NOTE 260 – CTDS File Formats

Ger van Diepen, ASTRON Dwingeloo

November 10, 2017

Abstract

The Casacore Table Data System (CTDS) uses several files to store data in. Each storage manager has its own files. The document describes the formats of all files that can be used by CTDS.

Contents

1 Introduction **3**

2 AipsIO **4**
- 2.1 Array object . . . 5
- 2.2 IPosition object . . . 6
- 2.3 Block and std::vector object . . . 6
- 2.4 std::map object . . . 6
- 2.5 TableRecord object . . . 6

3 PlainTable **8**
- 3.1 General file table.dat . . . 8
  - 3.1.1 Table Description . . . 8
  - 3.1.2 ColumnSet . . . 11
- 3.2 Info file table.info . . . 12
- 3.3 Lock file table.lock . . . 12
- 3.4 Storage Managers . . . 13
  - 3.4.1 StandardStMan . . . 13
  - 3.4.2 IncrementalStMan . . . 18
  - 3.4.3 TiledStMan . . . 21
  - 3.4.4 AipsIOStMan . . . 21
  - 3.4.5 Indirect Array File . . . 21
- 3.5 Virtual Column Engines . . . 22
  - 3.5.1 ForwardColumnEngine . . . 22
  - 3.5.2 ScaledComplex . . . 22
  - 3.5.3 VirtualTaQLColumn . . . 22

4 RefTable **22**

5 ConcatTable **22**

6 MultiFile Format **22**
6.1 MultiFile 22
6.1.1 MultiFile version 1 22
6.1.2 MultiFile version 2 23
6.2 MultiHDF5 25

1 Introduction

The Casacore Table Data System (CTDS) is an RDBMS-like system to store data in tables consisting of a number of columns and rows. The table and each column can have an associated set of keywords to define global table data (such as subtable names) or column specific data (such as units). The keywords are meant for small amounts of data. The bulk data will be stored in the column-row cells.

Besides the usual scalar data, a table column or keyword can hold N-dimensional arrays. The data type of a keyword and scalar or array column can be:

- Bool, boolean values which are often stored as bits
- uChar, unsigned 8-bit integer
- Short, signed 16-bit integer
- uShort, unsigned 16-bit integer
- Int, signed 32-bit integer
- uInt, unsigned 32-bit integer
- Int64, signed 64-bit integer
- Float, 32-bit IEEE floating point
- Double, 64-bit IEEE floating point
- Complex, real and imaginary 32-bit float
- DComplex, real and imaginary 64-bit float
- String, ASCII string array with a fixed or variable size

Besides these basic types it is also possible to store (Table)Record objects which are basically maps of name strings to values of one of the data types mentioned above. The records can be nested arbitrarily deeply. When stored in a column, a record is serialized (using AipsIO) to a uChar array and stored as such.

There is a strict distinction between the logical and physical model of the data columns. The table description defines the logical model, while data managers (storage managers and virtual data managers) implement the physical model.

On disk a CTDS table is a directory containing a number of files to hold data and meta data. The formats of these files vary and can have quite complicated structures.
A table can have a zero or more subtables that are usually stored as a subdirectory of the main table. A good example of such a table is the Casacore MeasurementSet (see note 229.

A CTDS table can be a reference to one or more other tables which is comparable to a view in a relational data base. CTDS supports a few types of tables.

- A PlainTable is the basic table type. It contains all data, either stored on disk using storage managers or calculated on-the-fly using virtual data managers.

- A RefTable is a referencing table. It only references rows in another table. Usually it is the result of a select, sort, or iteration operation on another table. The subtables of a RefTable are the same as those of its parent; it references them.

- A ConcatTable is the virtual concatenation of tables with an identical structure (identical columns). It only references the tables to be concatenated. The subtables of a ConcatTable can be concatenated at will. It means that a subtable can be the subtable of the first table or a concatenation of the subtable in all tables.

