:,:,:,:), pointer
  The 4-dimensional column data is returned as a 5-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array4data

  use dal
  use errorhandling

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int16Array4Data(col1)
  c2 => int16Array4Data(col1)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Attribute`

PURPOSE

Get the value of an attribute as an int16.

INTERFACE

function int16ArrayAttribute(array, name)
function int16AttributableAttribute(attributable, name)
function int16Attribute(attribute)
function int16BlockAttribute(Block, name)

function int16ColumnAttribute(column, name)
function int16DataSetAttribute(dataSet, name)
function int16TableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required attribute.
- type(AttributableT), intent(in) :: attributable
  A handle of the attributable containing the required attribute.
- type(AttributeT), intent(in) :: attribute
  A handle of the attribute.
- type(BlockT), intent(in) :: block
  A handle of the block containing the required attribute.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required attribute.
- type(DataSetT), intent(in) :: dataSet
  A handle of the dataset containing the required attribute.
- character(len=*), intent(in) :: name
  The name of the required attribute.
- type(TableT), intent(in) :: table
  A handle of the table containing the required attribute.

RETURNS

- integer(kind=INT16)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how int16 attributes are used.
! The program creates a dataset containing two int16 attributes,
! together with a table containing two int16 attributes.
! The attributes are then accessed, by name, with
! the int16Attribute() function.
! Also, it is shown how to access the attributes by position.
program example_int16attribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"int1",1,"int comment")
  call setAttribute(set,"int2",2,"int comment")
```

```
  tab = addTable(set,"table",10);
  call setAttribute(tab,"int1",3,"int comment")
  call setAttribute(tab,"int2",4,"int comment")

  write(*,*) int16Attribute( set, "int1" ) ! output '1'
  write(*,*) int16Attribute( set, "int2" ) ! output '2'
  write(*,*) int16Attribute( tab, "int1" ) ! output '3'
  write(*,*) int16Attribute( tab, "int2" ) ! output '4'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) int16Attribute( att ) ! output the sequence 1, 2
  end do

  call release(set)

end program example_int16attribute
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16Data`

PURPOSE

Get the int16 data from an array, column or column cell.

INTERFACE

function int16ArrayData(array)
function int16ColumnData(column)
function int16ColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT8), dimension(:), pointer
  The data is returned as a flat vector regardless of the dimensionality of the data.

DESCRIPTION

The data is returned in a vector regardles of the dimensionality of the data. In particular, when accessing a scalar column cell, a vector of length 1 is returned, which contains the single scalar value.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_int16data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:,:,:,:), pointer :: c1, c2
  integer(kind=INT16), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int16Array4Data(col1)
  c2 => int16Array4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
```

```
  cd => int16Data( col2 ) ! Access the column's 4-dimensional data as a flat vector.

  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_int16data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16VectorData`

PURPOSE

Get the int16 data from an array or column cell containing vector data.

INTERFACE

function int16ArrayVectorData(array)
function int16ColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- integer(kind=INT16), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
```

```
! before the dataset is released (closed).
program example_cellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int16VectorData(col1,m)
    c2 => int16VectorData(col2,m)
    do i=0,2
      c1(i) = n
      c2(i) = c1(i)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellvectordata
! This example shows how to use the int16Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The array is then initialised,
program example_arrayvectordata

  use dal
  use errorhandling

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT16), dimension(:), pointer :: a1, a2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER16, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER16, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int16VectorData(arr1)
  a2 => int16VectorData(arr2)
  do i=0,2
    a1(i) = n
    a2(i) = a1(i)
    n = n + 1
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int16VectorData`

PURPOSE

Get the int16 data from a column containing vector data.

INTERFACE

function int16ColumnVectorData(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- integer(kind=INT16), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second column has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT16), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 10, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER16, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int16VectorData(col1)
  c2 => int16VectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
      c1(i,m) = n
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

INT32

PURPOSE

An enumeration value which is used to indicate that integer32 data is being used.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array2Data`

PURPOSE

Get the int32 data from an array or column cell containing 2-dimensional array data.

INTERFACE

function int32ArrayArray2Data(array)
function int32ColumnArray2DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- integer(kind=INT32), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the int32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
```

```
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
    c1 => int32Array2Data(col1,k)
    c2 => int32Array2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = n
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray2data
! This example shows how to use the int32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray2data
```

```
  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT32), dimension(:,:), pointer :: a1, a2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER32, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int32Array2Data(arr1)
  a2 => int32Array2Data(arr2)
  do j=0,3
    do i=0,2
      a1(i,j) = n
      a2(i,j) = a1(i,j)
      n = n + 1
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array2Data`

PURPOSE

Get the int32 data from a column containing 2-dimensional array data.

INTERFACE

function int32ColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

RETURNS

- integer(kind=INT32), dimension(:,:,:), pointer
  The 2-dimensional data is returned as a 3-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int32Array2Data(col1)
  c2 => int32Array2Data(col2)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = n
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
```

```
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array3Data`

PURPOSE

Get the int32 data from an array or column cell containing 3-dimensional array data.

INTERFACE

function int32ArrayArray3Data(array)
function int32ColumnArray3DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- integer(kind=INT32), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
  use errorhandling
```

```
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => int32Array3Data(col1,l)
    c2 => int32Array3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = n
          c2(i,j,k) = c1(i,j,k)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray3data

  use dal
  use errorhandling

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER32, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr2)
  do k=0,4
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array3Data`

PURPOSE

Get the int32 data from a column containing 3-dimensional array data.

INTERFACE

function int32ColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- integer(kind=INT32), dimension(:,:,:,:), pointer
  The 3-dimensional data is returned as a 4-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int32Array3Data(col1)
  c2 => int32Array3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = n
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
```

```
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array4Data`

PURPOSE

Get the int16 data from a column cell containing 4-dimensional array data.

INTERFACE

function int32ColumnArray4Data(column)
function int32ColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT32), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray4data

  use dal
  use errorhandling

  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int32Array4Data(col1,m)
    c2 => int32Array4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = n
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Array4Data`

PURPOSE

Get the int32 data from a column containing 4-dimensional array data.

INTERFACE

function int32ColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- integer(kind=INT32), dimension(:,:,:,:,:), pointer
  The 4-dimensional column data is returned as a 5-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int32Array4Data(col1)
  c2 => int32Array4Data(col1)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
```

```
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32Attribute`

PURPOSE

Get the value of an attribute as an int32.

INTERFACE

function int32ArrayAttribute(array, name)
function int32AttributableAttribute(attributable, name)
function int32Attribute(attribute)
function int32BlockAttribute(block, name)
function int32ColumnAttribute(column, name)
function int32DataSetAttribute(dataSet, name)
function int32TableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required attribute.
- type(AttributableT), intent(in) :: attributable
  A handle of the attributable containing the required attribute.
- type(AttributeT), intent(in) :: attribute
  A handle of the attribute.
- type(BlockT), intent(in) :: block
  A handle of the block containing the required attribute.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required attribute.
- type(DataSetT), intent(in) :: dataSet
  A handle of the dataset containing the required attribute.
- character(len=*), intent(in) :: name
  The name of the required attribute.
- type(TableT), intent(in) :: table
  A handle of the table containing the required attribute.

RETURNS

- integer(kind=INT32)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how int32 attributes are used.
! The program creates a dataset containing two int32 attributes,
! together with a table containing two int32 attributes.
! The attributes are then accessed, by name, with
! the int32Attribute() function.
! Also, it is shown how to access the attributes by position.
program example_int32attribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"int1",1,"int comment")
  call setAttribute(set,"int2",2,"int comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"int1",3,"int comment")
  call setAttribute(tab,"int2",4,"int comment")

  write(*,*) int32Attribute( set, "int1" ) ! output '1'
  write(*,*) int32Attribute( set, "int2" ) ! output '2'
  write(*,*) int32Attribute( tab, "int1" ) ! output '3'
  write(*,*) int32Attribute( tab, "int2" ) ! output '4'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) int32Attribute( att ) ! output the sequence 1, 2
  end do

  call release(set)

end program example_int32attribute
```

SEE ALSO

BUGS AND LIMITATIONS

NAME

int32Data

PURPOSE

Get the int32 data from an array, column or column cell.

INTERFACE

function int32ArrayData(array)
function int32ColumnData(column)
function int32ColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer, intent(in) :: row
  The row number of the column cell containing the required data.

RETURNS

- integer(kind=INT32), dimension(:), pointer
  The data is returned as a flat vector regardless of the dimensionality of the data.

DESCRIPTION

The data is returned in a vector regardles of the dimensionality of the data. In particular, when accessing a scalar column cell, a vector of length 1 is returned, which contains the single scalar value.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_int32data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:,:,:,:), pointer :: c1, c2
  integer(kind=INT32), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
```

```
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int32Array4Data(col1)
  c2 => int32Array4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
  cd => int32Data( col2 ) ! Access the column's 4-dimensional data as a flat vector.

  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_int32data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32VectorData`

PURPOSE

Get the int32 data from an array or column cell containing vector data.

INTERFACE

function int32ArrayVectorData(array)
function int32ColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- integer(kind=INT32), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => int32VectorData(col1,m)
    c2 => int32VectorData(col2,m)
    do i=0,2
```

```
      c1(i) = n
      c2(i) = c1(i)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellvectordata
! This example shows how to use the int32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The array is then initialised,
program example_arrayvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  integer(kind=INT32), dimension(:), pointer :: a1, a2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", INTEGER32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", INTEGER32, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => int32VectorData(arr1)
  a2 => int32VectorData(arr2)
  do i=0,2
    a1(i) = n
    a2(i) = a1(i)
    n = n + 1
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`int32VectorData`

PURPOSE

Get the int32 data from a column containing vector data.

INTERFACE

function int32ColumnVectorData(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

RETURNS

integer(kind=INT32), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second column has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=INT32), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
```

```
  tab = addTable(set, "table", 10, "table comment" )
  col1 = addColumn( tab, "column1", INTEGER32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => int32VectorData(col1)
  c2 => int32VectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
      c1(i,m) = n
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

INTEGER8

PURPOSE

An enumeration value which is used to indicate int8 data.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

INTEGER16

PURPOSE

An enumeration value which is used to indicate int16 data.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

INTEGER32

PURPOSE

An enumeration value which is used to indicate int16 data.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`intNullValue`

PURPOSE

Get the value of the integer null value.

INTERFACE

subroutine intNullValueArray(array)
subroutine intNullValueColumn(column)
subroutine intNullValueDataComponent(dataComponent)
subroutine intNullValueNullable(nullable)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array whose null value is to be retrieved.
- type(ColumnT), intent(in) :: column
  A handle of the column whose null value is to be retrived.
- type(DataComponentT), intent(in) :: dataComponent
  A handle of the dataComponent whose null value is to be retrieved.
- type(NullableT), intent(in) :: nullable
  A handle of the nullable whose null value is to be retrieved.

RETURNS

DESCRIPTION

Get the null value of an object containing integer data. It is an error to call this function if the object's null value has not been defined. The logical function nullDefined() may be used to determine if the null value of a geven object has been defined.

The null value of an object containing integer data, may be defined with a call to setNull-Value().

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
```

```
      ! Would have given an error, if the null
      ! value of array arr1 had not been set.

  if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
    write(*,*) "Using null value of arr1, in arr2"
    call setNullValue( arr2, intNullValue( arr1 ))
  else
    call setNullValue( arr2, 999999 )
  end if

  call check( nullable( arr2 ) )

  call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

  call release(arr1)
  call release(arr2)

  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif
```

```
  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls isNotNull isNull nullable nullDefined nullType setNullValue setToNull`

BUGS AND LIMITATIONS

None known.

NAME

`isNotNull`

PURPOSE

NOT IMPLEMENTED Determine the state of all the values in an array or column.

INTERFACE

function isNotNull(array)
function isNotNull(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the values to be checked.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the values to be checked.

RETURNS

- logical, dimension(:), pointer :: isNotNullArray A vector whose elements indicate the state of the corresponing values in the given object. If the vector element value is true, the corresponding value in the object is null.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
  use errorhandling
```

```
  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
       ! Would have given an error, if the null
       ! value of array arr1 had not been set.

  if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
    write(*,*) "Using null value of arr1, in arr2"
    call setNullValue( arr2, intNullValue( arr1 ))
  else
    call setNullValue( arr2, 999999 )
  end if

  call check( nullable( arr2 ) )

  call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

  call release(arr1)
  call release(arr2)
```

```
  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNull` `nullable` `nullDefined` `nullType` `setToNull` `setNullValue`

BUGS AND LIMITATIONS

None known.

NAME

`isNull`

PURPOSE

Determines if a value is null.

INTERFACE

function isNullArray(array, position)

function isNullCell(column, row, position)
function isNullColumn(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the value to be checked.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the value to be checked.
- integer(kind=INT32), intent(in) :: position
  The position of the value within the array (or the column cell in the case of a multi-dimensional column) which is to be checked.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the value to be checked.

RETURNS

- logical
  True, if the value is null, false otherwise.

DESCRIPTION

In the case of integer values, an error will be raised if the object (array or column) does not have a null-value defined.

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
```

```
set = dataSet("test.dat",CREATE)
arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

! fill with unique numbers
a1 => int32Array3Data(arr1)
a2 => int32Array3Data(arr1)

n = 0
do k=0,1
  do j=0,3
    do i=0,2
      a1(i,j,k) = n
      a2(i,j,k) = a1(i,j,k) + 1
      n = n + 1
    end do
  end do
end do

call setNullValue( arr1, 999999 )
call check( nullable( arr1 ) )

call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
     ! Would have given an error, if the null
     ! value of array arr1 had not been set.

if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
  write(*,*) "Using null value of arr1, in arr2"
  call setNullValue( arr2, intNullValue( arr1 ))
else
  call setNullValue( arr2, 999999 )
end if

call check( nullable( arr2 ) )

call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

call release(arr1)
call release(arr2)

tab = addTable(set,"some table",100)

col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

i32 => int32Data(col1)
do i=0,numberOfRows(tab)-1
  i32(i) = 3*i
end do
call setNullValue( col1, 999999 )
call check( nullable( col1 ) )

call setToNull( col1, 0 ) ! Set the first element of column col1 to null.
```

```
  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNotNull` `nullable` `nullDefined` `nullType` `setNullValue` `setToNull`

BUGS AND LIMITATIONS

None known.

NAME

`keepDataSet`

PURPOSE

Tells the data set server object to not to discard the named data set.

ARGUMENTS

- character(len=*), intent(in) :: dataSetName
  The name of the dataset.

RETURNS

None

DESCRIPTION

The named data set will not be released from memory.

This subroutine must only be called by Meta Tasks.

ERRORS

EXAMPLES

```
! This example shows how to use the keepDataSet
! subroutine
program example_keepdiscarddataset

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet("test.dat",CREATE)
  call release(set)   ! The dataset will be released from memory

  call keepDataSet("test.dat")   ! Tell the dataset server not to discard
     ! the dataset with name "test.dat"

  set = dataSet("test.dat",READ)
  call release(set)   ! The dataset will not be released from memory

  set = dataSet("test.dat",READ)  ! The dataset is already in memory, so this
     ! operation has virtually no overhead.

  call release(set)   ! The dataset will not be released from memory
  call discardDataSet("test.dat") ! Tell the dataset server to discard and
     ! release the dataset with name "test.dat"

end program example_keepdiscarddataset
```

SEE ALSO

`discardDataSet`

BUGS AND LIMITATIONS

None known.

NAME

`label`

PURPOSE

Get the label (comment) of an object.

INTERFACE

function arrayAttributeComment(array, name)
function attributableAttributeComment(attributable, name)
function blockAttributeComment(block, name)
function columnAttributeComment(column, name)
function dataSetAttributeComment(dataSet, name)
function labelOfAttributable(attributable)
function labelOfAttribute(attribute)
function labelOfArray(array)
function labelOfBlock(block)
function labelOfColumn(column)
function labelOfDataSet(dataSet)
function labelOfLabelled(labelled)
function labelOfTable(table)
function tableAttributeComment(table, name)

ARGUMENTS

- type(ArrayT) :: array
  A handle of an array.
- type(AttributableT) :: attributable
  A handle of an attributable.
- type(AttributeT), intent(in) :: attribute
  A handle of an attribute.
- type(BlockT) :: block
  A handle of a block.
- type(ColumnT) :: column
  A handle of a column.
- type(DataSetT) :: dataSet
  A handle of a dataset.
- type(LabelledT) :: labelled
  A handle of a labelled.
- character(len=*), intent(in) :: name
  The name of the attribute from which the comment is to be retrieved.
- type(TableT) :: table
  A handle of a table.

RETURNS

- character(len=IdentifierLength)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none

  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att
```

```
  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )

  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display

program example_labelled

  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
```

```
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`labelled`

PURPOSE

INTERFACE

function arrayLabelled(array)
function attributableLabelled(attributable)
function attributeLabelled(attribute)
function blockLabelled(block)
function columnLabelled(column)
function datasetLabelled(dataSet)
function tableLabelled(table)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of an array.
- type(AttributableT), intent(in) :: attributable
  A handle of an attributable.
- type(AttributeT), intent(in) :: attribute
  A handle of an attribute.
- type(BlockT), intent(in) :: block
  A handle of a block.
- type(ColumnT), intent(in) :: column
  A handle of a column.
- type(DataSetT), intent(in) :: dataSet
  A handle of a dataSet.
- type(TableT), intent(in) :: table
  A handle of a table.

RETURNS

- type(LabelledT)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none

  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att

  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )

  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display
```

```
program example_labelled

  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

LabelledT

PURPOSE

A derived type which is used to declare Labelled handle objects.

DESCRIPTION

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none
```

```
  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att

  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )

  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display

program example_labelled

  use dal

  implicit none

  type(DataSetT) set
```

```
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

`label` `name`

BUGS AND LIMITATIONS

None known.

NAME

LOW_MEMORY

PURPOSE

An enumeration value which is used to indicate that the Low Memory Modelshould be used to open a dataset.

DESCRIPTION

ERRORS

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`mode`(dataSet)

PURPOSE

Get the access mode of an object.

ARGUMENTS

- type(DataSetT) :: dataSet

RETURNS

- integer
  The value returned is one of the enumeratioin values: READ, CREATE, MODIFY, TEMP

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the mode()
! function is used.
function addTableToSet( s, n, r )

  use dal

  implicit none

  type(DataSetT), intent(in) :: s
  character(len=*), intent(in) :: n
  integer, intent(in) :: r
  type(TableT) :: addTableToSet

  if( mode( s ).eq.READ ) then
    write(*,*) 'The table with name ', n, ' is read only'
  else
    addTableToSet = addTable(s,n,r)
  end if
end function addTableToSet

program example_mode

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(BlockT) blk
  integer i
  type(TableT) :: addTableToSet

  set = dataSet("test.dat",CREATE)
  tab = addTableToSet(set,"table1",10)
  call release( set )

  set = dataSet("test.dat",READ)
  tab = addTableToSet(set,"table2",10)
  call release( set )
```

```
end program example_mode
```

SEE ALSO

READ CREATE MODIFY TEMP

BUGS AND LIMITATIONS

None known.