On disk all table types consist of a directory with the name of the table and several files. The following two files are always present. A PlainTable can contain several more files.

- `table.dat` is a binary file holding the table meta data. It contains the table description and the information how columns are mapped to data managers. It also contains the keyword sets. The first part of its contents is the same for all table types. The remainder is type specific.

- `table.info` is a text file containing a brief description of the table.

This document describes the format of all files that are part of the core CTDS. Note that CTDS can also use arbitrary storage managers dynamically loaded from a shared library. The file formats of such third party storage managers are not described in this document.

CTDS is designed such that it is fully backward compatible. Version information make it possible that even tables from the very first days can be read back despite the fact that the format has changed considerably.
CTDS understands both little and big endian data representations. At table creation time it is decided which one to use which normally is the local endian type.

2 AipsIO

CTDS makes heavily use of the AipsIO class to store its meta data. Therefore a brief description of the AipsIO format is given.
AipsIO is basically a mechanism to serialize a C++ object into a stream of bytes and to read it back. The format consists of a header followed by the object's data. The objects can be nested arbitrary deeply. The maximum total size is 4 GBytes.
The header contains the following fields:

Data Type	Description
uInt	Object length (including the header)
String	Object type
uInt	Version

The version can be used to make software fully backward compatible in case an object changes over time. This feature is used a lot by CTDS to be able to access tables created by older CTDS versions.
The object data are stored after the header as a stream of bytes. The data are unaligned in the stream, thus, say, a float does not need to start at a multiple of 4 bytes.

The length of an object is the total length; it includes the length of the header and the length of possibly nested objects. Of course, each nested object has its own header.
The following native data types are supported. Other data types (classes) can be supported by writing the appropriate shift functions (similar to the C++ std::iostream). A few such objects used by CTDS are described in subsequent sections.

Bool	1 byte
Char and uChar	1 byte
Short and uShort	2 bytes
Int and uInt	4 bytes
Int64 and uInt64	8 bytes
Float	4 bytes IEEE
Double	8 bytes IEEE
Complex	2 Floats
DComplex	2 Doubles
String	uInt length followed by its characters (can be length 0)
C-array	uInt length followed by its elements (can be length 0). A bool C-array is compressed to bits.

AipsIO can store the data in little or big endian order. In fact, it can also handle the old VAX/VMS and IBM/360 data formats.
CTDS only uses the canonical AipsIO format (which is big endian).

2.1 Array object

Templated Array<T> objects are used quite heavily in Casacore. Only arrays with a size up to 2**31 bytes can be stored in AipsIO. An Array object is stored as follows.

header	Array (version 1, 2, or 3)
uInt	number of dimensions
uInt[ndim]	ndim shape values
Int[ndim]	origin (only present for version < 3); it is ignored
T[nelem]	all array values. Bools are compressed to bits.

- Casacore Array objects hold their data in Fortran order (fastest varying axis first). The shape and data are also stored that way.
- In principle the template parameter can be of any type. However, CTDS only uses Arrays of the standard types.
- Until 1995 the Array class supported an origin. It has been removed because it was a source of errors. Theoretically an origin can still be present in Array objects in very old tables. If so, it is read and silently discarded.
- Before RTTI was fully supported in C++, Casacore used its own RTTI functionality. At that time the name of the template parameter was made part of the AipsIO header, for example `Array<Int>`. Once the standard C++ RTTI was used by Casacore, this was not

done anymore and an array of any type has the same header name `Array`. When reading an older Array object from AipsIO, the template part in the name (enclosed in angle brackets) is still recognized, but not used anymore.

2.2 IPosition object

An IPosition object defines a shape of an array or a location in an array. It is stored in AipsIO as follows.

header	IPosition (version 1 or 2)
uInt	number of dimensions
Int[ndim]	shape (for version 1)
Int64[ndim]	shape (for version 2)

Until 2009 a shape element was represented as a 32-bit integer, thereafter as a 64-bit integer. To be as forward compatible as possible, an IPosition is still stored as version 1 if all elements in the shape have a value fitting in a 32-bit integer.