NAME

MODIFY

PURPOSE

An enumeration value which is used to indicate that modify (Read and Write) mode should be used.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`name`

PURPOSE

Get the name of an object.

INTERFACE

function nameOfArray(array)
function nameOfAttributable(attributable)
function nameOfAttribute(attribute)
function nameOfBlock(block)
function nameOfColumn(column)
function nameOfDataSet(dataSet)
function nameOfLabelled(labelled)
function nameOfTable(table)

ARGUMENTS

- type(ArrayT) :: array
  A handle of the array whose name is required.
- type(AttributableT) :: attributable
  A handle of the attributable whose name is required.
- type(AttributeT) :: attribute
  A handle of the attribute whose name is required.
- type(BlockT) :: block
  A handle of the block whose name is required.
- type(ColumnT) :: column
  A handle of the column whose name is required.

- type(DataSetT) :: dataSet
  A handle of the dataset whose name is required.
- type(LabelledT) :: labelled
  A handle of the labelled whose name is required.
- type(TableT) :: table
  A handle of the table whose name is required.

RETURNS

- character(len=IdentifierLength) :: nameOfAttribute

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none

  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att

  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
```

```
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )

  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display

program example_labelled

  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

`label` LabelledT

BUGS AND LIMITATIONS

None known.

NAME

`next`

PURPOSE

Iterate to the next subtable.

INTERFACE

function subTableNext(subTable)

ARGUMENTS

- type(SubTableT), intent(in) :: subTable
  A handle of the subTable.

RETURNS

- logical

DESCRIPTION

ERRORS

EXAMPLES

TBD

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`nullable`

PURPOSE

Convert a subclass of Nullable to Nullable.

INTERFACE

function arrayNullable(array)
function columnNullable(column)
function dataComponentNullable(dataComponent)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  The handle of an array which is to be converted to a DataComponent
- type(ColumnT), intent(in) :: column
  The handle of a column which is to be converted to a DataComponent
- type(DataComponentT), intent(in) :: dataComponent
  The handle of a dataComponent object which is to be converted to a Nullable object.

RETURNS

- type(DataComponentT)
  The converted object is returned as a handle to a DataComponent object.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example illustrates the use of the dataComponent() function.
! The units of objects with data type BOOLEAN and STRING are meaningless
! and so are not displayed.
subroutine displayUnits( dcomponent )
  use dal

  implicit none

  type(DataComponentT) dcomponent
  integer dattype

  dattype = dataType( dcomponent )
  write(*,*) dattype
  if(dattype.eq.INTEGER8.or.dattype.eq.INTEGER16.or.dattype.eq.INTEGER32 &
    .or.dattype.eq.REAL32.or.dattype.eq.REAL64) then
    write(*,*) units( dcomponent )
  end if

end subroutine displayUnits

program example_datacomponent

  use dal

  implicit none

  type(ArrayT) arr
  type(BlockT) blk
  type(ColumnT) col
  type(DataSetT) set
  type(TableT) tab
  integer i, j
  integer, dimension(3), parameter :: s = (/ 2,3,4 /)

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))
  arr = addArray(set, "array1", INTEGER16, dimensions=s, units="klm" )
  arr = addArray(set, "array2", INTEGER32, dimensions=s, units="kla" )

  do i = 0, numberOfBlocks( set ) - 1
    blk = block( set, i, READ )
    if( blockType( blk ).eq.ARRAY_BLOCK ) then
```

```
      arr = array( set, name( blk ), READ )
      call displayUnits( dataComponent( arr ) )
    else
      tab = table( set, name( blk ) )
      do j = 0, numberOfColumns( tab ) - 1
        col = column( tab, j, READ )
        call displayUnits( dataComponent( col ) )
      end do
    end if
  end do
  call release(set)

end program example_datacomponent
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`nullDefined`

PURPOSE

Determine if the integer null value has been set.

INTERFACE

function nullDefinedArray(array)
function nullDefinedColumn(column)
function nullDefinedDataComponent(dataComponent)
function nullDefinedNullable(nullable)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array.
- type(ColumnT), intent(in) :: column
  A handle of the column.
- type(DataComponentT), intent(in) :: dataComponent
  A handle of the dataComponent.
- type(NullableT), intent(in) :: nullable
  A handle of the nullable.

RETURNS

- logical

DESCRIPTION

This function is only relevant for objects containing boolean data.

For real objects, this function always returns true.

The null value of an object containing integer data may be defined by calling setNullValue().

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
       ! Would have given an error, if the null
       ! value of array arr1 had not been set.
```

```
    if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
      write(*,*) "Using null value of arr1, in arr2"
      call setNullValue( arr2, intNullValue( arr1 ))
    else
      call setNullValue( arr2, 999999 )
    end if

    call check( nullable( arr2 ) )

    call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
  ! Would have given an error, if the null
  ! value of array arr2 had not been set.

    call release(arr1)
    call release(arr2)

    tab = addTable(set,"some table",100)

    col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

    i32 => int32Data(col1)
    do i=0,numberOfRows(tab)-1
      i32(i) = 3*i
    end do
    call setNullValue( col1, 999999 )
    call check( nullable( col1 ) )

    call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


    col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
    r64 => real64Data(col2)
    do i=0,numberOfRows(tab)-1
      r64(i) = 0.25*i
    end do

    ! col is a non-integer column and it would be an
    ! an error to call setNullValue().
    call check( nullable( col2 ) )

    call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

    if( hasNulls( col2 ) ) then
      do i=0,numberOfRows(tab)-1
        if( isNull( col2, i ) ) then
          write(*,*) "element", i, "is null"
        else
          write(*,*) "element", i, "is", r64(i)
        endif
      end do
    endif

    call release(col1)
    call release(col2)
    call release(set)
```

```
end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNotNull` `isNull` `nullable` `nullType` `setNullValue` `setToNull`

BUGS AND LIMITATIONS

None known.

NAME

`nullType`

PURPOSE

Gets the null value type of an object.

INTERFACE

function nullTypeArray(array)
function nullTypeColumn(column)
function nullType(dataComponent)
function nullTypeNullable(nullable)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array.
- type(ColumnT), intent(in) :: column
  A handle of the column.
- type(DataComponentT), intent(in) :: dataComponent
  A handle of the dataComponent.
- type(NullableT), intent(in) :: nullable
  A handle of the nullable.

RETURNS

- integer
  Returns one of: INTEGER_NULL, REAL_NULL, STRING_NULL, UNDEFINED_NULL

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
```

```
use errorhandling

implicit none

type(DataSetT) set
type(ArrayT) arr1, arr2
type(TableT) tab
type(ColumnT) col1, col2
integer(kind=int32), dimension(:), pointer :: i32
real(kind=double), dimension(:), pointer :: r64
integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
integer, dimension(3), parameter :: s = (/ 3,4,2 /)
integer :: i,j,k,n

! create a set
set = dataSet("test.dat",CREATE)
arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

! fill with unique numbers
a1 => int32Array3Data(arr1)
a2 => int32Array3Data(arr1)

n = 0
do k=0,1
  do j=0,3
    do i=0,2
      a1(i,j,k) = n
      a2(i,j,k) = a1(i,j,k) + 1
      n = n + 1
    end do
  end do
end do

call setNullValue( arr1, 999999 )
call check( nullable( arr1 ) )

call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
     ! Would have given an error, if the null
     ! value of array arr1 had not been set.

if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
  write(*,*) "Using null value of arr1, in arr2"
  call setNullValue( arr2, intNullValue( arr1 ))
else
  call setNullValue( arr2, 999999 )
end if

call check( nullable( arr2 ) )

call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

call release(arr1)
```

```
  call release(arr2)

  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNotNull` `isNull` `nullable` `nullDefined` `setNullValue` `setToNull`

BUGS AND LIMITATIONS

None known.

NAME

`numberOfAttributes`

PURPOSE

Get the number of attributes in an object.

INTERFACE

function numberOfAttributesOfArray(array)
function numberOfAttributesOfAttrib(attributable)
function numberOfAttributesOfBlock(block)
function numberOfAttributesOfColumn(column)
function numberOfAttributesOfDataSet(dataSet)
function numberOfAttributesOfTable(table)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array for which the number of attributes is required.
- type(AttributableT), intent(in) :: attributable
  A handle of the attributable for which the number of attributes is required.
- type(BlockT), intent(in) :: block
  A handle of the block for which the number of attributes is required.
- type(ColumnT), intent(in) :: column
  A handle of the column for which the number of attributes is required.
- type(DataSetT), intent(in) :: dataSet
  A handle of the dataset for which the number of attributes is required.
- type(TableT), intent(in) :: table
  A handle of the table for which the number of attributes is required.

RETURNS

- integer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example show how the numberOfAttributes interface
! is used.
program example_numberofattributes

  use dal
  implicit none

  type(DataSetT) set
  type(TableT) tab

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.false.,"dataset bool comment")

  write(*,*) numberOfAttributes( set ) ! 2 attributes
  tab = addTable(set,"table",10);
  call addAttributes(attributable(tab),attributable(set))
  call setAttribute(tab,"sbool3",.false.,"dataset bool comment")
  write(*,*) numberOfAttributes( tab ) ! 3 attributes
```

```
  call release(set)

end program example_numberofattributes
```

SEE ALSO

setAttribute

BUGS AND LIMITATIONS

None known.

NAME

numberOfBlocks(dataSet)

PURPOSE

Get the number of blocks in a dataset.

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet

RETURNS

- integer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the numberOfBlocks interface
! is used.
program example_numberofblocks

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(BlockT) blk
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table1",10)
  tab = addTable(set,"table2",100)
  tab = addTable(set,"table3",1000)

  write(*,*) numberOfBlocks( set ) ! 3 blocks

  ! For each block, display the name, and
  ! add a comment.
  do i=0,numberOfBlocks( set ) - 1
    blk = block( set, i, MODIFY )
    write(*,*) name( blk )
```

```
    call addComment( blk, "A table comment" )
  end do

  call release(set)

end program example_numberofblocks
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfColumns(table)

PURPOSE

Get the number of columns in a table.

ARGUMENTS

- type(TableT), intent(in) :: table

RETURNS

- integer

DESCRIPTION

ERRORS

EXAMPLES

```
! This examples shows how the numberOfColumns()
! function is used.
program example_numberofcolumns

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
```

```
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))

  write(*,*) numberOfColumns( tab ) ! 7 columns

  ! For each column, display the name and
  ! add an attribute.
  do i=0, numberOfColumns( tab ) - 1
    col = column( tab, i, MODIFY )
    write(*,*) name( col )
    call setAttribute( col, "TLMAX", 10, "tlmax attribute" )
  end do

  call release(set)

end program example_numberofcolumns
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`numberOfDimensions`

PURPOSE

Get the number of dimensions of the data contained in an object.

INTERFACE

function numberOfDimensionsOfArray(array)
function numberOfDimensionsOfColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array for which the number of dimensions is required.
- type(ColumnT), intent(in) :: column
  A handle of the column for which the number of dimensions is required.

RETURNS

- integer
  The number of the dimensions of the given object.

DESCRIPTION

For arrays the number of dimensions is between 1 and 3, and for columns is between 1 and 4.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! two 3-dimensional arrays, and one table.
!
! It illustrates the use of the numberofdimensions interface.
```

```
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
program example_numberofdimensions

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  integer :: i,j

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr = addArray(set, "array1", INTEGER32, dimensions=(/3/) )
  arr = addArray(set, "array2", dataType( arr ), dimensions=(/3,4/) )
  arr = addArray(set, "array3", dataType( arr ), dimensions=(/3,4,5/) )
  tab = addTable(set,"table",10)
  col = addColumn(tab,"col1",INTEGER8) ! scalar
  col = addColumn(tab,"col2",dataType(col),dimensions=(/3/)) ! vector
  col = addColumn(tab,"col3",dataType(col),dimensions=(/3,4/)) ! 2-dimensions
  col = addColumn(tab,"col4",dataType(col),dimensions=(/3,4,5/)) ! 3-dimensions
  col = addColumn(tab,"col5",dataType(col),dimensions=(/3,4,5,6/)) ! 4-dimensions

  do i = 0, numberOfBlocks( set ) - 1
    ! For each block which is an array, display the
    ! name and number of dimensions.
    if( blockType( set, i ).eq.ARRAY_BLOCK ) then
      arr = array( set, i, READ )
      write(*,*) name( arr ), numberOfDimensions( arr )
    else
      tab = table( set, i )
      do j = 0, numberOfColumns( tab ) - 1
        ! For each column, display the name
! and the number of dimensions.
        col = column( tab, j, READ )
write(*,*) name( col ), numberOfDimensions( col )
      end do
    end if
  end do

  call release(set)

end program example_numberofdimensions
```

SEE ALSO

`addArray` `addColumn`

BUGS AND LIMITATIONS

None known.

NAME

numberOfElements

PURPOSE

Get the number of data elements in an object.

INTERFACE

function numberOfElementsOfColumn(column)
A handle of the column for which the number of elements is required. function numberOfElementsOfArray(array) A handle of the array for which the number of elements is required.

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array for which the number of elements is required.
- type(ColumnT), intent(in) :: column

RETURNS

- integer

DESCRIPTION

For fixed-length columns the number of elements in each cell is returned. The total number of elements in a column is therefore calculated by multiplying the number of the rows in the column by the result of this function. For variable-lenghth columns, zero is returned.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! two 3-dimensional arrays, and one table.
!
! It illustrates the use of the numberofelements interface.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
program example_numberofdimensions

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  integer :: i,j

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr = addArray(set, "array1", INTEGER32, dimensions=(/3/) )
```

```
  arr = addArray(set, "array2", dataType( arr ), dimensions=(/3,4/) )
  arr = addArray(set, "array3", dataType( arr ), dimensions=(/3,4,5/) )
  tab = addTable(set,"table",10)
  col = addColumn(tab,"col1",INTEGER8) ! scalar
  col = addColumn(tab,"col2",dataType(col),dimensions=(/3/)) ! vector
  col = addColumn(tab,"col3",dataType(col),dimensions=(/3,4/)) ! 2-dimensions
  col = addColumn(tab,"col4",dataType(col),dimensions=(/3,4,5/)) ! 3-dimensions
  col = addColumn(tab,"col5",dataType(col),dimensions=(/3,4,5,6/)) ! 4-dimensions

  do i = 0, numberOfBlocks( set ) - 1
    ! For each block which is an array, display the
    ! name, number of dimensions and the number of elements.
    if( blockType( set, i ).eq.ARRAY_BLOCK ) then
      arr = array( set, i, READ )
      write(*,*) name( arr ), numberOfDimensions( arr ), numberOfElements( arr )
    else
      tab = table( set, i )
      do j = 0, numberOfColumns( tab ) - 1
        ! For each column, display the name,
! number of dimensions and total number of elements.
        col = column( tab, j, READ )
write(*,*) name( col ), numberOfDimensions( col ), numberOfRows( col ) * numberOfElements( co
      end do
    end if
  end do

  call release(set)

end program example_numberofdimensions
```

SEE ALSO

`addArray` `addColumn`

BUGS AND LIMITATIONS

None known.

NAME

`numberOfRows`(table)

PURPOSE

Get the number of rows in a table.

INTERFACE

function numberOfRowsInColumn(column)
function numberOfRowsInTable(table)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column for which the number of rows is required.
- type(TableT), intent(in) :: table
  A handle of the table for which the number of rows is required.