2.3 Block and std::vector object

The templated Casacore Block class is basically the same as the std::vector class. Both are stored in the same way in AipsIO, thus a Block can be read back as a std::vector and vice versa.

header	Block (version 1)
uInt	number of elements
T[n]	the elements

Note that the number of elements is written as an unsigned 32 bit number. This is sufficient since an AipsIO object cannot exceed 4 GB.

2.4 std::map object

Casacore had its own SimpleOrderedMap class before the C++ standard library with its std::map class was developed. Nowadays Casacore only uses std::map, but for backward compatibility an std::map object is serialized in AipsIO in the old SimpleOrderedMap way.
A std::map¡K,V¿ object is stored in AipsIO as follows.

header	SimpleOrderedMap (version 1)
V	default value; not used
uInt	nentries (number of key-value pairs)
uInt	increment (number of entries to add when the map is extended); not used
KV[nentries]	nentries key-value pairs (first the key, then the value)

2.5 TableRecord object

A TableRecord object holds a set of keyword/value pairs. They are used in CTDS to represent the keyword set attached to the table and each column.

A value can be a scalar or array of one the standard data types including complex and string values. Furthermore a value can be a TableRecord in itself; the nesting can be arbitrarily deep. Finally a value can be a Table object which is used to hold the subtables of a table.

A TableRecord is described by a RecordDesc object that also contains the descriptions of possible nested records. A TableRecord can have a fixed or variable format. In CTDS only variable format TableRecords are used.
In the very early days of CTDS a keyword set was stored in a very different way. Reading back such keyword sets is still supported in the code.

A TableRecord is stored in AipsIO as follows:

Data Type	Description
AipsIO header	TableRecord (version 1)
RecordDesc	The TableRecord description
Int	record type (CTDS always uses variable format records)
any	All values are stored consecutively. Their data types are defined in the RecordDesc. Scalars and arrays are stored in the standard AipsIO way. For a nested TableRecord only the values are stored, because its description is part of the record description. However, a nested empty TableRecord is stored as such. If a value is a Table, only the table name is stored. It is tried to make the name relative to the name of the parent table. In that way moving a table has no effect its subtable names. A relative name is prefixed with ././ if the full parent table name has to be used as directory, while the prefix ./ is used if the directory name of the parent table has to be used. Note that in all practical cases (like a MeasurementSet) ././ is used as prefix.

The RecordDesc is stored as follows:

Data Type	Description
AipsIO header	RecordDesc (version 1 or 2)
uInt	number of keywords
String	keyword name
Int	data type (from DataType.h)
Specific	specific info for a value containing an array, Table, or Record. - shape for an array (shape [-1] means it is not fixed). A shape is stored as a vector of Int values. - TableDesc name for a Table (empty name means it is not fixed). - description for a Record.
String	comment to explain the meaning of a keyword (not in version 1)

The latter 4 fields are repeated for each keyword.

3 PlainTable

A PlainTable object represents a basic CTDS table with its own keywords, columns, and associated data managers. It is a directory containing several files named `table.*`.
Besides the files `table.dat` and `table.info` a PlainTable consists of the following files.

- `table.lock` is used to facilitate concurrent SWMR access (single writer, multiple readers) to a table. Locks are acquired and released for this file as described (see note 256. Furthermore, it contains information telling another process if a table has changed.
- Each storage manager has a file `table.f<i>*` to store the data. The storage managers are numbered 0..n and use the number as the suffix `i` in the file name. Note that the numbers do not need to be consecutive. This can happen in case a storage manager has been deleted or a virtual data manager is used.
- A storage manager can use extra files, typically called `table.f<i>i` to store so-called indirect arrays. The TiledStMan can use extra files called `table.f<i>_TSM<j>`.

Note it is possible to combine these files into a single so-called MultiFile to reduce the number of actual files.

3.1 General file table.dat

This file is the main table file. It contains the table definition, the keywords and their values, and the binding of columns to data managers. It is basically a nested AipsIO object containing the following fields.

Data Type	Description
AipsIO header	Table (version 1 or 2)
uInt	number of table rows
uInt	endianness (0=little, 1=big)
String	table type (PlainTable)
TableDesc	table description and table keyword set
TableRecord	table keyword set (in version 1 only)
ColumnSet	column data manager info and column keyword sets

Note that the set of table and column keywords is part of the table description. Only in the very first CTDS version (before 1995), the table keyword set was stored separately. For backward compatibility it is still possible to read such a keyword set, but it is not stored anymore.

It is instructive to use the UNIX `strings` command on a table.dat file. It shows all the strings mentioned in this section.

3.1.1 Table Description

The TableDesc object contains the table description defining the logical view of a table. It contains the name, data type, and some extra info of each column. Furthermore it contains the keyword set attached to the table and each column. Finally it contains a private keyword set that is used by the TableDesc object to hold some extra meta data.

Data Type	Description
AipsIO header	TableDesc (version 1 or 2)
String	name of table description
String	version info of table description
String	comment about table description
TableRecord	table keyword set
TableRecord	private table keyword set (not in version 1)
uInt	ncol (number of columns)
ColumnDesc[ncol]	description of each column

A ColumnDesc object defines a column and is stored as follows.

Data Type	Description
uInt	ColumnDesc version (=1)
String	Type of column using the name of the object containing the description. It can have one of the following values: - ScalarColumnDesc<T - ArrayColumnDesc<T - ScalarRecordColumnDesc - SubTableDesc where T is a string defining the type. It can be: Bool, uChar, Short, uShort, Int, uInt, Int64, float, double, Complex, DComplex, and String.
uInt	BaseColumnDesc version (=1)
String	column name
String	comment
String	default data manager type
String	default data manager group
Int	data type (enum from DataType.h, e.g., TpInt). It must match the type in the column type string.
Int	column options 1 = direct array; meant for small fixed sized arrays for which a storage manager can decide to put it directly. 2 = undefined value can exist (not used). 4 = array has a fixed shape.
uInt	array dimensionality (-1 means scalar)
IPosition	fixed array shape (only for columns containing arrays)
uInt	maximum length of a string value (0 = no maximum)
TableRecord	column keyword set
For a column containing scalars	
uInt	ScalarColumnDesc version (=1)
T	default value (stored according to the data type)
For a column containing arrays	
uInt	ArrayColumnDesc version (=1)
Bool	dummy flag (always False); is present for backward compatibility
For a column containing records	
uInt	ScalarRecordColumnDesc version (=1)
For a column containing subtables	
uInt	SubTableDesc version (=1)
String	TableDesc name
Bool	Flag telling how the description of the subtable is found. True = table description is by name, thus in a file with that name. False = table description is in the next field.
TableDesc	the description of the subtable (only if flag=False)

- A column containing records makes it possible to store arbitrary data in a column. It is handled as an AipsIO byte stream (vector of uChar) by the data managers. No Record

description is kept, so the record can differ from row to row.

- A column containing subtables is supported in the description, but it has never been fully implemented in CTDS, so it cannot be used. This feature is only used in a test program.