RETURNS

- integer
  The number of rows.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the numberOfRows
! interface is used.
program example_numberofrows

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table1",10)
  tab = addTable(set,"table2",100)
  tab = addTable(set,"table3",1000)

  do i=0,numberOfBlocks( set ) - 1
    tab = table( set, i )
    write(*,*) name( tab ), numberOfRows( tab )
  end do

  call release(set)

end program example_numberofrows
```

SEE ALSO

addTable

BUGS AND LIMITATIONS

None known.

NAME

parent

PURPOSE

Get the parent object of an object.

INTERFACE

function parentAttributable(attribute)

ARGUMENTS

- type(AttributeT), intent(in) :: attribute

RETURNS

- type(AttributableT)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the parent interface
! is used.
subroutine test1( set, tab, arr, col )

  use dal

  type(DataSetT), intent(in) :: set
  type(TableT), intent(in) :: tab
  type(ArrayT), intent(in) :: arr
  type(ColumnT), intent(in) :: col
  type(AttributeT) att



  att = attribute( set,0 )
  write(*,*) name( parent( att ))
  if( name( parent( att )) /= name( set )) then
call error('internalError',"problem in parent method" )
  end if

  att = attribute( tab,0 )
  write(*,*) name( parent( att ))
  if( name( parent( att )) /= name( tab )) then
call error('internalError',"problem in parent method" )
  end if

  write(*,*) name( parent( tab ))
  if( name( parent( tab )) /= name( set )) then
call error('internalError',"problem in parent method" )
  end if

  att = attribute( arr,0 )
  write(*,*) name( parent( att ))
  if( name( parent( att )) /= name( arr )) then
call error('internalError',"problem in parent method" )
  end if

  write(*,*) name( parent( arr ))

  if( name( parent( arr )) /= name( set )) then
call error('internalError',"problem in parent method" )
  end if

  att = attribute( col,0 )
  write(*,*) name( parent( att ))
```

```
  if( name( parent( att )) /= name( col )) then
call error('internalError',"problem in parent method" )
  end if

  write(*,*) name( parent( col ))

  if( name( parent( col )) /= name( tab )) then
call error('internalError',"problem in parent method" )
  end if

  write(*,*) name( parent( parent( col )))

  if( name( parent( parent( col ))) /= name( set )) then
call error('internalError',"problem in parent method" )
  end if

end subroutine test1


program example_parent

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  type(ArrayT) arr
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sint8",1_int8,"int8 unit","set int8 comment")
  tab = addTable(set,"some table",100)
  call setAttribute(tab,"sint8",1_int8,"int8 unit","set int8 comment")
  arr = addArray(set, "some array", INTEGER32, dimensions=s )
  call setAttribute(arr,"sint8",1_int8,"int8 unit","set int8 comment")
  col = addColumn(tab,"bool",BOOLEAN)
  call setAttribute(col,"TLMIN",1_int8,"int8 unit","set int8 comment")

  call test1( set,tab,arr,col )

  call release(set)

  set = dataSet("test.dat",READ)
  tab = table(set,0)
  arr = array(set,1,READ)
  col = column(tab,0,READ)

  call test1( set,tab,arr,col )

  call release(set)

end program example_parent
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`parent`

PURPOSE

Get the parent object of an object.

INTERFACE

function parentDataSetOfArray(array)

ARGUMENTS

- type(ArrayT), intent(in) :: array

RETURNS

- type(DataSetT)

DESCRIPTION

ERRORS

EXAMPLES

See above.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`parent`

PURPOSE

Get the parent object of an object.

INTERFACE

function parentDataSetOfBlock(block)

ARGUMENTS

- type(BlockT), intent(in) :: block

RETURNS

- type(DataSetT)

DESCRIPTION

ERRORS

EXAMPLES

See above.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`parent`

PURPOSE

Get the parent object of an object.

INTERFACE

function parentDataSetOfTable(table)

ARGUMENTS

- type(TableT), intent(in) :: table

RETURNS

- type(DataSetT)

DESCRIPTION

ERRORS

EXAMPLES

See above.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`parent`

PURPOSE

Get the parent object of an object.

INTERFACE

function parentTable(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- type(TableT)

DESCRIPTION

ERRORS

EXAMPLES

See above.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`qualifiedName`

PURPOSE

Get the fully qualified name of an object.

INTERFACE

function qualifiedNameOfArray(array)
function qualifiedNameOfAttributable(attributable)
function qualifiedNameOfAttribute(attribute)
function qualifiedNameOfBlock(block)
function qualifiedNameOfColumn(column)
function qualifiedNameOfDataSet(dataSet)
function qualifiedNameOfLabelled(labelled)
function qualifiedNameOfTable(table)

ARGUMENTS

- type(ArrayT) :: array
  A handle of the array whose fully qualified name is required.
- type(AttributableT) :: attributable
  A handle of the attributable whose fully qualified name is required.
- type(AttributeT) :: attribute
  A handle of the attribute whose fully qualified name is required.
- type(BlockT) :: block
  A handle of the block whose fully qualified name is required.
- type(ColumnT) :: column
  A handle of the column whose fully qualified name is required.
- type(DataSetT) :: dataSet
  A handle of the dataset whose fully qualified name is required.
- type(LabelledT) :: labelled
  A handle of the labelled whose fully qualified name is required.
- type(TableT) :: table
  A handle of the table whose fully qualified name is required.

RETURNS

- character(len=IdentifierLength)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the qualifiedName
! interface is used.
program example_qualifiedname

  use dal

  type(DataSetT) :: set
  type(ArrayT) :: arr
  type(TableT) :: tab
  type(ColumnT) :: col
  type(AttributeT) :: att

  set = dataSet("test.dat",create)
  call setAttribute(set,"sbool",.false.,"set bool comment")

  arr = addArray(set,"array",integer32, dimensions=(/ 1,2,3 /))
  call setAttribute(arr,"abool",.true.,"arr bool comment")

  tab = addTable(set,"table",10)
  call setAttribute(tab,"tbool",.false.,"tab bool comment")

  col = addColumn(tab,"column",INT32,units="UNITS",comment="Column")
  call setAttribute(col,"tlmin",1_int32,"int32 unit","col int32 comment")

  write(*,*) "qualified data set name: ", qualifiedName( set ) ! test.dat
  att = attribute( set, "sbool" )
  write(*,*) "qualified data set attribute name: ", qualifiedName( att ) !"test.dat:sbool
  write(*,*) "qualified table name: ", qualifiedName( tab )! test.dat:table
  att = attribute( tab, "tbool" )
  write(*,*) "qualified table attribute name: ", qualifiedName( att ) ! test.dat:table:tbool
  write(*,*) "qualified array name: ", qualifiedName( arr ) ! test.dat:array
  att = attribute( arr, "abool" )
  write(*,*) "qualified array attribute name: ", qualifiedName( att ) ! test.dat:array:abool
  write(*,*) "qualified column name: ", qualifiedName( col ) ! test.dat:table:column
  att = attribute( col, "tlmin" )
  write(*,*) "qualified array attribute name: ", qualifiedName( att ) ! test.dat:table:column

  call release(set)

end program example_qualifiedname
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

READ

PURPOSE

An enumeration value which is used to indicate read access to an object.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

REAL32

PURPOSE

An enumeration value which is used to indicate an object contains data of type real32 (float).

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Array2Data`

PURPOSE

Get the real32 data from an array or a column cell containing 2-dimensional array data.

INTERFACE

function real32ArrayArray2Data(array)
function real32ColumnArray2DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=SINGLE), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the real32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
    c1 => real32Array2Data(col1,k)
    c2 => real32Array2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = n
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
```

```
  call release(col2)
  call release(set)

end program example_cellarray2data
! This example shows how to use the real32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=SINGLE), dimension(:,:), pointer :: a1, a2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL32, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => real32Array2Data(arr1)
  a2 => real32Array2Data(arr2)
  do j=0,3
    do i=0,2
      a1(i,j) = n
      a2(i,j) = a1(i,j)
      n = n + 1
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Array2Data`

PURPOSE

Get the real32 data from a column containing 2-dimensional array data.

INTERFACE

function real32ColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=SINGLE), dimension(:,:,:), pointer
  The 2-dimensional column data is returned as a 3-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )
```

```
  ! fill with unique numbers
  c1 => real32Array2Data(col1)
  c2 => real32Array2Data(col2)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = n
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Array3Data`

PURPOSE

Get the real32 data from an an array or a column cell containing 3-dimensional array data.

INTERFACE

function real32ArrayArray3Data(array)
function real32ColumnArray3DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=SINGLE), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => real32Array3Data(col1,l)
    c2 => real32Array3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = n
          c2(i,j,k) = c1(i,j,k)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 3-dimensional arrays.
```

```
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=SINGLE), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL32, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => real32Array3Data(arr1)
  a2 => real32Array3Data(arr2)
  do k=0,4
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

real32Array3Data

PURPOSE

Get the real32 data from an column containing 3-dimensional array data.

INTERFACE

function real32ColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=SINGLE), dimension(:,:,:,:), pointer
  The 3-dimensional column data is returned as a 4-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real32Array3Data(col1)
  c2 => real32Array3Data(col1)

  n = 0
```

```
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = n
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Array4Data`

PURPOSE

Get the real32 data from a column cell containing 4-dimensional array data.

INTERFACE

function real32ColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=SINGLE), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
```

```
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => real32Array4Data(col1,m)
    c2 => real32Array4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = n
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Array4Data`

PURPOSE

Get the real32 data from a column containing 4-dimensional array data.

INTERFACE

function real32ColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=SINGLE), dimension(:,:,:,:,:), pointer
  The 4-dimensional data is returned as a 5-dimensional array.

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )
```

```
  ! fill with unique numbers
  c1 => real32Array4Data(col1)
  c2 => real32Array4Data(col1)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32Attribute`

PURPOSE

Get the value of an attribute as a real32.

INTERFACE

function real32ArrayAttribute(array, name)
function real32AttributableAttribute(attributable, name)
function real32Attribute(attribute)
function real32BlockAttribute(Block, name)
function real32ColumnAttribute(column, name)
function real32DataSetAttribute(dataSet, name)
function real32TableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required attribute.
- type(AttributableT), intent(in) :: attributable
  A handle of the attributable containing the required attribute.
- type(AttributeT), intent(in) :: attribute
  A handle of the attribute.
- type(BlockT), intent(in) :: block
  A handle of the block containing the required attribute.

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required attribute.
- type(DataSetT), intent(in) :: dataSet
  A handle of the dataset containing the required attribute.
- character(len=*), intent(in) :: name
  The name of the required attribute.
- type(TableT), intent(in) :: table
  A handle of the table containing the required attribute.

RETURNS

- real(kind=SINGLE)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how real32 attributes are used.
! The program creates a dataset containing two real32 attributes,
! together with a table containing two real32 attributes.
! The attributes are then accessed, by name, with
! the real32Attribute() function.
! Also, it is shown how to access the attributes by position.
program example_real32attribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"real1",1.0,"real comment")
  call setAttribute(set,"real2",2.0,"real comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"real1",3.0,"real comment")
  call setAttribute(tab,"real2",4.0,"real comment")

  write(*,*) real32Attribute( set, "real1" ) ! output '1.0'
  write(*,*) real32Attribute( set, "real2" ) ! output '2.0'
  write(*,*) real32Attribute( tab, "real1" ) ! output '3.0'
  write(*,*) real32Attribute( tab, "real2" ) ! output '4.0'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) real32Attribute( att ) ! output the sequence 1.0, 2.0
```

```
  end do

  call release(set)

end program example_real32attribute
```

SEE ALSO

BUGS AND LIMITATIONS

NAME

real32Data

PURPOSE

Get the real32 data from an array, column or column cell.

INTERFACE

function real32ArrayData(array)
function real32ColumnData(column)
function real32ColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column A handle of the column containing the required data.
- integer, intent(in) :: row The row number of the column cell containing the required data.

RETURNS

- real(kind=SINGLE), dimension(:), pointer
  The data is returned as a flat vector regardless of the dimensionality of the

DESCRIPTION

The data is returned in a vector regardles of the dimensionality of the data. In particular, when accessing a scalar column cell, a vector of length 1 is returned, which contains the single scalar value.

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_real32data

  use dal
```

```
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:,:,:,:), pointer :: c1, c2
  real(kind=SINGLE), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real32Array4Data(col1)
  c2 => real32Array4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
  cd => real32Data( col2 ) ! Access the column's 4-dimensional data as a flat vector.

  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_real32data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32VectorData`

PURPOSE

Get the real32 data from an array or column cell containing vector data.

INTERFACE

function real32ArrayVectorData(array)
function real32ColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the required data.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the data to be accessed.

RETURNS

- real(kind=SINGLE), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
```

```
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => real32VectorData(col1,m)
    c2 => real32VectorData(col2,m)
    do i=0,2
      c1(i) = n
      c2(i) = c1(i)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellvectordata
! This example shows how to use the int32Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The array is then initialised,
program example_arrayvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=SINGLE), dimension(:), pointer :: a1, a2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL32, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL32, s, "km", "array comment" )

  ! fill with unique numbers
```

```
  n = 0
  a1 => real32VectorData(arr1)
  a2 => real32VectorData(arr2)
  do i=0,2
    a1(i) = n
    a2(i) = a1(i)
    n = n + 1
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real32VectorData`

PURPOSE

Get the real32 data from a column containing vector data.

INTERFACE

function real32ColumnVectorData(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle of the column containing the required data.

RETURNS

real(kind=SINGLE), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second column has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
```

```
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=SINGLE), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 10, "table comment" )
  col1 = addColumn( tab, "column1", REAL32, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real32VectorData(col1)
  c2 => real32VectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
      c1(i,m) = n
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

REAL64

PURPOSE

An enumeration value which is used to indicate an object contains data of type real64 (double).

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Array2Data`

PURPOSE

Get the real64 data from an array or column cell containing 2-dimensional array data.

INTERFACE

function real64ArrayArray2Data(array) result(ptr) function real64ColumnArray2DataElement(column, row) result(ptr)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to use the real64Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
```

```
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do k=0,numberOfRows(tab) - 1
    c1 => real64Array2Data(col1,k)
    c2 => real64Array2Data(col2,k)
    do j=0,3
      do i=0,2
        c1(i,j) = n
        c2(i,j) = c1(i,j)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray2data
! This example shows how to use the real64Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=DOUBLE), dimension(:,:), pointer :: a1, a2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
```

```
  integer :: i,j,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL64, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL64, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => real64Array2Data(arr1)
  a2 => real64Array2Data(arr2)
  do j=0,3
    do i=0,2
      a1(i,j) = n
      a2(i,j) = a1(i,j)
      n = n + 1
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Array2Data`

PURPOSE

Get the real64 data from a column containing 2-dimensional array data.

INTERFACE

function real64ColumnArray2Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=DOUBLE), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 2-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array2data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(2), parameter :: s = (/ 3,4 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real64Array2Data(col1)
  c2 => real64Array2Data(col2)

  n = 0
  do k=0,numberOfRows(tab) - 1
    do j=0,3
      do i=0,2
        c1(i,j,k) = n
        c2(i,j,k) = c1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array2data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

real64Array3Data

PURPOSE

Get the real64 data from an array or a column cell containing 3-dimensional array data.

INTERFACE

function real64ArrayArray3Data(array)
function real64ColumnArray3DataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
```

```
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do l=0,numberOfRows(tab) - 1
    c1 => real64Array3Data(col1,l)
    c2 => real64Array3Data(col2,l)
    do k=0,4
      do j=0,3
        do i=0,2
          c1(i,j,k) = n
          c2(i,j,k) = c1(i,j,k)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray3data
! This example shows how to use the int8Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 3-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_arrayarray3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=DOUBLE), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL64, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL64, s, "km", "array comment" )
```

```
  ! fill with unique numbers

  n = 0
  a1 => real64Array3Data(arr1)
  a2 => real64Array3Data(arr2)
  do k=0,4
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k)
        n = n + 1
      end do
    end do
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayarray3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Array3Data`

PURPOSE

Get the real64 data from a column containing 3-dimensional array data.

INTERFACE

function real64ColumnArray3Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=DOUBLE), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 3-dimensional arrays.
!
```

```
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array3data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(3), parameter :: s = (/ 3,4,5 /)
  integer :: i,j,k,l,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real64Array3Data(col1)
  c2 => real64Array3Data(col1)

  n = 0
  do l=0,numberOfRows(tab) - 1
    do k = 0,4
      do j=0,3
        do i=0,2
          c1(i,j,k,l) = n
          c2(i,j,k,l) = c1(i,j,k,l)
          n = n + 1
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array3data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Array4Data`

PURPOSE

Get the real64 data from a column cell containing 4-dimensional array data.

INTERFACE

function real64ColumnArray4DataElement(column, row)

ARGUMENTS

- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellarray4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )
```

```
  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => real64Array4Data(col1,m)
    c2 => real64Array4Data(col2,m)
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l) = n
            c2(i,j,k,l) = c1(i,j,k,l)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellarray4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Array4Data`

PURPOSE

Get the real64 data from a column containing 4-dimensional array data.

INTERFACE

function real64ColumnArray4Data(column)

ARGUMENTS

- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=DOUBLE), dimension(:,:,:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_array4data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:,:,:), pointer :: c1, c2
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real64Array4Data(col1)
  c2 => real64Array4Data(col1)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_array4data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Attribute`

PURPOSE

Get the value of an attribute as a real64.

INTERFACE

function real64ArrayAttribute(array, name)
function real64AttributableAttribute(attributable, name)
function real64Attribute(attribute)
function real64BlockAttribute(Block, name)
function real64ColumnAttribute(column, name)
function real64DataSetAttribute(dataSet, name)
function real64TableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute
- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- character(len=*), intent(in) :: name
- type(TableT), intent(in) :: table

RETURNS

- real(kind=DOUBLE)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how real64 attributes are used.
! The program creates a dataset containing two real64 attributes,
! together with a table containing two real64 attributes.
! The attributes are then accessed, by name, with
! the real64Attribute() function.
! Also, it is shown how to access the attributes by position.
```

```
program example_real64attribute

  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"real1",1.0,"real comment")
  call setAttribute(set,"real2",2.0,"real comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"real1",3.0,"real comment")
  call setAttribute(tab,"real2",4.0,"real comment")

  write(*,*) real64Attribute( set, "real1" ) ! output '1.0'
  write(*,*) real64Attribute( set, "real2" ) ! output '2.0'
  write(*,*) real64Attribute( tab, "real1" ) ! output '3.0'
  write(*,*) real64Attribute( tab, "real2" ) ! output '4.0'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) real64Attribute( att ) ! output the sequence 1.0, 2.0
  end do

  call release(set)

end program example_real64attribute
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64Data`

PURPOSE

Get the real64 data from an array, column or column cell.

INTERFACE

function real64ArrayData(array)
function real64ColumnData(column)
function real64ColumnDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column

- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two 4-dimensional arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, and then the second column
! is output by accessing the column's data as a flat vector.
program example_real64data

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:,:,:,:), pointer :: c1, c2
  real(kind=DOUBLE), dimension(:), pointer :: cd
  integer, dimension(4), parameter :: s = (/ 3,4,5,6 /)
  integer :: i,j,k,l,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 5, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real64Array4Data(col1)
  c2 => real64Array4Data(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do l=0,5
      do k=0,4
        do j=0,3
          do i=0,2
```

```
            c1(i,j,k,l,m) = n
            c2(i,j,k,l,m) = c1(i,j,k,l,m)
            n = n + 1
          end do
        end do
      end do
    end do
  end do

  call release(col1)
  call release(col2)

  ! Output the col2
  cd => real64Data( col2 ) ! Access the column's 4-dimensional data as a flat vector.

  do n = 0,numberOfElements(col1) * numberOfRows(tab) - 1
    write(*,*) cd(n)
  end do

  call release(col2)
  call release(set)

end program example_real64data
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64VectorData`

PURPOSE

Get the real64 data from an array or column cell containing vector data.

INTERFACE

function real64ArrayVectorData(array)
function real64ColumnVectorDataElement(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second array has the same data type as the first; this
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised, on a row-by-row
! basis (i.e. accessing the column's data cell-by-cell),
! before the dataset is released (closed).
program example_cellvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 100, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers

  n = 0
  do m=0,numberOfRows(tab) - 1
    c1 => real64VectorData(col1,m)
    c2 => real64VectorData(col2,m)
    do i=0,2
      c1(i) = n
      c2(i) = c1(i)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_cellvectordata
! This example shows how to use the int64Array2Data interface.
! In the example a dataset is created (opened) containing
! a table with 2 vector arrays.
!
! The second array has the same data type as the first; this
```

```
! is ensured by using the arrayDataType() function to determine
! the data type of the first array.
!
! The array is then initialised,
program example_arrayvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ArrayT) arr1, arr2
  real(kind=DOUBLE), dimension(:), pointer :: a1, a2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray( set, "array1", REAL64, s, "km", "array comment" )
  arr2 = addArray( set, "array2", REAL64, s, "km", "array comment" )

  ! fill with unique numbers

  n = 0
  a1 => real64VectorData(arr1)
  a2 => real64VectorData(arr2)
  do i=0,2
    a1(i) = n
    a2(i) = a1(i)
    n = n + 1
  end do

  call release(arr1)
  call release(arr2)
  call release(set)

end program example_arrayvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`real64VectorData`

PURPOSE

Get the real64 data from a column containing vector data.

INTERFACE

function real64ColumnVectorData(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: row

RETURNS

- real(kind=DOUBLE), dimension(:,:), pointer

DESCRIPTION

ERRORS

EXAMPLES

```
! In this example add dataset is created (opened) containing
! a table with 2 columns of two vector arrays.
!
! The second column has the same data type as the first; this
! is ensured by using the columnDataType() function to determine
! the data type of the first array.
!
! The columns are then  initialised before the
! dataset is released (closed).
program example_columnvectordata

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col1, col2
  real(kind=DOUBLE), dimension(:,:), pointer :: c1, c2
  integer, dimension(1), parameter :: s = (/ 3 /)
  integer :: i,m,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  tab = addTable(set, "table", 10, "table comment" )
  col1 = addColumn( tab, "column1", REAL64, "km", s, "column comment" )
  col2 = addColumn( tab, "column2", columnDataType( col1 ), "km", s, "column comment" )

  ! fill with unique numbers
  c1 => real64VectorData(col1)
  c2 => real64VectorData(col2)

  n = 0
  do m=0,numberOfRows(tab) - 1
    do i=0,2
```

```
      c1(i,m) = n
      c2(i,m) = c1(i,m)
      n = n + 1
    end do
  end do

  call release(col1)
  call release(col2)
  call release(set)

end program example_columnvectordata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`relabel`

PURPOSE

Relabel an object.

INTERFACE

subroutine relabelArray(array, newLabel)
subroutine relabelAttributable(attributable, newLabel)
subroutine relabelAttribute(attribute, newLabel)
subroutine relabelBlock(block, newLabel)
subroutine relabelColumn(column, newLabel)
subroutine relabelDataSet(dataSet, newLabel)
subroutine relabelLabelled(labelled, newLabel)
subroutine relabelTable(table, newLabel)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute
- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- type(LabelledT), intent(in) :: labelled
- character(len=*), intent(in) :: newLabel
- type(TableT), intent(in) :: table

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none

  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att

  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )
```

```
  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display

program example_labelled

  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`release`

PURPOSE

Release an object.

INTERFACE

subroutine releaseArray(array)
subroutine releaseBlock(block)
subroutine releaseColumn(column)

subroutine releaseDataSet(dataSet)
subroutine releaseTable(table)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- type(TableT), intent(in) :: table

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

Most examples call the release functions.

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`rename`

PURPOSE

Rename an object.

INTERFACE

subroutine renameArray(array, newName)
subroutine renameAttributable(attributable, newName)
subroutine renameAttribute(attribute, newName)
subroutine renameBlock(block, newName)
subroutine renameColumn(column, newName)
subroutine renameDataSet(dataSet, newName)
subroutine renameLabelled(labelled, newName)
subroutine renameTable(table, newName)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute

- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- type(LabelledT), intent(in) :: labelled
- character(len=*), intent(in) :: newName
- type(TableT), intent(in) :: table

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the label, relabel, name and rename interfaces are used.
subroutine displayLabelled( l )
  use dal

  implicit none

  type(LabelledT), intent(in) :: l

  write(*,*) "the object with name ", name( l ), " has label: ", label(l)

end subroutine displayLabelled

subroutine display( set )
  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  type(AttributeT) att

  att = attribute( set, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  arr = array( set, 0, READ )
  write(*,*) name(arr), label( arr )
  call displayLabelled( labelled( arr ) )

  att = attribute( arr, 0 )
```

```
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  tab = table( set, 1 )
  write(*,*) name(tab), label( tab )
  call displayLabelled( labelled( tab ) )

  att = attribute( tab, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

  col = column( tab, 0, READ )
  write(*,*) name(col), label( col )
  call displayLabelled( labelled( col ) )

  att = attribute( col, 0 )
  write(*,*) name(att), label( att )
  call displayLabelled( labelled( att ) )

end subroutine display

program example_labelled

  use dal

  implicit none

  type(DataSetT) set
  type(ArrayT) arr
  type(TableT) tab
  type(ColumnT) col
  ! type(AttributeT) att
  ! integer(kind=int32), dimension(:,:,:), pointer :: a
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)

  ! create a set
  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"att1","value1","a dataset attribute comment")
  arr = addArray(set, "array", INTEGER32, comment="an array comment", dimensions=s )
  call setAttribute(arr,"att2","value2","an array attribute comment")
  tab = addTable(set, "table", 10, comment="a table comment" )
  call setAttribute(tab,"att3","value3","a table attribute comment")
  col = addColumn(tab,"int8",INTEGER8,comment="a column comment")
  call setAttribute(col,"TLMAX","value4","a column attribute comment")

  call display( set )
  call relabel( tab, "a new table comment" )
  call rename( col, "newcolnm" )
  call display( set )

  call release( set )

end program example_labelled
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

RowT

PURPOSE

A derived type which is used to declare objects of type RowT.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`scale`

PURPOSE

THIS INTERFACE IS NOT IMPLEMENTED. Get the scale factor of an object's data.

INTERFACE

function scaleOfArray(array)
function scaleOfColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=DOUBLE)

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

NAME

`setAttribute`

PURPOSE

Create/Copy/Set an attribute.

INTERFACE

subroutine arraySetAttribute(array, attribute)
subroutine attributableSetAttribute(attributable, attribute)
subroutine blockSetAttribute(block, attribute)
subroutine columnSetAttribute(column, attribute)
subroutine datasetSetAttribute(dataset, attribute)
subroutine setBooleanArrayAttribute(array, name, booleanValue, comment)
subroutine setBooleanAttributableAttribute(attributable, name, booleanValue, comment)
subroutine setBooleanAttribute(attribute, booleanValue, comment)
subroutine setBooleanBlockAttribute(block, name, booleanValue, comment)
subroutine setBooleanColumnAttribute(column, name, booleanValue, comment)
subroutine setBooleanDataSetAttribute(dataSet, name, booleanValue, comment)
subroutine setBooleanTableAttribute(table, name, booleanValue, comment)
subroutine setInt8ArrayAttribute(array, name, int8Value, units, comment)
subroutine setInt8AttributableAttribute(attributable, name, int8Value, units, comment)
subroutine setInt8Attribute(attribute, int8Value, units, comment)
subroutine setInt8BlockAttribute(block, name, int8Value, units, comment)
subroutine setInt8ColumnAttribute(column, name, int8Value, units, comment)
subroutine setInt8DataSetAttribute(dataSet, name, int8Value, units, comment) subroutine setInt8TableAttribute(table, name, int8Value, units, comment)
subroutine setInt16ArrayAttribute(array, name, int16Value, units, comment)
subroutine setInt16AttributableAttribute(attributable, name, int16Value, units, comment) subroutine setInt16Attribute(attribute, int16Value, units, comment)
subroutine setInt16BlockAttribute(block, name, int16Value, units, comment)
subroutine setInt16ColumnAttribute(column, name, int16Value, units, comment)
subroutine setInt16DataSetAttribute(dataSet, name, int16Value, units, comment)
subroutine setInt16TableAttribute(table, name, int16Value, units, comment)
subroutine setInt32ArrayAttribute(array, name, int32Value, units, comment)
subroutine setInt32AttributableAttribute(attributable, name, int32Value, units, comment)
subroutine setInt32Attribute(attribute, int32Value, units, comment)
subroutine setInt32BlockAttribute(block, name, int32Value, units, comment)
subroutine setInt32ColumnAttribute(column, name, int32Value, units, comment)
subroutine setInt32DataSetAttribute(dataSet, name, int32Value, units, comment)
subroutine setInt32TableAttribute(table, name, int32Value, units, comment)
subroutine setReal32ArrayAttribute(array, name, real32Value, units, comment)
subroutine setReal32AttributableAttribute(attributable, name, real32Value, units, comment)
subroutine setReal32Attribute(attribute, real32Value, units, comment)
subroutine setReal32BlockAttribute(block, name, real32Value, units, comment)
subroutine setReal32ColumnAttribute(column, name, real32value, units, comment)
subroutine setReal32DataSetAttribute(dataSet, name, real32Value, units, comment)
subroutine setReal32TableAttribute(table, name, real32Value, units, comment)
subroutine setReal64ArrayAttribute(array, name, real64Value, units, comment)
subroutine setReal64AttributableAttribute(attributable, name, real64Value, units, comment)
subroutine setReal64Attribute(attribute, real64Value, units, comment)
subroutine setReal64BlockAttribute(block, name, real64Value, units, comment)
subroutine setReal64ColumnAttribute(column, name, real64Value, units, comment)
subroutine setReal64DataSetAttribute(dataSet, name, real64Value, units, comment)
subroutine setReal64TableAttribute(table, name, real64Value, units, comment)
subroutine setStringArrayAttribute(array, name, stringValue, comment)
subroutine setStringAttributableAttribute(attributable, name, stringValue, comment)

subroutine setStringAttribute(attribute, stringValue, comment)
subroutine setStringBlockAttribute(block, name, stringValue, comment)
subroutine setStringColumnAttribute(column, name, stringValue, comment)
subroutine setStringDataSetAttribute(dataSet, name, stringValue, comment)
subroutine setStringTableAttribute(table, name, stringValue, comment)
subroutine tableSetAttribute(table, attribute)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute
- type(BlockT), intent(in) :: block
- logical, intent(in) :: booleanValue
- type(ColumnT), intent(in) :: column
- character(len=*), intent(in), optional :: comment
- type(DataSetT), intent(in) :: dataset
- type(TableT), intent(in) :: table
- character(len=*), intent(in), optional :: units
- integer(kind=INT8), intent(in) :: int8Value
- integer(kind=INT16), intent(in) :: int16Value
- integer(kind=INT32), intent(in) :: int32Value
- real(kind=SINGLE), intent(in) :: real32Value
- real(kind=DOUBLE), intent(in) :: real64Value
- character(len=*), intent(in) :: stringValue

RETURNS N/A

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the setAttribute
! interface is used.
program example_setattribute

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet("test.dat",CREATE)

  call setAttribute(set,"test1","some value","some comment to the attribute")
  call setAttribute(set,"TELESCOP","XMM","Telescope (mission) name")

  write(*,*) numberOfAttributes( set ) ! 2 attributes

  call release(set)

end program example_setattribute
```

SEE ALSO

`attribute` AttributeT

BUGS AND LIMITATIONS

None known.

NAME

`setAttributes`(attributable, origin)

PURPOSE

Replace the attributes in an attributable set with the attributes in a source set.

ARGUMENTS

- type(AttributableT), intent(in) :: attributable
- type(AttributableT), intent(in) :: origin

RETURNS

DESCRIPTION

The attributes in source are copied to destination. Attributes, which have the same name are overwritten.

ERRORS

EXAMPLES

```
! This example shows how the setAttributes interface
! is used.
program example_setattributes
```

```
  use dal
  use errorhandling
  implicit none

  type(DataSetT) set
  type(TableT) tab

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"sbool1",.false.,"dataset bool comment")
  call setAttribute(set,"sbool2",.false.,"dataset bool comment")

  tab = addTable(set,"table",10);
  call setAttributes(attributable(tab),attributable(set))

  write(*,*) numberOfAttributes( tab ) ! 2 attributes
  call release(set)

end program example_setattributes
```

SEE ALSO

`addAttributes`

BUGS AND LIMITATIONS

None known.

NAME

`setData`

PURPOSE

Set the data in a variable length column.

INTERFACE

subroutine setBoolCell(column, row, booleanValues)
subroutine setInt8Cell(column, row, int8Values)
subroutine setInt16Cell(column, row, int16Values)
subroutine setInt32Cell(column, row, int32Values)
subroutine setReal32Cell(column, row, real32values)
subroutine setReal64Cell(column, row, real64Values)
subroutine setStringVariableCell(column, row, stringValues)

ARGUMENTS

- logical(kind=BOOL), dimension(:), intent(in) :: booleanValues
- type(ColumnT), intent(in) :: column
- integer(kind=INT8), dimension(:), intent(in) :: int8Values
- integer(kind=INT16), dimension(:), intent(in) :: int16Values
- integer(kind=INT32), dimension(:), intent(in) :: int32Values
- real(kind=SINGLE), dimension(:), intent(in) :: real32Values

- real(kind=DOUBLE), dimension(:), intent(in) :: real64Values
- integer(kind=INT32), intent(in) :: row
- character(len=*) :: stringValues

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how to set the data in
! a variable length column.
program example_setdata

  use dal

  implicit none

  integer, parameter :: nRows = 10
  integer, parameter :: maxCellSize = 100
  integer, dimension(0) :: zeroSize
  integer(kind=INT32) :: i
  type(DataSetT) :: set
  type(TableT) :: tab
  type(ColumnT) :: i8col1, i8col2, i16col1, i16col2, i32col1, i32col2
  type(ColumnT) :: r32col1, r32col2, r64col1, r64col2
  type(ColumnT) :: scol1, scol2, bcol1, bcol2

  logical(kind=bool), dimension(maxCellSize) :: b
  integer(kind=INT8), dimension(maxCellSize) :: i8
  integer(kind=INT16), dimension(maxCellSize) :: i16
  integer(kind=INT32), dimension(maxCellSize) :: i32
  real(kind=SINGLE), dimension(maxCellSize) :: r32
  real(kind=DOUBLE), dimension(maxCellSize) :: r64
  character(len=maxCellSize) :: s

  real(kind=SINGLE), dimension(:), pointer :: r32Data

  s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  do i = 1, maxCellSize
    i8(i) = i
    i16(i) = i
    i32(i) = i
    r32(i) = i
    r64(i) = i
    b(i) = ((i / 2).eq.0 )
  end do
```

```
  set = dataSet("test.dat",Create)
  tab = addTable(set,"someTable",nRows)

  bcol1 = addColumn(tab,"bcol1",Boolean, &
dimensions=zeroSize,comment="bool data")

  bcol2 = addColumn(tab,"bcol2",Boolean, &
dimensions=zeroSize,comment="bool data")

  i8col1 = addColumn(tab,"i8col1",Integer8,units="m", &
dimensions=zeroSize,comment="int8 data")

  i8col2 = addColumn(tab,"i8col2",Integer8,units="m", &
dimensions=zeroSize,comment="int8 data")

  i16col1 = addColumn(tab,"i16col1",Integer16,units="m", &
dimensions=zeroSize,comment="int16 data")

  i16col2 = addColumn(tab,"i16col2",Integer16,units="m", &
dimensions=zeroSize,comment="int16 data")

  i32col1 = addColumn(tab,"i32col1",Integer32,units="m", &
dimensions=zeroSize,comment="int32 data")

  i32col2 = addColumn(tab,"i32col2",Integer32,units="m", &
dimensions=zeroSize,comment="int32 data")

  r32col1 = addColumn(tab,"r32col1",Real32,units="m", &
dimensions=zeroSize,comment="real32 data")

  r32col2 = addColumn(tab,"r32col2",Real32,units="m", &
dimensions=zeroSize,comment="real32 data")

  r64col1 = addColumn(tab,"r64col1",Real64,units="m", &
dimensions=zeroSize,comment="real64 data")

  r64col2 = addColumn(tab,"r64col2",Real64,units="m", &
dimensions=zeroSize,comment="real64 data")

  scol1 = addColumn(tab,"scol1",String,units="m", &
dimensions=zeroSize,comment="string data")

  scol2 = addColumn(tab,"scol2",String, &
dimensions=zeroSize,comment="string data")

  do i=0,nRows - 1
   call setData( bcol1, i, b( 1 : i + 1 ))
   call setData( bcol2, i, b( 1 : nRows - i ))
   call setData( i8col1, i, i8( 1 : i + 1 ))
   call setData( i8col2, i, i8( 1 : nRows - i ))
   call setData( i16col1, i, i16( 1 : i + 1 ))
   call setData( i16col2, i, i16( 1 : nRows - i ))
   call setData( i32col1, i, i32( 1 : i + 1 ))
   call setData( i32col2, i, i32( 1 : nRows - i ))
   call setData( r32col1, i, r32( 1 : i + 1 ))
```

```
     call setData( r32col2, i, r32( 1 : nRows - i ))
     call setData( r64col1, i, r64( 1 : i + 1 ))
     call setData( r64col2, i, r64( 1 : nRows - i ))
     call setData( scol1, i, s( 1 : i + 1 ))
     call setData( scol2, i, s( 1 : nRows - i ))
    end do

    call release( set )

    set = dataSet("test.dat",Modify)
    tab = table(set,"someTable")

    bcol1 = column(tab,"bcol1",Read)
    bcol2 = column(tab,"bcol2",Read)
    i8col1 = column(tab,"i8col1",Read)
    i8col2 = column(tab,"i8col2",Read)
    i16col1 = column(tab,"i16col1",Read)
    i16col2 = column(tab,"i16col2",Read)
    i32col1 = column(tab,"i32col1",Read)
    i32col2 = column(tab,"i32col2",Read)
    r32col1 = column(tab,"r32col1",Read)
    r32col2 = column(tab,"r32col2",Read)
    r64col1 = column(tab,"r64col1",Read)
    r64col2 = column(tab,"r64col2",Read)
    scol1 = column(tab,"scol1",Read)
    scol2 = column(tab,"scol2",Read)

    do i = 0, nRows - 1
      write(*,*) boolData( bcol1, i )
      write(*,*) boolData( bcol2, i )
      write(*,*) int8Data( i8col1, i )
      write(*,*) int8Data( i8col2, i )
      write(*,*) int16Data( i16col1, i )
      write(*,*) int16Data( i16col2, i )
      write(*,*) int32Data( i32col1, i )
      write(*,*) int32Data( i32col2, i )
      write(*,*) real32Data( r32col1, i )
      write(*,*) real32Data( r32col2, i )
      write(*,*) real64Data( r64col1, i )
      write(*,*) real64Data( r64col2, i )
      write(*,*) stringCell( scol1, i )
      write(*,*) stringCell( scol2, i )
    end do

    call release( set )

  end program example_setdata
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`setExists`(setName)

PURPOSE

Determine if a dataset exists.

ARGUMENTS

- character(len=*), intent(in) :: setName

RETURNS

- logical

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the setexists()
! function is used.
program example_setexists

  use dal

  implicit none

  type(DataSetT) set

  set = dataSet("test.dat",CREATE)
  call release(set)

  if( setExists( "test.dat" ) ) then
    write(*,*) 'Very strange'
  end if
end program example_setexists
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`setNullValue`

PURPOSE

Set the value of the integer null value.