3.1.2 ColumnSet

The ColumnSet class contains the information how the logical table columns are mapped to their physical counterparts served by the data managers. As part of the table.dat file it is written into the TableDesc's AipsIO as follows,

Data Type	Description
Int	version as a negative number (only for version > 1)
uInt or Int64	number of rows (Int64 for version >= 3)
Int	StorageOption::Option); only for version >= 3
uInt	StorageOption::blockSize; only for version >= 3
uInt	highest data manager sequence number used
uInt	nrdm; number of data managers used
for each data manager (nrdm entries):	
string	name of the data manager
uInt	sequence number of the data manager (as used in `f<i>` in file name
ColumnInfo[nrcol]	info about each column (see below)
DMInfo[nrdm]	info about each data manager (see below)

- In the first versions of CTDS the version value was not written; instead the first value written was the number of rows (as Int). In later versions the version value was added and was written as a negative number to distinguish it from the non-negative number of rows.
- Usually the highest data manager sequence number is the same as the number of data managers used. However, if a data manager gets deleted (because all its columns are deleted), the number of data managers decreases, but the highest sequence number does not. In this way it is ensured that a new data manager always has a unique sequence number.
- The number of columns is not stored in ColumnSet, but is taken from the Table Description.
- For version 1 the columns are stored in order of name. For later versions they are stored in order of time of addition.

ColumnInfo

For each column some information is stored in the ColumnSet's AipsIO structure.

Data Type	Description
Int TableRecord String derived column info	version (1 or 2) column keywordset; only for version 1 original column name (the name used at table creation, thus without a possible rename)
uInt uInt	version (1) data manager sequence number
for array columns	
Bool IPosition	shapeColDef; has the column a fixed shape? the shape of each array in the column; only if shapeColDef is True

DMInfo

Each data manager can write some short information into the ColumnSet's AipsIO structure. This is kept as an array of bytes, stored as a length (uInt) followed by the bytes. The data manager is responsible for the coding and decoding of these bytes.

Most data managers do not write anything, thus for them only the length 0 is stored. Only the StandardStMan and IncrementalStMan write some info which is described in the **Header** paragraphs of the respective sections.

3.2 Info file table.info

This file is a text file giving some brief info about the table. It is read and written by class `TableInfo`.
It contains the following lines:

- The table type like `Type = MeasurementSet`
- The table subtype like `SubType =`
- Zero or more lines giving some more explanation.

3.3 Lock file table.lock

CTDS supports concurrent access to a table. One writer or multiple readers can be active at the same time. A lock needs to be acquired to access the table, although there is a mode to bypass locking when reading or to bypass locking entirely. See note 256 for more info about how locking is done.

The first few bytes in the lock file are used by the concurrency mechanism of CTDS to manage a read or write lock (on the entire table).

- The first byte is used to acquire/release a lock using the system's file locking mechanism (`fcntl`). The lock can be shared (for read) or exclusive (for write).
- The second byte is used to tell other processes that the table is opened. This is done by creating a shared lock on it.

- The third byte is used to tell other processes that a process holds a permanent lock on the table. It is also using a shared lock.
- The fourth till sixth byte are used for special internal lock purposes.

The lock file contains two data structures to assist processes in acquiring locks and synchronizing their internal data structures with table data changes made in another process. All data are written in big-endian format.

- A list of process identifications is kept in the first 260 bytes of the file. Note that the first 6 bytes of that list are using to do the locking as described above. The list contains the host id and process id of processes wanting to acquire a lock, but cannot because another process is holding the lock.
  The list consists of 4-byte integers, where the first value contains the number of processes in the list. The other values are pairs of host id and process id, so the list can contain up to 32 processes.
- Info telling if and how table data have changed. It is written when a write lock is released. It is used by a process acquiring a lock to synchronize its internal data objects.
  The info is written as an AipsIO stream described below. Bytes 260-263 of the lock file contain the length of the info stream. Thereafter the AipsIO stream is written.

Data Type	Description
AipsIO header	sync (version 1 or 2)
uInt or uInt64	number of table rows (uInt in version 1; uInt64 in version 2)
uInt	number of table columns
uInt	modify counter
uInt	table change counter. The counter is incremented if the `table.dat` file has changed (e.g. by adding a keyword).
Block<uInt>	change counter per data manager. A counter is incremented if that data manager has written data.

A process acquiring a lock detects if the main table file or a data manager file has changed by comparing its change counters with the ones in the lock file.