INTERFACE

subroutine setNullValue(array, value)
subroutine setNullValue(column, value)
subroutine setNullValue(dataComponent, value)
subroutine setNullValue(nullable, value)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array whose null value is to be set.
- type(ColumnT), intent(in) :: column
  A handle of the column whose null value is to be set.
- type(DataComponentT), intent(in) :: dataComponent
  A handle of the dataComponent whose null value is to be set.
- type(NullableT), intent(in) :: nullable
  A handle of the nullable whose null value is to be set.
- integer(kind=INT32), intent(in) :: value
  The value of the null value.

RETURNS

DESCRIPTION

This function is only relevant for objects containing integer data, and should not be called for objects containing other data types.

The null value of an object containing integer data (if it has been defined) may be obtained with the function intNullValue().

The logical function nullDefined() may be used to determine if the null value has been defined.

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )

end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
```

```
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
       ! Would have given an error, if the null
       ! value of array arr1 had not been set.

  if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
    write(*,*) "Using null value of arr1, in arr2"
    call setNullValue( arr2, intNullValue( arr1 ))
  else
    call setNullValue( arr2, 999999 )
  end if

  call check( nullable( arr2 ) )

  call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

  call release(arr1)
  call release(arr2)

  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.
```

```
  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNotNull` `isNull` `nullable` `nullDefined` `nullType` `setToNull`

BUGS AND LIMITATIONS

None known.

NAME

`setScaling`

PURPOSE

NOT IMPLEMENTED. Set the scaling parameters to be applied to an object's data.

INTERFACE

subroutine setScalingOfArray(array, zero, scale, toType)
subroutine setScalingOfColumn(column, zero, scale, toType)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- real(kind=DOUBLE), intent(in) :: scale
- integer, intent(in) :: toType
- real(kind=DOUBLE), intent(in) :: zero

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

NAME

`setStringCell`(column, row, value)

PURPOSE

Set a cell in a string column.

ARGUMENTS

- type(ColumnT), intent(in) :: column
  A handle to the column which contains the cell to be set.
- integer(kind=INT32), intent(in) :: row
  Set row number of the cell to be set.
- character(len=*) :: value
  This value will be copied into the specified cell.

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the setStringCell()
! function is used.
program example_setstringcell

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  character(len=12) :: s
  integer i

  s = "abcdef"

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))
  do i=0,numberOfRows(tab)-1
    write(s,'(A6,I2)') "string",i
    call setStringCell(col,i,s)
```

```
    write(*,*) stringCell( col, i )
  end do

  call release(set)

end program example_setstringcell
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`setToNull`

PURPOSE

Set a value to null.

INTERFACE

subroutine setToNullArray(array, position)
subroutine setToNullCell(column, row, position)
subroutine setToNullColumn(column, row)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of the array containing the value to be set.
- type(ColumnT), intent(in) :: column
  A handle of the column containing the value to be set.
- integer(kind=INT32), intent(in) :: position
  The position of the value within the array (or the column cell in the case of a multi-dimensional column) which is to be set.
- integer(kind=INT32), intent(in) :: row
  The row number of the column cell containing the value to be set.

RETURNS

DESCRIPTION

In the case of integer values, an error will be raised if the object's null value has not been defined. The null value of an object containing integer data may be set with a call to setNullValue(). The logical function nullDefined() determines if the null value of an object has been defined.

ERRORS

EXAMPLES

```
! This example shows how null values are used.
subroutine check( thisNullable )
  use dal
  type(NullableT), intent(in) :: thisNullable

  write(*,*) "Null defined?: ", nullDefined( thisNullable ), nullType( thisNullable )
```

```
end subroutine check

program example_nullvalues

  use dal
  use errorhandling

  implicit none

  type(DataSetT) set
  type(ArrayT) arr1, arr2
  type(TableT) tab
  type(ColumnT) col1, col2
  integer(kind=int32), dimension(:), pointer :: i32
  real(kind=double), dimension(:), pointer :: r64
  integer(kind=int32), dimension(:,:,:), pointer :: a1, a2
  integer, dimension(3), parameter :: s = (/ 3,4,2 /)
  integer :: i,j,k,n

  ! create a set
  set = dataSet("test.dat",CREATE)
  arr1 = addArray(set, "array1", INTEGER32, dimensions=s )
  arr2 = addArray(set, "array2", arrayDataType( arr1 ), dimensions=s )

  ! fill with unique numbers
  a1 => int32Array3Data(arr1)
  a2 => int32Array3Data(arr1)

  n = 0
  do k=0,1
    do j=0,3
      do i=0,2
        a1(i,j,k) = n
        a2(i,j,k) = a1(i,j,k) + 1
        n = n + 1
      end do
    end do
  end do

  call setNullValue( arr1, 999999 )
  call check( nullable( arr1 ) )

  call setToNull( arr1, 0 ) ! Set the first element of array arr1 to null.
       ! Would have given an error, if the null
       ! value of array arr1 had not been set.

  if( nullType( arr1 ) .eq. INTEGER_NULL ) then !
    write(*,*) "Using null value of arr1, in arr2"
    call setNullValue( arr2, intNullValue( arr1 ))
  else
    call setNullValue( arr2, 999999 )
  end if

  call check( nullable( arr2 ) )
```

```
  call setToNull( arr2, 1 ) ! Set the second element of array arr2 to null.
! Would have given an error, if the null
! value of array arr2 had not been set.

  call release(arr1)
  call release(arr2)

  tab = addTable(set,"some table",100)

  col1 = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")

  i32 => int32Data(col1)
  do i=0,numberOfRows(tab)-1
    i32(i) = 3*i
  end do
  call setNullValue( col1, 999999 )
  call check( nullable( col1 ) )

  call setToNull( col1, 0 ) ! Set the first element of column col1 to null.


  col2 = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
  r64 => real64Data(col2)
  do i=0,numberOfRows(tab)-1
    r64(i) = 0.25*i
  end do

  ! col is a non-integer column and it would be an
  ! an error to call setNullValue().
  call check( nullable( col2 ) )

  call setToNull( col2, 0 ) ! Set the first element of column col2 to null.

  if( hasNulls( col2 ) ) then
    do i=0,numberOfRows(tab)-1
      if( isNull( col2, i ) ) then
        write(*,*) "element", i, "is null"
      else
        write(*,*) "element", i, "is", r64(i)
      endif
    end do
  endif

  call release(col1)
  call release(col2)
  call release(set)

end program example_nullvalues
```

SEE ALSO

`hasNulls` `intNullValue` `isNotNull` `isNull` `nullable` `nullDefined` `nullType` `setNullValue`

BUGS AND LIMITATIONS

None known.

NAME

`setUnits`

PURPOSE

Set the units of an attribute, array or column.

INTERFACE

subroutine setArrayAttributeUnits(array, attributeName, units)
subroutine setArrayUnits(array, units)
subroutine setAttributableAttributeUnits(attributable, attributeName, units)
subroutine setAttributeUnits(attribute, units)
subroutine setBlockAttributeUnits(block, attributeName, units)
subroutine setColumnAttributeUnits(column, attributeName, units)
subroutine setColumnUnits(column, units)
subroutine setDataSetAttributeUnits(dataSet, attributeName, units)
subroutine setTableAttributeUnits(table, attributeName, units)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- character(len=*), intent(in) :: attributeName
- type(AttributeT), intent(in) :: attribute
- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- type(TableT), intent(in) :: table
- character(len=*), intent(in) :: units

RETURNS

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the setUnits interface
! is used.
program example_setunits

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
```

```
  call release(set)

  set = dataSet("test.dat",MODIFY)
  tab = table( set, 0 )
  col = column( tab, 0, MODIFY )

  write(*,*) units( col )
  call setUnits( col, "mm" )
  write(*,*) units( col )

  call release(set)

end program example_setunits
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

SINGLE

PURPOSE

An enumeration value which is used to indicate single precision (real32) data.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

STRING

PURPOSE

An enumeration value which is used to indicate data of type character string.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`stringAttribute`

PURPOSE
Get the value of an attribute as a character string.

INTERFACE
function stringArrayAttribute(array, name)
function stringAttribute(attribute)
function stringAttributableAttribute(attributable, name)
function stringBlockAttribute(Block, name)
function stringColumnAttribute(column, name)
function stringDataSetAttribute(dataSet, name)
function stringTableAttribute(table, name)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(AttributableT), intent(in) :: attributable
- type(AttributeT), intent(in) :: attribute
- type(BlockT), intent(in) :: block
- type(ColumnT), intent(in) :: column
- type(DataSetT), intent(in) :: dataSet
- character(len=*), intent(in) :: name
- type(TableT), intent(in) :: table

RETURNS

- character(len=stringAttributeLength)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how string attributes are used.
! The program creates a dataset containing two string attributes,
! together with a table containing two string attributes.
! The attributes are then accessed, by name, with
! the stringAttribute() function.
! Also, it is shown how to access the attributes by position.
program example_stringattribute

  use dal
  use errorhandling
  implicit none
```

```
  type(DataSetT) set
  type(TableT) tab
  type(AttributeT) att
  integer i

  set = dataSet("test.dat",CREATE)
  call setAttribute(set,"string1","abcdef","string comment")
  call setAttribute(set,"string2","ghijkl","string comment")

  tab = addTable(set,"table",10);
  call setAttribute(tab,"string1","abcdef","string comment")
  call setAttribute(tab,"string2","ghijkl","string comment")

  write(*,*) stringAttribute( set, "string1" ) ! output 'abcdef
  write(*,*) stringAttribute( set, "string2" ) ! output 'ghijkl'
  write(*,*) stringAttribute( tab, "string1" ) ! output 'abcdef
  write(*,*) stringAttribute( tab, "string2" ) ! output 'ghijkl'

  do i = 0, numberOfAttributes( set ) - 1
   att = attribute( set, i )
write(*,*) stringAttribute( att ) ! output the sequence 'abcdef', 'ghijkl'
  end do

  call release(set)

end program example_stringattribute
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`stringCell`(column, row)

PURPOSE

Get the character string data from a column cell.

ARGUMENTS

- type(ColumnT), intent(in) :: column
- integer(kind=INT32), intent(in) :: row

RETURNS

- character(len=columnStringCellLength)

DESCRIPTION

ERRORS

EXAMPLES

```
! This example shows how the stringCell()
! function is used.
program example_stringcell

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  character(len=12) :: s
  integer i

  s = "abcdef"

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/12/))
  do i=0,numberOfRows(tab)-1
    write(s,'(A6,I2)') "string",i
    call setStringCell(col,i,s)
    write(*,*) stringCell( col, i )
  end do

  call release(set)

end program example_stringcell
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`subTable`(table, from, to)

PURPOSE

Get a subtable from a table.

ARGUMENTS

- type(TableT), intent(in) :: table
- integer, intent(in), optional :: from
- integer, intent(in), optional :: to

RETURNS

- type(SubTableT)

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

NAME

SubTableT

PURPOSE

A derived type used to declare SubTable handles.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

TABLE_BLOCK

PURPOSE

An enumeration value which is used to indicate a table.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`table`

PURPOSE

Get a table from a dataset.

INTERFACE

function tableWithName(dataSet, tableName)
function tableWithNumber(dataSet, tableNumber)

ARGUMENTS

- type(DataSetT), intent(in) :: dataSet
- character(len=*), intent(in) :: tableName
- integer, intent(in) :: tableNumber

RETURNS

- type(TableT)

DESCRIPTION

The table may be specified either by number (the first block in a dataset has position zero) or by name.

ERRORS

EXAMPLES

```
! This example shows how the table
! interface is used.
program example_table

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  integer i

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"table1",10)
  tab = addTable(set,"table2",100)
  tab = addTable(set,"table3",1000)

  do i=0,numberOfBlocks( set ) - 1
    tab = table( set, i ) ! Access table by number
    write(*,*) name( tab )
  end do

  tab = table( set, "table1" ) ! Access table by name
  write(*,*) name( tab )

  call release(set)

end program example_table
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

TableT

PURPOSE

A derived type which is used to declare Table handles.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

TEMP

PURPOSE

An enumeration value which is used to indicate temporary access to an object.

DESCRIPTION

All changes made to an object, which has TEMP access, will be discarded, when the object is released.

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`units`

PURPOSE

Get the units of an object.

INTERFACE

function arrayAttributeUnits(array, name)
function arrayUnits(array)
function attributableAttributeUnits(attributable, name)
function blockAttributeUnits(block, name)
function columnAttributeUnits(column, name)
function columnUnits(column) function dataComponentUnits(dataComponent)
function dataSetAttributeUnits(dataSet, name)
function tableAttributeUnits(table, name) function unitsOfAttribute(attribute)

ARGUMENTS

- type(ArrayT), intent(in) :: array
  A handle of an array from which to get an attribute's units.
- type(AttributableT), intent(in) :: attributable
  A handle of an attributable from which to get an attribute's units.
- type(AttributeT), intent(in) :: attribute
  A handle of an attribute from which to get the units.
- type(BlockT), intent(in) :: block
  A handle of a block from which to get an attribute's units.
- type(ColumnT), intent(in) :: column
  A handle of a column from which to get an attribute's units.
- type(DataComponentT), intent(in) :: dataComponent
  A handle of a dataComponent.
- type(DataSetT), intent(in) :: dataSet
  A dataset handle from which to get an attribute's units.
- character(len=*), intent(in) :: name
  The name of the attribute.
- type(TableT), intent(in) :: table
  A table handle from which to get an attribute's units.

RETURNS

- character(len=IdentifierLength)

DESCRIPTION

ERRORS

EXAMPLES

```
program example_columnunits

  use dal

  implicit none

  type(DataSetT) set
  type(TableT) tab
  type(ColumnT) col
  integer i, coltype

  set = dataSet("test.dat",CREATE)
  tab = addTable(set,"some table",100)

  col = addColumn(tab,"bool",BOOLEAN)
  col = addColumn(tab,"int8",INTEGER8,units="cm",comment="int8 column")
  col = addColumn(tab,"int16",INTEGER16,units="dm",comment="int16 column")
  col = addColumn(tab,"int32",INTEGER32,units="m",comment="in32 column")
  col = addColumn(tab,"real32",REAL32,units="Dm",comment="real32 column")
  col = addColumn(tab,"real64",REAL64,units="hm",comment="real64 column")
```

```
  col = addColumn(tab,"string",STRING,comment="string column",dimensions=(/80/))

  do i=0, numberOfColumns( tab ) - 1
    col = column( tab, i, READ )
    coltype = columnDataType( col )
    if(coltype.eq.INTEGER8.or.coltype.eq.INTEGER16.or.coltype.eq.INTEGER32 &
    .or.coltype.eq.REAL32.or.coltype.eq.REAL64) then
      write(*,*) units( col )
    end if
  end do

  call release(set)

end program example_columnunits
```

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`unsetScaling`

PURPOSE

NOT IMPLEMENTED. Remove the scaling factors from an object.

INTERFACE

subroutine unsetScalingOfArray(array, toType)
subroutine unsetScalingOfColumn(column, toType)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column
- integer, intent(in) :: toType

RETURNS

None

DESCRIPTION

toType specifies the data type which the object should have after the (un)scaling has been performed.

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

NAME

USE_ENVIRONMENT

PURPOSE

An enueration value which is used to indicates that the users environment should be used to establish which option ahould be taken.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

WRITE

PURPOSE

An enumeration which is used to indicate that an object should be accessed with read and modify permissions.

DESCRIPTION

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

`zero`

PURPOSE

NOT IMPLEMENTED. Get the scaling origin from an object.

INTERFACE

function zeroOfArray(array)
function zeroOfColumn(column)

ARGUMENTS

- type(ArrayT), intent(in) :: array
- type(ColumnT), intent(in) :: column

RETURNS

- real(kind=DOUBLE)

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

! Extended DAL

subroutine subTableSeek(table, from, count) type(SubTableT), intent(in) :: table integer, intent(in) :: from, count

call error("", errorMessage)

end subroutine

end module Dal

19 Errors

blockExists An attempt was made to add a block with the name of an exisitng block. invalidBlockPosition The position is invalid.

20 C++ API

Abstract interface definition for DAL

CLASS
: Dal

PURPOSE
: Information used by all Dal objects.

DERIVED FROM
: None.

TYPES

enum DataType { Bool = 1, Int8, Uint16, Int16, Uint32, Int32, Real32, Real64, DString }
: Used to specify the data type of objects. These enumeration values represent the seven fundamental data types of the DAL Data Model. These types have the following meanings:

- Bool An 8-bit boolean object taking the values 0 (false) or 1 (true).

- Int8 An 8-bit integer object with values in the range [...]
- Uint16 A 16-bit unsigned integer object with values in the range [...]
- Int16 A 16-bit integer object with values in the range [...]
- Uint32 A 32-bit unsigned integer object with values in the range [...]
- Int32 A 32-bit integer object with values in the range [...]
- Real32 A 32-bit real object with values in the range [...]
- Real64 A 64-bit real object with values in the range [...]
- DString An array of character values.

enum AccessMode Read = 1, Create, Modify, Temp, AsParent

The AccessMode determines whether the data is read upon open and written upon close.

- Create Indicates that a new dataset is to be created. In the event that a dataset already exists with the specified name, the subsequent behaviour is determined by the setting of the SAS_CLOBBER environment variable.
- Modify Indicates that a dataset, table or column is to be modified.
- Read Indicates that a dataset, table or column is to be accessed but not modified.
- Temp Indicates that a dataset, table or column is to be accessed, but all modifications made will be discarded upon closure.

DATA

static const vector¡unsigned long¿ scalar

This data value is used to indicate scalar dimension.

static const vector¡unsigned long¿ zero

This data value is used to indicate a starting position or a zero length.

CLASS

DataSetServer

PURPOSE

The DataServer is an abstract interface for an object that keeps track of opening and closing data files. It will implement some strategy to determine which part of the file is kept in memory. The open mode can be used as a hint how to deal with the file.

DERIVED FROM

public virtual Dal

TYPES

enum FileType { Fits = 1, Dal, Deceit }

These enumeration are used to specify the output file type of new datasets. The values have the following meaning:

- Fits The output will be compliant to standard FITS format. This format is guaranteed to be platform independent.
- Dal The output will be a Dal specific format, closely related to the internal format of the Dal's data structures. This format will give the best performance, but is not guaranteed to be platform independent.
- Deceit A special format, not for general use, which will comply as far as is possible to the deceit-file format. This option is not implemented in the core implementation, and requires the extended Dal.

enum MemoryModel { High = 1, HighLow, Low, UseEnvironment }

These enumeration values are used to specify which memory model a particular dataset should be opened with. The values have the following meaning:

- High The high memory model will be used. The dataset will be loaded into memory in it's entirety. All subsequent dataset operations will be performaed on the memory-loaded version of the dataset. Upon closure, the memory is flushed back to disk. This option gives rise to high performance, but assumes that the machines core memory wil not be exhausted.
- HighLow This option should be used when the machines core memory is limited. When a dataset is opened with this option the data (arrays and tables) is not loaded. Only when the data is accessed is it loaded. When the data isrelased it will be flushed back to disk.
- Low This option is not implemented and is for future use. The intention is that a dataset opened with this option is guaranteed to work on a machine with very low memory.