3.4 Storage Managers

There are several storage managers, each writing the data in its own way. They write their data in files called `table.f<i>` where `<i>` is the sequence number of the storage manager. The sequence number can be seen using the program `showtableinfo`. As described above storage managers can use extra files, in particular an indirect array file.

3.4.1 StandardStMan

The data of the columns bound to this storage manager are stored in equally sized buckets. The buckets are stored sequentially in the file, so the bucket number determines the file offset. Indices tell which rows and columns are contained in the buckets.

StandardStMan uses the main file `table.f<i>` in the table directory where `<i>` is the sequence number. For indirect arrays it uses another file called `table.f<i>i`. The main file consists of four types of data:

- The header is stored in the first 512 bytes of the file and describes some general properties. It is followed by the buckets.
- The data buckets containing the fixed size data.
- String heap buckets containing variable sized strings and string arrays.
- Index buckets containing indices on the data buckets.

However, this initial simple scheme can get more complex when rows and/or columns are added or removed. Such operations can have the effect that buckets do not contain the same number of rows and that possibly new indices are created for groups of new columns. The storage manager will maintain a list of free buckets in case all data from a bucket are removed.

- The addition of rows is done at the end of the last bucket. If it is full, a new bucket is allocated and the indices are updated accordingly.
- The removal of a column is done by adding the space it took to a free space map. Since each bucket has space for N rows, the free space is the same for each bucket. Therefore the free space map is kept in the index. The free space map will combine adjacent free space (if e.g. another column is removed later). When all columns stored in a bucket are removed, all buckets occupied by these columns are added to the free bucket list.
- The addition of a column is done by looking if indices have free space (resulting from column removal). It tries to reuse the free space that fits best. If no free space is available, a new set of buckets is allocated for the column(s) to be added. Also a new index will be created to describe the new set of columns and buckets.
- The removal of a row is done in the bucket(s) containing the row. It is done by shifting for each column in that bucket the data after the deleted row to the left. It means that the offsets of the columns in the buckets do not change. It also means that other buckets ocuupied by these columns are not affected.
- The addition of a row in last bucket.

It is possible to remove rows, which is done by removing the row from its bucket and shifting the other rows in that bucket. Other buckets are not touched. Therefore not all buckets need to contain the same number of rows. An index block tells the starting row of each bucket. If all rows of a bucket are removed, the bucket is added to the empty bucket list. An empty bucket can be reused in a new bucket needs to be acquired when adding data to the table.

Initially all columns are combined in a single set and are stored jointly in the buckets. However, if columns are added to a table, they do not fit in the current set of buckets. They will be stored as a new set of columns having its own index block.

Small fixed sized data are stored directly in a data bucket. Such data are numeric scalar values and small arrays such as UVW coordinates. Small strings (<= 8 bytes) and fixed sized strings are

also stored directly. Larger strings are stored in heap buckets which are referred to in the data buckets. Strings can be span multiple heap buckets, thus it is possible to store extremely large strings.

Header

The header block is contained in the first 512 bytes of the `table.f<i>` file. It contains the following AipsIO structure.

Data Type	Description
AipsIO header	StandardStMan (version 1, 2, 3 or 4)
Bool	data stored in big endian format? (not present for version <= 2)
uInt	bucket size in bytes
uInt	number of buckets
uInt	Persistent cache size (in buckets)
uInt	number of free buckets
Int	first free bucket (-1 is no free bucket)
uInt	number of index buckets
Int	first index bucket
uInt	offset of index in bucket (not present for version 1); if >0, index fits in a single bucket
Int	last string heap bucket (-1 is no string heap)
uInt	index length (in bytes)
uInt	number of indices

DMInfo

StandardStMan stores the following data manager info in the DMInfo part of the ColumnSet AipsIO structure.

uInt[ncolumn]	the number of the column set each column belongs to
uInt[ncolumn]	the offset of each column in the data buckets

The number of columns is defined in the table description.