METHODS

virtual void client(const string& name) = 0

Tell the DataServer who is accessing the datasets i.e. name of the client; can be any arbitrary string. It is likey that this name will be written any createdi or modifed datasets.

virtual const string& client() const = 0

Get the value which was set by the client(const string&) method.

virtual void process(const string& processDescription) = 0

Register a description of the process that is going on. Can be any arbitrary string. It is likely that this desciprtion will be writtem to any created or modified datasets.

virtual const string& process() const = 0

Get the process description string.

virtual void process(
const DataSet* dataSet,
const string& processDescription) = 0

Register a description of the process that is going on, for the given dataset. Multiple invocations of this method is cummulative giving rise to an ordered list of descriptions. The description will be written to the dataset upon closure (provided it was opened for create or modify).

virtual const string& process(
const DataSet* dataSet,
unsigned int processNumber) const = 0

Get the process description string with the given ordinal number for the given dataSet.

virtual unsigned int processes(const DataSet* dataSet) const = 0

Get the number of process description strings for the given dataset.

virtual DataSet * open(
const string & setName,
AccessMode openMode,
MemoryModel memoryModel = UseEnvironment,
DataSetReaderWriter* readerWriter) = 0

Opens a dataset. This is how datasets are created, read or modified. The pointer which is returned by other methods to create additional components or access existing components. In particular, this pointer must be passed to the close() method in order to release the dataset from memory.

If a dataset is opened for modify or read, the DAL attempts to detect the format . The format may be determined with the ouputFileFormat() method. The output file format of newly created datasets may be specified by setting the SAS_FORMAT environment variable appropriately. The final argument, is currently only prototyped and should be ignored.

virtual DataSet * clone(
const string & from,
const string & to,
AccessMode openMode,
MemoryModel memoryModel = UseEnvironment,
DataSetReaderWriter* readerWriter) = 0

Clones a dataset. This method opens a dataset with the specified name (to) and specified mode (either Dal::Modify or Dal::Temp) and fills it with the contents of the given source dataset (from). The pointer which is returned by other methods to create additional components or access existing components. In particular, this pointer must be passed to the close() method in order to release the dataset from memory.

virtual void close(DataSet * dataSet) = 0
Closes the specified dataset.

virtual void keep(const string & setName) = 0
Tell the dataset server not to discard the named dataset.

NB. This method must only be called by Meta Tasks.

virtual void discard(const string & setName) = 0
Tell the dataset server to discard the named dataset.

NB. This method must only be called by Meta Tasks.

virtual bool exists(const string & setName) const = 0
Determines if the dataset with the given name exists, in which case, true is returned. .Otherwise false is returned.

virtual void copy(const string& from, const string& to) = 0
Copies the dataset with name from, to the dataset with name to.

virtual void clobber(bool b) = 0
Activate or Deactivate the clobber mechasism. If the clobber mechanism is activated, then datasets are overwritten when new datasets are created with the same name. When the machism is off, it is not possible to overwrite existing datasets, and any attempt to do so will give rise to an error.

virtual bool clobber() const = 0
Retursn the current clobber mechanism setting: ture on false off.

virtual void outputFileFormat(FileType fileType) = 0
Sets the output file format.

virtual FileType outputFileFormat() const = 0
Get the output file format.

virtual void printOn(ostream& os) const = 0

FUNCTION

ostream& operator¡¡(ostream& os, const DataSetServer& d)

PURPOSE

DATA

extern DataSetServer* dataSetServer;

PURPOSE

Single global instance of a DataSetServer.

CLASS

Labelled

PURPOSE

An object which is derived from Labelled will have a name, together with an associated short (typically one line) textual description (which is essentially comment).

DERVIED FROM

None

METHODS

virtual const string & name() const = 0

The name of the object is obtained with name()

virtual void rename(const string & newName) = 0

The object may be renamed with rename() and

virtual const string & label() const = 0

The short textual description is obtained with label()

virtual void relabel(const string & newLabel) = 0

The short textual description may be changed (i.e. replaced) with relabel().

virtual string qualifiedName() const = 0

qualifiedName() returns a colon-separated concatenation of the names, in hierarchical order, or the object and all its ancesters, up to and including the data set name.

CLASS

Attribute

PURPOSE

An attribute is an object which consists of a name, value, comment and units. Although for non numeric values, the units are meaningless. The name and comment methods are provided by the Labelled base class. An attribute belongs to an attributable object. The Attributable class is the ,managing class of attributes. The owner of a particular attribute is obtained with the parent() method. An attribute is created with Attributable::addAttribute().

DERIVED FROM

public virtual Dal

public virtual Labelled

public virtual Child¡Attributable¿

TYPES

enum DataType { Bool=1, Int, Real, DString }

The fundamental types of an attribute's value.

METHODS

virtual Attribute() {}

virtual Attribute & operator=(int value) = 0

Assign an integer value to the attribute. The current value is lost, and the data type becomes Attribute::Int.

virtual Attribute & operator=(double value) = 0
: Assign a real value to the attribute. The current value is lost and the data type becomes Attribute::Real.

virtual Attribute & operator=(bool value) = 0
: Assign a boolean to the attribute. The current value is lost and the type becomes Attribute::Bool.

virtual Attribute & operator=(const char * value) = 0
: Assign a character string to the attribute. The current value is lost and the data type becomes Attribute::DString.

virtual Attribute & operator=(const string & value) = 0
: Assign a string to the attribute. The current value is lost and the data type becomes DString.

virtual Attribute & operator=(const Attribute&) = 0
: Assignment operator. The value, type, comment and units are assign to the attribute. The owner of the attribute remains unchanged.

virtual int asInt() const = 0
: Returns the value of the attribute as an integer, converting it if necessary. An error is generated in case it is not possible to convert value to an int.

virtual double asReal() const = 0
: Returns the value of the attribute as a real, converting it if necessary. An error is generated in case it is not possible to convert value to a real.

virtual string asString() const = 0
: Returns the value of the attribute as a string, converting it if necessary. An error is generated in case it is not possible to convert value to a string.

virtual bool asBool() const = 0
: Returns the value of the attribute as a boolean, converting it if necessary. An error is generated in case it is not possible to convert value to a bool.

virtual const string & units() const = 0
: Get the units of the attribute's value. Only relevant for numeric types.

virtual void units(const string&) = 0
: Set the units of the attribute's value. Only relevant for numeric types.

virtual DataType dataType() const = 0
: Get the data type of the attribute's value.

virtual void dataType(DataType type) = 0
: Set the data type of the attribute's value.

virtual char* addressOfValue() const = 0
: Get the memory address of attribute's value.

virtual unsigned int dataSize() const = 0
: Get the size, in bytes, of the attribute's value.

virtual void printOn(ostream& os) const = 0
: Output an ASCII representation of the attribute to a given stream.

FUNCTION

ostream& operator¡¡(ostream& os, const Attribute& d)

PURPOSE

Output an ASCII representation of the attribute to a given stream.

CLASS

Attributable

PURPOSE

An object that is derived from Attributable has a set of attributes. An attribute is a dictionary of keyword-value pairs. Numeric attributes have a string describing the units; each attribute has a comment.

DERIVED FROM

public virtual Dal

public virtual Labelled

METHODS

virtual Attributable() {}

virtual Attributable& operator=(const Attributable&) = 0

virtual Attribute * addAttribute(const string & name, const string & comment = "", const string & units = "") = 0
: Create and add an attribute to the set. This does not yet define the data type of the attribute's value. This is done with one of the assignment operators in the Attribute class. An attribute with name name must not already exist in the set, otherwise an error is raised.

virtual Attribute * addAttribute(const Attribute * attribute) = 0
: Create and add an attribute to the set using the name, value, comment and units of the given attribute.

virtual void addAttributes(const Attributable*) = 0
: Add the attributes from the given Attributable to this attributable set.

virtual bool hasAttribute(const string & attributeName) const = 0
: Determines if an attribute with the specified name exists in the set. Returns true if an attribute of the specified name exists.

virtual Attribute * attribute(const string & attributeName) = 0
: Get the attribute with the given name. If it does not exist, an error is generated.

virtual const Attribute * attribute(const string & attributeName) const = 0
: Same as above except applies to constant objects.

virtual unsigned int attributes() const = 0
: Get the number of attributes in the set.

virtual Attribute * attribute(unsigned int number) = 0
: Returns the attribute with the the given number (ordinal position within the set). Can be used to iterate over all attributes in the set. number must be in the range [0,n-1] where n in the number of attributes in the set.

virtual const Attribute * attribute(unsigned int number) const = 0
: Same as above except applies to constant objects.

virtual void deleteAttribute(const string & name) = 0
: Deletes the attribute with the given name. If the attribute was not found an error is raised.

virtual void deleteAttribute(unsigned int number) = 0
: Deletes attribute with the given number (ordinal position within the set). If the attribute was not found an error is raised. number must be in the range [0,n-1] where n is the number of attributes in the set.

virtual void addComment(const string& comment) = 0
: Add a comment to the set. This may be any arbitrary string.

virtual unsigned int comments() const = 0
: Returns the number of comment lines in the set.

virtual const string & comment(unsigned int number) const = 0
: Returns comment line with the specified number starting from 0.

virtual void addHistory(const string& historyComment) = 0
: Add a history record to the set. This may be any arbitrary string.

virtual unsigned int historys() const = 0
: Returns the number of history records in the set.

virtual const string & history (unsigned int number) const = 0
: Returns the specified history record (starting from 0).

virtual void printOn(ostream& os) const = 0
: Output an ASCII representation of the attributable object to the given stream.

FUNCTION

ostream& operator¡¡(ostream& os, const Attributable& d)
Output an ASCII representation of the attributable object to the given stream.

CLASS

DataSet

PURPOSE

Structure classes. These objects do not necessarily contain the data itself, but they contain the information associated with the objects. For example, the column object can tell you its name, data type, number of rows etc, but to access the data the data in the column itself one of the data objects is needed. A DataSet is attributable, and contains a set of blocks, where a block is either a data table or an array.

DERIVED FROM

public virtual Attributable

public virtual Child¡DataSetServer¿

METHODS

virtual DataSet() {}

virtual Table * addTable(const string & name, unsigned long rows, const string & label = "", int position = -1) = 0
: Create and add a new table to the dataset. A pointer to the new table if returned. The arguments are:

- name The name of the table. It may, in priciple, be any arbitrary string, but should be limited to be FITS compliant. If a block with this name already exists in the dataset an error is raised.
- rows Specifies the number of rows of the table. This is used internally to ensure that all table columns have the same length.
- label A short description (typically one line) for the table.
- position The ordinal position, within the dataset, which the table is to occupy. Existing blocks will be moved along if necessary. The default value of -1 ensures that the table is placed at the end of the dataset.

virtual Array * addArray(const string& name, DataType dataType, const vector¡unsigned long¿ & size, const string & units = "", const string & label = "", int position = -1) = 0
: Creates and adds a new array to the dataset. A pointer to the new array is returned. The arguments are:

- name The name of the array. Can in priciple be any arbitrary string, but should be limited to be FITS compliant. If a block with this name already exists in the dataset an error is raised.
- dataType The data type of the array's data. It must be one of Int8, Int16, Int32, Real32, Real64. Note that Bool and DString types are not supported for arrays.
- size A vector whose elements describe the length along each dimension of the array's data. Note that the number of elements in this vector is the same as the number of dimensions of the array's data.
- units The units for the array's data.
- label A short description (typically one line) for the array.
- position The ordinal position, within the dataset, which the array is to occupy. Existing blocks will be moved along if necessary. The default value of -1 ensures that the array is placed at the end of the dataset.

virtual Block * add(const Block * block, const string& newName = "", int position = -1) = 0

Adds a copy of the given block to the dataset. The arguments are:

- block The block to be copied to the dataset. The name may be overwridden by the newName argument. The owner is not copied, and the owner of the copied block is this dataset.
- newName The name for the new block. The defualt value ensures that the given block's name is also copied.
- position The ordinal position, within the dataset, which the block is to occupy. Existing blocks will be moved along if necessary. The default value of -1 ensures that the block is placed at the end of the dataset.

virtual bool hasBlock(const string & blockName) const = 0

Determines if a block with the given name exists in the dataset. Returns true if a block of the specified name exists.

virtual Table * table(unsigned int blockNumber, AccessMode=AsParent) = 0

Get the table with the given number from the dataset. The arguments are:

- blockNumber The ordinal position of the table within the dataset. Must be in the range [0,n-1] where n is the number of blocks within the dataset.
- AccessMode The access mode which the table is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Table * table(unsigned int blockNumber, AccessMode=AsParent) const = 0

Same as above except applies to constant objects.

virtual Table * table(const string & blockName, AccessMode=AsParent) = 0

Get the table with the given name from the dataset. The arguments are:

- blockname The name of the table to be retrieved from the dataset. In the event that the table with name name is not found (usually because the block is eitehr an array or does not exist at all) an error is raised.
- AccessMode The access mode which the block is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Table * table(const string & blockName, AccessMode=AsParent) const = 0

Same as above except applies to constant objects.

virtual Array * array(unsigned int blockNumber, AccessMode=AsParent) = 0

- blockNumber The ordinal position of the array within the dataset. Must be in the range [0,n-1] where n is the number of blocks within the table.
- AccessMode The access mode which the array is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Array * array(unsigned int blockNumber, AccessMode=AsParent) const = 0
: Same as above except applies to constant objects.

virtual Array * array(const string & blockName, AccessMode=AsParent) = 0
: Get the array with the given name from the dataset. The arguments are:

- blockname The name of the array to be retrieved from the dataset. In the event that the array with name name is not found (usually because the block is either a table or does not exist at all) an error is raised.
- AccessMode The access mode which the block is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Array * array(const string & blockName, AccessMode=AsParent) const = 0
: Same as above except applies to constant objects.

virtual Block * block(unsigned int blockNumber, AccessMode=AsParent) = 0
: Get the block with the given name from the dataset. The arguments are:

- blockNumber The ordinal position of the block within the dataset. Must be in the range [0,n-1] where n is the number of blocks within the dataset.
- AccessMode The access mode which the block is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Block * block(unsigned int blockNumber, AccessMode=AsParent) const = 0
: Same as above except applies to constant objects.

virtual Block * block(const string & blockName, AccessMode=AsParent) = 0
: Get the block with the given name from the dataset. The arguments are:

- blockname The name of the block to be retrieved from the dataset. In the event that the block with name name is not found (usually block does not exist at all) an error is raised.
- AccessMode The access mode which the block is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Block * block(const string & blockName, AccessMode=AsParent) const = 0
: Same as above except applies to constant objects.

virtual void deleteBlock(unsigned blockNumber) = 0
: Delete the block with the given ordinal position from the dataset. In the event that the block was not found an error is raised.

virtual void deleteBlock(const string & blockName) = 0
: Delete the block with the given name from the dataset. In the event that the block was not found an error is raised.

virtual unsigned blockNumber(const string & name) const = 0
: Returns the number of the block with the given name. In the event that the block is not found an error is raised.

virtual unsigned blocks() const = 0
: Returns the number of blocks in the data set.

virtual AccessMode mode() const = 0
: Get the access mode of this dataset.

virtual void printOn(ostream& os) const = 0
: Output an ASCII representation of this dataset to the given stream.

FUNCTION

ostream& operator¡¡(ostream& os, const DataSet& d)
Output an ASCII representation of the given dataset to the given stream.

CLASS

Block

PURPOSE

A block is an abstract interface for all component of a DataSet.

DERIVED FROM

public virtual Attributable

public virtual Child¡DataSet¿

DATA

enum Type TableT = 1, ArrayT

These enumeration values are used to indicate the fundamental block types. The values are:

- TableT A Table.
- ArrayT An Array.

METHODS

virtual Block() {}

virtual Type type() const = 0

Returns ArrayT if the block is an Array, and returns TableT if the block is a table.

virtual void printOn(ostream&) const = 0

Output an ASCII repreentation of the block to the given putput stream.

TEMPLATE CLASS T

Seekable

PURPOSE

An object which is seekable contains data which may be accessed in a restricted (as a subrange) manner. Seekable provides the methods for setting subranges of data.

DERIVED FROM

None.

METHODS

virtual Seekable() {}

virtual void seek(T from, T count) = 0

Set a seek to the given range [from,from+count]. The arguments are:

- from The location of the start of the range.
- count The number of items to include in the range.

virtual T from() const = 0

Get the from value of the current range.

virtual T count() const = 0

Get the count value of the current range.

CLASS

Table

PURPOSE

A table is block which contains a set of columns. The columns in a table all have the same length, but may have different data types.

DERIVED FROM

public virtual Block

public virtual Seekable¡unsigned long¿

METHODS

virtual Table() {}

virtual Column * addColumn(const string& name, DataType dataType, const string& label = ””, const string& units = ””, const vector¡unsigned long¿ & size = scalar, int position = -1) = 0
Create and add a new column to the table. A pointer to the new column is returned. The length of column is set to the number of rows of the table. The arguments are:

- name The name of the column. It may, in priciple, be any arbitrary string, but should be limited to be FITS compliant. If a column with this name already exists in the dataset an error is raised.
- dataType The data type of the column's data. It may be any one of the enumeration values given in the Dal::dataType type.
- label A short description (typically one line) for the column.
- units The units for the column's data.
- size The dimensionality of the column's data.
- position The ordinal position, within the table, which the column is to occupy. Existing columns will be moved along if necessary. The default value of -1 ensures that the column is placed at the end of the table.

virtual Column * add(const Column * column, const string& newName = ””, int position = -1) = 0
Copy and add the given column to the table. A pointer to the new column is returned. The given column must have the same number of rows as the table, otherwise an error is raised. The arguments are:

- column The source column. The dataType, size, units, label, attributes and data will be copied to the new column.
- newName The name of the new column. The default value ensures that the name of the source column is used. If a column with this name already exists in the table an error will be raised.
- position The ordinal position, within the table, which the column is to occupy. Existing columns will be moved along if necessary. The default value of -1 ensures that the column is placed at the end of the table.

virtual Column * column(unsigned columnNumber, AccessMode=AsParent) = 0
Get the column with the given ordinal position from the table. A pointer to the required column is returned. The arguments are:

- columnNumber The ordinal position of the column within the table. Must be in the range [0,n-1] where n is the number of columns within the table.
- AccessMode The access mode which the column is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Column * column(unsigned columnNumber, AccessMode=AsParent) const = 0
Same as above except applies to constant objects.

virtual Column * column(const string & columnName, AccessMode=AsParent) = 0
Get the column with the given name from the table. A pointer to the required column is returned. The arguments are:

- columnName The name of the column to retrieve. An error is raised in the event that the column was not found.