Data Buckets

The columns handled by this storage manager are grouped into sets. At table creation time there is a single set, but sets might be added as columns are added to this storage manager at a later stage. Similarly, sets might be removed as columns are removed. Each set of columns has its own set of data buckets and its own index to describe the contents of the buckets.

For a particular set of columns a bucket can store N rows where N depends on the size of the bucket and the sizes of the columns' data types. The data are stored in a columnar way in each bucket, thus each bucket contains N rows of column 1, thereafter N rows of column 2, etc. without any padding. Boolean values are storeds as bits. Note that the column data types are defined in the Table Description and not in the storage manager. The offset of a column is fixed and the same in the data buckets is fixed. It will never change, even when rows or columns are removed. In

principle the first bucket contains rows 0..N, the next bucket contains rows N..2N, etc. However, this can change as rows are deleted. Row deletion only affects the bucket the row is in.
The index defines which rows are contained in the buckets by keeping the last row number for each bucket. The index is described in a next paragraph.

Data with a fixed length are stored directly in the data buckets, other data (variable length strings and arrays) are stored elsewhere and referred to from the data buckets. StandardStMan can store all scalar and array following data types supported by CTDS (as little or big endian).

Data Type	Size	Directly
Scalar		
Bool	1 byte	yes
uChar	1 byte	yes
Short	1 byte	yes
uShort	2 byte	yes
Int	4 byte	yes
uInt	4 byte	yes
Int64	8 byte	yes
Float	4 byte	yes
Double	8 bytes	yes
Complex	8 bytes	yes
DComplex	16 bytes	yes
String with max length	maxlen bytes	yes; has trailing zero if string size < maxlen
Variable string <= 8 bytes	12 bytes	yes; 8 string bytes followed by Int giving length
Variable string > 8 bytes	12 bytes	reference to heap bucket as Int[3] (bucketnr, offset, le
Direct Arrays		
Bool .. DComplex	nelem * data type size	yes; (the shape is known in the Table Description); Bo
String	12 bytes	reference to heap bucket as Int[3] (bucketnr, offset, le
Indirect Arrays		
all types	8 bytes	offset in the indirect file

String Heap Buckets

These buckets contain arrays of strings and variable length scalar strings. Each bucket start with a small header of 4 integers whereafter the string data are stored.

Data Type	Description
Int	reserved for the free bucket list
Int	used length; the number of bytes used (including gaps), thus the next free byte in the bucket.
Int	ndeleted; the total length of the gaps arising from deletion or updating a value with a shorter string.
Int	next bucket; the bucket containing the possible continuation of the last string (array) in this bucket. -1 means no continuation. Note that the continuation can be continued again.

Strings are stored consecutevily in the buckets. A long string and an array of strings are continued in another bucket if they do not fit entirely in the current bucket. As described above the continuation is continued as well if it is very, very long.

It is possible in CTDS to change an existing value, thus a string (array) can be replaced by another one. If possible the new value is written at the same location, where a gap arises if the new value is shorter. If it is longer, it has to be stored at a new location and the old location will be a gap. Similarly, a gap arises if a string is deleted because a row or column is deleted. Only the total size of gaps is administered, not their locations. Thus usually gap space will be lost. Only if it is at the end of the used part of a bucket it can be deducted from the used length and be reused. Note that if all string space in a bucket is deleted, the bucket will be added to the free bucket list.

A string (array) is stored in the heap buckets depending on its type.

- For a scalar string only its characters are stored. Its length is already stored in the data bucket.
- A fixed shaped array of strings is stored as a string of bytes. Each string is stored as a length-value pair. The length is a 32-bit integer and a value is the formed by the string characters. Note that the shape of the array is known in the table description, thus does not need to be stored.
- A variable shaped array of string is stored in the same way as the fixed shaped array, but is preceeded by the shape and a boolean flag. The shape is stored as an array of `ndim+1` 32-bit integers; the number of axes followed by the axes lengths. A True flag value indicates that the shape is followed by string data because it is possible that the array shape is defined independently from the writing of the actual strings.