- AccessMode The access mode which the column is to have. The default ensures that the access mode is the same as that of the parent object.

virtual const Column * column(const string & columnName, AccessMode=AsParent) const = 0
Same as above except applies to constant objects.

virtual bool hasColumn(const string & columnName) const = 0
Determines if the table has a column with the given name.

virtual unsigned int columnNumber(const string & columnName) const = 0
Get the ordinal position of the column with the given name. In the event that no such column exists an error is raised.

virtual void deleteColumn(unsigned columnNumber) = 0
Delete the column with the given ordinal position from the table. In the event that no such column exists an error will be raised.

virtual void deleteColumn(const string & columnName) = 0
Delete the column with the given name from the table. In the event that no such column exists an error will be raised.

virtual unsigned long rows() const = 0
Get the number of rows in the table.

virtual unsigned columns() const = 0
Get the number of columns in the table.

virtual void copyRows(unsigned long from, unsigned long to, unsigned long count=1) = 0
Copy the specified range of rows.

virtual void deleteRows(unsigned long from, unsigned long count=1) = 0
Delete the specified range of rows from the table.

virtual void insertRows(unsigned long pos, unsigned long count=1) = 0
Insert the given number of rows into the table.

virtual void printOn(ostream& os) const = 0
Writes an ascii representation of the column to a stream. Output an ASCII representation of the table to the given stream.

virtual void forEachSubTable(void (*callThisFunction)(Table *)) = 0
Call the given function for each subtable.

CLASS Nullable

PURPOSE
Nullable allows the values in a data component to have a designated null (or undefined) value.

DERIVED FROM
None

METHODS

virtual Nullable() {}

enum NullType Integer = 1, Real, String, Undefined
Used to determine the null value type of an object.

virtual NullType nullType() const = 0
Get the null value type of an object.

virtual void nullValue(long value) = 0
Set the integer null value.

virtual long intNullValue() const = 0
Get the integer null value.

virtual bool nullDefined() const = 0
: Determine if the null value has been set.

virtual void deleteNullValue() = 0
: Delete the null value. An error is raised if null value is not defined. The nullDefined() method can be used to determine if the null value is defined. For integer-valued columns, the nullValue(int) method can be used to set the null value. For real-valued columns, the null value is always defined.

CLASS DataComponent

PURPOSE

A DataComponent is a collection of values all of the same type, arranged in a multidimensional array. The collection is referred to as teh object's data or simply the data.

DERIVED FROM

public virtual Nullable, public virtual Dal

METHODS

virtual DataComponent() {}

virtual DataType dataType() const = 0
: Get the data type of the data.

virtual unsigned int dimensions() const = 0
: Get the number of dimensions of the object's data.

virtual unsigned long elements() const = 0; Get the total number of elements comprising the object's data.

virtual const vector¡unsigned long¿& size() const = 0; Get the dimensionality of the object's data. Each element in the returned vector describes the size along each axis (dimension) of the object's data.

virtual const string & units() const = 0
: Get the units of the object's data.

virtual void units(const string&) = 0
: Set the units of the object's data.

virtual unsigned int dataSize() const = 0
: The size in bytes of a single value.

virtual void scaling(double zero, double scale) = 0
: Set the scaling of the object's data.

virtual void scale(bool onoff) = 0

virtual bool scaled() const = 0

virtual double scaleZero() const = 0

virtual double scaleFactor() const = 0

CLASS

Column

PURPOSE

A column resides within its parent table. The parent table can be obtained with the parent() method. Internally, the Column object is responsible for the allocation and deallocation of its data's memory and intialization of its data, but it does not allow its data to be accessed directly. The data is accessed through the data descriptor objects ColumnData and CellData.

It is possible to have several ColumnData and CellData descriptors at the same time. However, the [from,range] range specifications (in the data() and cellData() and seek() methods) give rise to a slice (or subrabge) of the Column's data. The only restriction on slices is that they must not overlap with existing slices (although a subslice is an existing slive is allowed).

Moreover, but they all have to be deleted manually to avoid memory leaks. In particular, the following example is eroneous as it leads to a memory leak, since the pointer to the ColumnData object (as returned by the data() method) is lost:

```
int
main()
{
  DataSet * set = dataSetServer -> open( "test.dat", DataSetServer::Create );
  Table * tab = set -> addTable( "tab1", 100 );
  Column * col = tab -> addColumn( "col1", Column::Int32 );
  int32 * data = col -> data() -> int32Data();   // Memory leak
  for( unsigned int i = 0; i < col -> elements() * col -> rows(); ++i ) data[i] = i;
  dataSetServer -> close( set );
}
```

The correct method is as follows:

```
int
main()
{
  DataSet * set = dataSetServer -> open( "test.dat", DataSetServer::Create );
  Table * tab = set -> addTable( "tab1", 100 );
  Column * col = tab -> addColumn( "col1", Column::Int32 );
  ColumnData * coldat = col -> data();
  int32 * data = int32Data();
  for( unsigned int i = 0; i < col -> elements(); ++i ) data[i] = i;
  delete coldat; // Need to manually delete columnData objects to avoid memory leak
  dataSetServer -> close( set );
}
```

The same is also true of the CellData object; it must be deleted after its final use, otherwise a memory leak is incurred.

DERIVED FROM

public virtual Attributable

public virtual DataComponent

public virtual Child¡Table¿

public virtual Seekable¡unsigned long¿

DATA

enum CellType { Fixed = 1, Variable }

The values have the following meaning:

- Fixed Specifies that the column has fixed length.

- Variable Specifies that the column has variable length.

METHODS

virtual Column() {}

virtual unsigned int columnNumber() const = 0
    Get the ordinal position of the column within the parent table.

virtual CellType cellType() const = 0

virtual unsigned long rows() const = 0

virtual ColumnData * data(unsigned long from=0, unsigned long count=0, AccessMode accessMode=AsParent) const = 0
    Get a data descriptor describing a range of column cells to be accessed. The range is specified as [from,from+count]. A pointer to the ColumnData object descriptor is returned, which must be deleted manually when it is no longer needed.

    Note that once the ColumnData object has been deleted, any corresponding pointers to the Column's data will be stale and can no longer be safely used.

    The arguments are as follows:

    - from The first row number in the range to be accessed.
    - count The number of rows to include in the range.
    - accessMode The access mode with which the data is accessed.

virtual CellData * cellData(unsigned long rowNumber, unsigned long from=0, unsigned long count=0, AccessMode accessMode=AsParent) const = 0
    Get a data descriptor describing a range of elements within a column cell to be accessed. The range is specified as [from,from+count]. A pointer to the ColumnData object descriptor is returned, which must be deleted manually when it is no longer needed.

    Note that once the CellData object has been deleted, any corresponding pointers to the Column's data will be stale and can no longer be safely used.

    The arguments are as follows:

    - rowNumber The number of the cell to be accessed.
    - from The element of the first element to be included in the range.
    - count The number of elements to include in the range.
    - accessMode The access mode with which the data is accessed.

virtual void printOn(ostream& os) const = 0
    Outputs an ASCII representation of the column to a stream.

FUNCTION

ostream& operator¡¡(ostream& os, const Column& c)
Outputs an ASCII representation of the given column to a stream.

CLASS

Array

PURPOSE

An array is a Block that consists of an n-dimensional array of values all of the same type. An array resides within its parent dataset. The parent dataset can be obtained with the parent() method. Internally, the Array object is responsible for the allocation, deallocation and initialisation of its data's memory, but it does not allow its data to be accessed directly. The data is accessed through the data descriptor object ArrayData.

It is possible to have several ArrayData descriptors at the same time. However, the [from,range] range specifications (in the data() and seek() methods) give rise to a slice (or subrabge) of the Column's data. The only restriction on slices is that they must not overlap with existing slices (although a subslice of an existing slice is allowed).

Moreover, they all have to be deleted manually after their last use to avoid memory leaks.

In particular, the following is considered eroneous as it leads to a memory leak, since the pointer to the ArrayData object (as returned by the data() method) is lost:

```
int
main()
{
  DataSet * set = dataSetServer -> open( "test.dat", DataSetServer::Create );
  Array * arr = set -> addArray( "arr1", size );
  int32 * data = arr -> data() -> int32Data();   // Memory leak
  for( unsigned int i = 0; i < arr -> elements(); ++i ) data[i] = i;
  dataSetServer -> close( set );
}
```

The correct method is as follows:

```
int
main()
{
  DataSet * set = dataSetServer -> open( "test.dat", DataSetServer::Create );
  Array * arr = set -> addArray( "arr1", size );
  ArrayData * arrdat = arr -> data();
  int32 * data = arrdat() -> int32Data();
  for( unsigned int i = 0; i < arr -> elements(); ++i ) data[i] = i;
  delete coldat; // Need to manually delete columnData objects to avoid memory leak.
  dataSetServer -> close( set );
}
```

DERIVED FROM

public virtual Block

public virtual DataComponent

public virtual Seekable¡ vector¡unsigned long¿ ¿

METHODS

virtual Array() {}

virtual ArrayData * data(const vector¡unsigned long¿& from=zero, const vector¡unsigned long¿& count=zero, AccessMode accessMode=AsParent) const = 0

Get a data descriptor describing a range of elements within the array's data to be accessed. The range is specified as [from,from+count]. A pointer to the ArrayData object descriptor is returned, which must be deleted manually when it is no longer needed. The arguments are as follows:

- from The element of the first element to be included in the range.
- count The number of elements to include in the range.
- accessMode The access mode with which the data is accessed.

CLASS

Data

PURPOSE

Data access. The data access functions int8Data(), uint16Data(), int16Data(), uint32Data(), int32Data(), real32Data(), real64Data(), boolData() and stringData() generate an error when the data cannot be accessed as a contiguous chunck of memory, such as a variable-size column. The data is typed and no type conversion is possible. The data type is determined with dataType(). An error will be raised if the incorrect data access function is called.

DERIVED FROM

public virtual Dal

METHODS

virtual Data(){}

virtual bool isNull(unsigned long pos) const = 0
: Determine if the element in position pos is a null value.

virtual void setToNull(unsigned long pos) const = 0
: Set the element at position pos to null.

virtual bool hasNulls() const = 0
: Determine if object has any null values.

virtual int8* int8Data() const = 0
: Return a pointer to the start of the data.

virtual uint16* uint16Data() const = 0
: Return a pointer to the start of the data.

virtual int16* int16Data() const = 0
: Return a pointer to the start of the data.

virtual uint32* uint32Data() const = 0
: Return a pointer to the start of the data.

virtual int32* int32Data() const = 0
: Return a pointer to the start of the data.

virtual real32* real32Data() const = 0
: Return a pointer to the start of the data.

virtual real64* real64Data() const = 0
: Return a pointer to the start of the data.

virtual bool8* boolData() const = 0
: Return a pointer to the start of the data.

virtual char* stringData() const = 0
: Return a pointer to the start of the data.

virtual unsigned int dataSize() const = 0

virtual unsigned long elements() const = 0
: Get the number of data elements.

virtual DataType dataType() const = 0
: Get the size in types of a single data element.

virtual void printOn(ostream& os, const string& sep=””) const = 0
: Outputs an ASCII representation of the data to a stream.

FUNCTION

ostream& operator¡¡(ostream& os, const Data& d)
Outputs an ASCII representation of the given data to a stream.

CLASS

ColumnData

PURPOSE

The ColumnData object gives access to (part of) the data in a column. The correct access function must be called (this is dependent on the data type) otherwise an error will be raised. This object is constructed by a Column, but has to be explicitly deleted after it's last use. The owner of a ColumnData object is a Column object, which is determined with the parent() method.

DERIVED FROM

public virtual Data

public virtual Seekable¡unsigned long¿

public virtual Child¡Column¿

METHODS

virtual ColumnData() {}

virtual const vector¡unsigned long¿& size() const = 0

virtual unsigned long position(unsigned long row) const = 0
: Get the offset index of the first element in the specified row of the (parent) column.

virtual unsigned long position(unsigned long row, unsigned long pos) const = 0
: Get the offset index of the element in position pos in the specified row of the (parent) column.

virtual unsigned long position(unsigned long row, const vector¡unsigned long¿& pos) const = 0
: Get the offset index of the element in the position described by pos in the specified row in the (parent) column.

CLASS

MatrixData

PURPOSE

MatrixData provides an interface to a rectangular multi-dimensional array.

DERIVED FROM

public virtual Data

public virtual Seekable¡ vector¡unsigned long¿ ¿

METHODS

virtual MatrixData() {}

virtual unsigned long position(const vector¡unsigned long¿& pos) const = Get the offset index of the element in the position described by pos in the (parent) array. 0

CLASS

CellData

PURPOSE

The CellData object gives access to (part of) a single cell in a column. The correct access function must be called (this is dependent on the data type) otherwise an error will be raised. This object is constructed by a Column, but has to be explicitly deleted after it's last use. The CellData object is owned by the parent column object, which is determined with the parent() method. For Variable length columns, the number of dimensions is 1.

DERIVED FROM

public virtual Data

public virtual Seekable¡unsigned long¿

public virtual Child¡Column¿

METHODS

virtual CellData() {}

virtual unsigned long size() const = 0
    Get the number of elements in the cell.

virtual void size(unsigned long size) = 0
    Set the number of elements in the cell.

virtual unsigned long row() const = 0
    Get the cell (row) number of the cell.

CLASS

ArrayData

PURPOSE

The ArrayData object gives access to (part of) the data in an Array. The correct access function must be called (this is dependent on the data type) otherwise an error will be raised. Note that stringData() and boolData() never called since DString and Bool are not supported by Arrays. This object is constructed by an Array, but has to be explicitly deleted after it's last use. The ArrayData object is owned by the parent Array object, which can be determined with the parent() method.

DERIVED FROM

public virtual MatrixData

public virtual Child¡Array¿

METHODS

virtual const vector¡unsigned long¿& size() const = 0
    Get the dimensionality of the array.

21 C API

C interface definition for DAL

Pointers rather than handles. No default values.

typedef enum Read = 1, Create, Modify, Temp, AsParent AccessMode; typedef enum High = 1, HighLow, Low, UseEnvironment MemoryModel; /*typedef enum Bool = 1, Int8, Uint16, Int16, Uint32, Int32, Real32, Real64, DString DataType; */ typedef enum TableType = 1, ArrayType BlockType; typedef enum EraseAllFirst = 1, Merge CopyMode; typedef enum Fixed = 1, Variable CellType;

typedef void Array; typedef void Block; typedef void Column; typedef void DataSet; typedef void Table; typedef void SubTable; typedef void Row; typedef void Attribute; typedef void Attributable; typedef void Labelled; typedef void DataComponent; typedef void * TableIteratorFunction; /* typedef void(*TableIteratorFunction)(Table *); */

NAME

addArray(dataSet, name, dataType, numberofDimensions, dimensions, units, label, position);

PURPOSE

Create and add an array to a dataset.

ARGUMENTS

- DataSet * dataSet
  A pointer to the dataset which the new array is to be added.
- const char * name
  The name of the new array.
- DataType dataType
  The type of the data. It must be one of the values: Int8, Int16, Int32, Real32, Real64.
- int numberOfDimensions
  The number of dimensions of the array. This must be in the range 1 ¡= numberOfDimensions ¡= 3.
- unsigned long * dimensions
  A vector with numberOfDimensions elements. Each element describes the size along each dimension, of the array, respectively.
- const char * units
  The units of the array.
- const char * label
  A short description (i.e. a user defined comment) to be attached to the array.
- int position
  The ordinal position, within the dataset, which the new array will occupy.

RETURNS

- Array *
  A pointer to the newly created array.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

String and Boolean types are not supported.

NAME

addColumn(table, name, dataType, units, label, numberofDimensions, dimensions, position)

PURPOSE

Create and add a column to a table.

ARGUMENTS

- Table * table
  A pointer to the table to which the new column will be added.
- const char * name
  The name of the new column.
- DataType dataType
  The data type of the new column. It can be any of the values in the enumeration type DataType.
- const char * units
  The units of the column.
- const char * label
  A short description (i.e. a user-defined comment) of the column.
- int numberofDimensions
  The number of dimensions of the column.
- const unsigned long * dimensions
  The size along each dimension of the column.
- int position
  The ordinal position wihtin the table which the new column will occupy.

RETURNS

- Column *
  A pointer to the new column.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

addCommentTo¡type¿(object, comment)

PURPOSE

Add a comment record to an object.

ARGUMENTS

- type * object
  A pointer to the object to which the comment is to be added. The supported types are: Array, Attributable, Block, DataSet, Table
- const char * comment
  The comment which is to be to be added to the object.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
addHistoryTo¡type¿(object, history)

PURPOSE
Add a history record to an object.

ARGUMENTS

- type * object
  A pointer to the object to which the history record is to be added. The supported types are: Array, Attributable, Block, DataSet, Table
- char * history
  The history record which is to be to be added to the object.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

addTable(dataSet, name, numberOfRows, label, position)

PURPOSE

Create and add a table to a dataset.

ARGUMENTS

- DataSet * dataSet
  A handle of the dataset to which the new table is to be added.
- char * name
  The name of the new table.
- int numberOfRows
  The nmber of rows of the new table.
- const char * label
  A short textual description (i.e. user-defined comment) to be attached to the table.
- int position
  The ordinal position of the new table within the dataset.

RETURNS

- Table *
  A pointer to the newly created table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿AttributeComment(object, name)

PURPOSE

Get the comment associated with an attribute, from an attributable object.

ARGUMENTS

- const type * object
  The object containing the attribute. Supported types are: Array, Attributable, Block, Column, DataSet, Table.

- const char * name
  The name of the attribute.

RETURNS

- const char *
  A pointer to the comment.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿AttributeUnits(object, name)

PURPOSE

Get the comment associated with an attribute, from an attributable object.

ARGUMENTS

- const type * object
  The object containing the attribute. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- const char * name
  The name of the attribute.

RETURNS

- const char *
  A pointer to the units.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿AttributeWithName(object, name)

PURPOSE

Get an attribute from an attributable object.

ARGUMENTS

- const type * object
  A pointer to the object which contains the required attribute. Supported types are: Array, Attributable, Block, Column, DataSet, Table
- const char * name
  The name of the required attribute.

RETURNS

- Attribute *
  A pointer to the required attribute.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

arrayDataType(array)

PURPOSE

Get the type of an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- DataType
  The type of the array. It will be one of the values: Int8, Int16, Int32, Real32, Real64.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

bool8 ¡type¿HasAttribute(attributable, name)

PURPOSE

Determine if an attributable object contains an attribute with a given name.

ARGUMENTS

- const ¡type¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- const char * name
  The name of an attribute.

RETURNS

- bool8
  Returns true if an attribute with the given name was found, otherwise falise is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿Label(object)

PURPOSE

Get the label associated with an object.

ARGUMENTS

- const ¡type¿ * object
  A pointer to an object. Supported types are: Array, Block, Column.

RETURNS

- const char *
  A pointer to the array's label.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿ToAttributable(attributable)

PURPOSE

Convert an attributable object to the Attributable type.

ARGUMENTS

- ¡type¿ * attributable
  A pointer to the attributable object to be converted. Supported types are: Array, Block, Column, DataSet, Table.

RETURNS

- Attributable * A pointer to Attributable.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

arrayUnits(array)

PURPOSE

Get the units associated with an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- const char *
  A pointer to the array's units.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

arrayWithName(dataSet, name)

PURPOSE

Get the array with a given name from a dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the data set which contains the required array.
- const char * name
  The name of the required array.

RETURNS

- Array *
  A pointer to the array.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

*Array * arrayWithNumber(dataSet, position)*

PURPOSE

Get the array with a given ordinal position from a dataset.

ARGUMENTS

- const DataSet * dataSet,
  A pointer to the data set which contains the required array.
- unsigned int position
  The ordinal position of the required array.

RETURNS

- Array *
  A pointer to the array.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

attributeDataType(attribute)

PURPOSE

Get the type of an attribute.

ARGUMENTS

- Attribute * attribute
  A pointer to the attribute.

RETURNS

- DataType
  The data type of the attribute.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

attributeLabel(attribute)

PURPOSE

Get the label associated with an attribute.

ARGUMENTS

- const Attribute * attribute
  A pointer to the attribute.

RETURNS

- const char *
  A pointer to the attribute's label.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

attributeUnits(attribute)

PURPOSE

Get the units associated with an attribute.

ARGUMENTS

- const Attribute * attribute
  A pointer to the attribute.

RETURNS

- const char *
  Get the units associated with an attribute.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

blockNumber(dataSet, name)

PURPOSE

Get the number of a block (ordinal position within it's dataset) with a given name.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset containing the block with the given name.
- const char * name
  The name of the block.

RETURNS

- unsigned int
  The ordinal position of the blockj within the dataset.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

blockType(block)

PURPOSE

Get the type of a block.

ARGUMENTS

- const Block * block
  A pointer to the block.

RETURNS

- BlockType
  The block type.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

blockWithName(dataSet, name)

PURPOSE

Get a block with a given name from a dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset containing the required block.
- const char * name
  The name of the required block.

RETURNS

- Block *
  A pointer to the block.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

blockWithNumber(dataSet, position)

PURPOSE

Get a block with a given number (ordinal position) from a dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset containing the required block.
- int position
  The ordinal position of the block within the dataset.

RETURNS

- Block *
  A pointer to the block.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿¡attributable¿Attribute(object, name);

PURPOSE

Get the value of an attribute contained in an attributable object.

ARGUMENTS

- const ¡attributable¿ * object
  A pointer to the attributable object containing the required attribute. Supported types are: Array, Attributable, Block, Column, DataSet, and Table.
- const char * name
  The name of the required attribute.

RETURNS

- ¡type¿
  Supported types are: bool, int8, int16, int32, real32, real64, string

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿ArrayData(array)

PURPOSE

Get the data from an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- ¡type¿ *
  A pointer to the data of the appropriate type.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿Attribute(attribute)

PURPOSE

Get an attribute's data.

ARGUMENTS

- const Attribute * attribute
  A pointer to the attribute.

RETURNS

- ¡type¿
  A value of the appropriate type. Supported types are Bool, Inter, Real, String.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿CellData(column, rowNumber)

PURPOSE

Get the data from a cell in a variable length column.

ARGUMENTS

- const Column * column
  A pointer to the variable length column.
- unsigned long rowNumber
  The number of the column cell to be accessed.

RETURNS

- ¡type¿ * A pointer, of the appropriate type, to the data. Supported types are: Bool, Int8, Int16, Int32, Real32, Real64, String.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

¡type¿ColumnData(column)

PURPOSE

Get the data from a fixed length column.

ARGUMENTS

- const Column * column
  A pointer to the fixed length column.

RETURNS

- ¡type¿ * A pointer, of the appropriate type, to the column's data. Supported types are: Bool, Int8, Int16, Int32, Ral32, Real64, String.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

int cellSize(column, rowNumber)

PURPOSE

Get the size of a cell in a variable-length column.

ARGUMENTS

- Column * column
  A pointer to the variable-length column.
- int rowNumber
  The cell number.

RETURNS

- int
  The size of the cell in bytes.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

cellType(column)

PURPOSE

Get the cell-type of a column.

ARGUMENTS

- Column * column
  A pointer to the cell.

RETURNS

- CellType

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

clobber()

PURPOSE

Get the clobber setting.

ARGUMENTS

RETURNS

- bool8

DESCRIPTION
The clobber setting is determined by the environment variable SAS_CLOBBER.

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
unsigned long columnDataIndex(column, rowNumber, numberOfDimensions, dimensions)

PURPOSE
Get the memory offset of a column's row.

ARGUMENTS

- const Column * column
- unsigned long rowNumber
- unsigned int numberOfDimensions
- const unsigned long * dimensions

RETURNS

- unsigned long
  The omemory offset.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

columnDataType(column)

PURPOSE

Get the data type of a column.

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- DataType

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

columnNumber(table, name)

PURPOSE

Get the number of a column with a given name (i.e. the ordinal position of the column with it's table).

ARGUMENTS

- const Table * table
  A pointer to the table containing the required column.
- const char * name
  The name of the required column.

RETURNS

- int The ordinal position of the column within the given table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

columnUnits(column)

PURPOSE

Get the units associated with a column.

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- const char *
  A pointer to the column's units.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

columnWithName(table, name)

PURPOSE

Get a column with the given name from a table.

ARGUMENTS

- const Table * table
  A pointer to the table containing the required column.

- const char * name
  The name of the required column.

RETURNS

- Column *
  A pointer to the column.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

columnWithNumber(table, position)

PURPOSE

Get the column with the given ordinal position from a table.

ARGUMENTS

- const Table * table
  A pointer to the table containing the required column.
- unsigned int position
  The ordinal position of the required column.

RETURNS

- Column *
  A pointer to the column.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

*const char * commentOf¡object¿(attributable, number)*

PURPOSE

Get a comment record from an attributable object.

ARGUMENTS

- ¡object¿ attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet and Table.
- unsigned int number
  The ordinal number of the comment record to be retrieved.

RETURNS

- const char *
  A pointer to the comment record.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

copyAttributesOf¡object¿(to, from, copyMode)

PURPOSE

ARGUMENTS

- ¡object¿ * to
  A pointer to the destination object.
- const ¡object¿ * from
  A pointer to the source object.
- CopyMode copyMode
  The mode to be used for the copy.

RETURNS

void

DESCRIPTION

Supported types are: Array, Block, Column, DataSet, Table

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

The source and destination must be of the same attributable type.

NAME

copyAttributeTo¡object¿(attributable, attribute)

PURPOSE

Copy an attribute to an attributable object.

ARGUMENTS

- ¡object¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- const Attribute * attribute

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

copyRows(table, from, to, count)

PURPOSE

Copy rows in a table.

ARGUMENTS

- Table * table
  A pointer to the table within which the rows are to be copied.
- unsigned int from
  The row number (starting at 0) from which to begin the copying.

- unsigned int to
  The row number (starting at 0) to which to begin the copying.
- unsigned int count
  The number of rows to copy.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
dataServerClient(clientName)

PURPOSE
Set the name of the dataset server client.

ARGUMENTS

- const char * clientName

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

*DataSet * dataSet(dataSetName, openMode, memoryModel)*

PURPOSE

Open a dataset.

ARGUMENTS

- const char * dataSetName
- AccessMode openMode
- MemoryModel memoryModel

RETURNS

- DataSet *
  A pointer to the dataset.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

dataSetHasBlock(dataSet, name)

PURPOSE

Determine if a dataset has a block with the given name.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset.
- const char * name
  The name of the desired block.

RETURNS

- bool8
  Returns true if a block with the given name was found in the specified dataset, otherwise false is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

delete¡type¿AttributeWithName(attributable, name)

PURPOSE

Delete the named attribute from an attributable object.

ARGUMENTS

- ¡type¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- const char * name
  The name of the attribute to delete.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

delete¡type¿AttributeWithNumber(attributable, number)

PURPOSE

Delete the attribute with the given ordinal position from an attributable object.

ARGUMENTS

- ¡type¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- unsigned int number
  The ordinal position of the attribute to delete.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
deleteAttribute(attribute)

PURPOSE
Delete the given attribute.

ARGUMENTS

- const Attribute * attribute
  A pointer to the attribute to be deleted.

RETURNS
void

DESCRIPTION
The attribute is deleted from it's parent attributable.

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

deleteBlockWithName(dataSet, name)

PURPOSE

Delete the block with the given name from the specified dataset.

ARGUMENTS

- DataSet * dataSet
  A pointer to the dataset containing the block to be deleted.
- const char * name
  The name of the block to be deleted.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

deleteBlockWithNumber(dataSet, position)

PURPOSE

Delete the block with the specified ordinal position from the given dataset.

ARGUMENTS

- DataSet * dataSet
- unsigned int position

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

deleteColumnWithName(table, name)

PURPOSE

Delete the column with the specified name from the given table.

ARGUMENTS

- Table * table
  A pointer to the table containing the column to be deleted.
- const char * name
  The name of the column to delete.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

deleteColumnWithNumber(table, position)

PURPOSE

Delete the column with the specified ordinal position from the given table.

ARGUMENTS

- Table * table
  A pointer to the table containing the column to be deleted.
- unsigned int position
  The ordinal position of the column to be deleted.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

deleteRows(table, from, count)

PURPOSE

Delete a range of rows from a table.

ARGUMENTS

- Table * table
  A pointer to the table from which the rows are to be deleted.
- unsigned int from
  The row number (starting at 0) from which to begin the deleting.
- unsigned int count
  The number of rows to delete.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

dimensionsOfArray(array)

PURPOSE

Get the dimensions of an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- unsigned long *
  The dimensions are returned in a vector, each element of which describes the size along each axis.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

dimensionsOfArray(column)

PURPOSE

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- unsigned long *
  The dimensions are returned in a vector, each element of which describes the size along each axis.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

discardDataSet(dataSetName)

PURPOSE

Tell the dataset server object to discard the named dataset.

ARGUMENTS

- const char * dataSetName
  The name of the dataset.

RETURNS

void

DESCRIPTION

This function must only be called by Meta Tasks.

ERRORS

EXAMPLES

SEE ALSO

keepDataSet

BUGS AND LIMITATIONS

None known.

NAME

forEachSubTable(table, callThisFunction)

PURPOSE

Subtable iteration.

ARGUMENTS

- const Table * table
  A pointer to the table for which subtable iteration is required.
- TableIteratorFunction callThisFunction
  The function to be called for each subtable iteration.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

hasScalingOfArray(array)

PURPOSE

Determine if an array has been scaled.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- bool8
  Returns true if the array has been scaled, otherwise false is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

bool8 hasScalingOfColumn(column)

PURPOSE

Determine if a column has been scaled.

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- bool8
  Returns true if the column has been scaled, otherwise false is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

historyOf¡type¿(attributable, number)

PURPOSE

Get a history record from an attributable object.

ARGUMENTS

- ¡type¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- unsigned int number
  The ordinal number of the history record to be retrieved.

RETURNS

- const char *
  A pointer to the history record.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

insertRows(table, pos, count)

PURPOSE

Insert rows in a table.

ARGUMENTS

- Table * table
  A pointer to the table within which the rows are to be inserted.

- unsigned int from
  The row number (starting at 0) from which to begin the insertion.
- unsigned int count
  The number of rows to insert.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
keepDataSet(dataSetName)

PURPOSE
Tell the dataset server object not to discard the named dataset.

ARGUMENTS

- const char * dataSetName
  The name of the dataset.

RETURNS
void

DESCRIPTION
This function must only be called by Meta Tasks.

ERRORS

EXAMPLES

SEE ALSO
discardDataSet

BUGS AND LIMITATIONS

None known.

NAME

mode(dataSet)

PURPOSE

Get the access mode of a dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset.

RETURNS

- AccessMode
  The access mode with which the dataset was opened.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

nameOf¡object¿(labelled)

PURPOSE

Get the named of a labelled object.

ARGUMENTS

- const ¡object¿ * attributable
  A pointer to the labelled object. Supported types are: Array, Attributable, Column, Attribute, DataSet, Table, Block.

RETURNS

- const char *
  A pointer to the name of the object.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfBlocks(dataSet)

PURPOSE

Get the number of blocks in a dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to the dataset.

RETURNS

- unsigned int
  The number of blocks in the dataset.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfColumns(table)

PURPOSE

Get the number of columns in a table.

ARGUMENTS

- const Table * table
  A pointer to the table.

RETURNS

- unsigned int
  The number of columns in the table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfCommentsOf¡object¿(attributable)

PURPOSE

Get the number of comments in an attributable object.

ARGUMENTS

- ¡object¿ * attributable
  A pointer to the attributable object. Supported types are: Attributable, Array, Block, Column, DataSet, Table

RETURNS

- unsigned int
  The number of comments in the attributable object.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfDimensionsOfArray(array)

PURPOSE

Get the number of dimensions of an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- unsigned int
  The number of dimensions of the array.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfDimensionsOfColumn(column)

PURPOSE

Get the number of dimensions of a column.

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- unsigned int
  The number of dimensions of the column.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfElementsOfArray(array)

PURPOSE

Get the total number of elements in an array.

ARGUMENTS

- const Array * array
  A pointer to the array.

RETURNS

- unsigned long
  The total number of elements in the array.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfElementsOfColumn(column)

PURPOSE

Get the total number of elements in a column's (fixed length column only) cell.

ARGUMENTS

- const Column * column
  A pointer to the column.

RETURNS

- unsigned long
  The total number of elements in the column's cells.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfHistorysOf¡object¿(attributable)

PURPOSE

Get the number of history records in an attributable object.

ARGUMENTS

- ¡object¿ * attributable
  A pointer to the attributable object. Supported types are: Attributable, Array, Block, Column, Array, DataSet, Table.

RETURNS

- unsigned int
  The number of history records in the attributable object.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfRowsOfTable(table)

PURPOSE

Get the number of rows in a table.

ARGUMENTS

- const Table * table
  A pointer to the table.

RETURNS

- unsigned long
  The number of rows in the table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

numberOfRowsOfColumn(column)

PURPOSE

Get the number of rows in a column.

ARGUMENTS

- const Column * column

RETURNS

- unsigned long
  The number of rows in the column.

DESCRIPTION

Same as the number of rows in the column's (parent) table.

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

relabel¡object¿(labelled, newLabel)

PURPOSE

Relabel an labelled object.

ARGUMENTS

- ¡object¿ * labelled
  A pointer to the labelled object. Supported types are: Attributable, Attribute, Array, Block, Column, DataSet, Table.
- const char * newLabel
  The new label.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

void release¡object¿(releasable)

PURPOSE

Release an object.

ARGUMENTS

- ¡object¿ * releasable
  A pointer to the object to be released. Supported types are: Array, Block, Column, DataSet, Table

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

rename¡object¿(labelled, newName)

PURPOSE

Rename a labelled object.

ARGUMENTS

- ¡object¿ * labelled
  A pointer to the labelled object. Supported types are: Attribute, Array, Attributable, Block, Column, DataSet, Table.
- const char * newName
  The new name for the object.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

set¡type¿¡object¿Attribute(attributable, name, value, comment);

PURPOSE

Set the value of an attribute.

ARGUMENTS

- ¡object¿ * attributable
  A pointer to the attributable object. Supported types are: Array, Attributable, Block, Column, DataSet, Table.
- const char * name
  The name of the attribute.
- ¡type¿ value
  The value of the attribute. Supported types are bool8, Int8, Int16, Int32, Real32, Real64, String.
- const char * comment
  The comment to associate with the attribute.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

set¡type¿Attribute(attribute, value, comment)

PURPOSE

ARGUMENTS

- Attribute * attribute
  A pointer to the attribute.
- ¡type¿ value
  The value to assign to the attribute. Supported types are: Bool8, Int8, Int16, Int32, Real32, Real64, String.
- const char * comment
  The comment to associate with the attribute.

RETURNS

void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

setCellSize(column, rowNumber, size)

PURPOSE

Set the size of a cell in a variable-length column.

ARGUMENTS

- Column * column
  A pointer to the column.
- int rowNumber
  The number of the cell to be sized.

- int size
  The size to set the cell-size to.

RETURNS
void

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
setExists(setName)

PURPOSE
Determine if a set exists.

ARGUMENTS

- const char * setName
  The name of the set.

RETURNS

- bool
  Returns true if the set exists, otherwise false is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME
tableHasColumn(table, name)

PURPOSE

Determine if a table contains a column with the given name.

ARGUMENTS

- const Table * table
  A pointer to the table.
- const char * name
  The name of the desired column.

RETURNS

- bool8
  Returns true if the column with the given name was found in the table, otherwise false is returned.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

tableWithName(dataSet, name)

PURPOSE

Get the table with the specified name from a given dataSet.

ARGUMENTS

- const DataSet * dataSet
  A pointer to a dataset.
- const char * name
  The name of the desired table.

RETURNS

- Table *
  A pointer to the table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

NAME

tableWithNumber(dataSet, position)

PURPOSE

Get the table with the specified ordinal position in a given dataset.

ARGUMENTS

- const DataSet * dataSet
  A pointer to a dataset.
- unsigned int position
  The ordinal position of the table within the dataset.

RETURNS

- Table *
  A pointer to the table.

DESCRIPTION

ERRORS

EXAMPLES

SEE ALSO

BUGS AND LIMITATIONS

None known.

/* forEachBlock forEachColumn forEachSubTable setStringCell subTable table */

22 PERL API

See PEDAL documentation

References