The data bucket refers to the string (array) by means of a bucket number, offset and length. In case of continuation that bucket number is the first bucket. The length is the total length, thus including shape and flag if applicable.

Index Buckets

The overall index contains one or more indices, one for each set of columns. An index defines the rows contained in the buckets used by the set of columns. This is done by keeping the last row number per bucket. The index is needed to cope with the possibly variable number of rows in the buckets due to removed rows.

An index also contains information about free space in the data buckets due to removed columns because that space can possibly be reused when a column is added later. The map consists of a `std::map<Int,Int>` object telling the offset and length of each free space part in a data bucket. Note it is the same for each data bucket described by the index.

The indices to write are serialized and stored in one or more buckets. The first 8 bytes of the index buckets contain the bucket number of the next part of the serialized indices as 2 big-endian signed integer values. This could be done in 4 bytes, but for redundancy purposes it is done twice. A value -1 indicates no next index bucket.

When rewriting the index, care is taken that it is done in different buckets to assure that the index is always present in case the system crashes in the middle of writing the index. Once the index is written successfully, the header is updated and the old index buckets are removed and

added to the free bucket list.
If the serialized index fits in half a bucket (which is often the case), that bucket is reused in subsequent index rewrites by alternating between both parts of the bucket. The index offset field in the header (see above) tells the offset of the index in the bucket.

The indices are serialized in the following AipsIO structure. Note that the number of indices is part of the header.

Data Type	Description
AipsIO header	SSMIndex (version 1 or 2)
uInt	Number of index entries
uInt	Number of rows fitting in a bucket
Int	Number of columns served by this index
std::map<Int,Int>	Free space map
Block<uInt> or Block<Int64>	Last row in each bucket; Int64 for version >1

Free Buckets

A bucket is added to of the free bucket list once it does not contain any data anymore. The first free bucket is given in the header, the others form a list by maintaining the next bucket number in the first 4 bytes of the bucket in big endian format.

Actually, a free bucket is added to the head of the list to avoid having to update the last free bucket. Thus the current first free bucket is put at the start of the new free bucket, which in its turn becomes the first free bucket to be stored in the header.

3.4.2 IncrementalStMan

The IncrementalStMan is meant for data that changes seldomly, so they can be stored in a compressed way by storing the number of rows the data value is the same. It can be done for scalars as well as arrays.

The storage manager cannot add nor remove columns, but it can add and remove rows. The data structures in the files are designed for these properties.

Header

The header block is contained in the first 512 bytes of the `table.f<i>` file. It contains the following AipsIO structure.

Data Type	Description
AipsIO header	IncrementalStMan (version 1, 2, 3, 4 or 5)
Bool	data stored in big endian format? (not present for version <= 2=4)
uInt	bucket size in bytes
uInt	number of buckets
uInt	Persistent cache size (in buckets)
uInt	unique column number
columns added	
uInt	number of free buckets
Int	first free bucket (-1 is no free bucket)

- The unique column number was only used by older versions (<= 2) of the IncrementalStMan. For those older versions each indirect array column stored its data in a separate file for which the unique column number to make the file name unique. The current version does not do that, but uses a single file to store all indirect array column data.

DMInfo

IncrementalStMan stores the following data manager info in the DMInfo part of the ColumnSet AipsIO structure.

uInt[ncolumn]	the number of the column set each column belongs to
uInt[ncolumn]	the offset of each column in the data buckets

The number of columns is defined in the table description.

Data Bucket

A data bucket contains the data of all columns for a given number of rows. It is split into a data part (at the beginning of each bucket) and an index part (at the end of each bucket). The first 4 bytes of each bucket give the offset of the index part (thus also imply the length of the data part). The endianness of the data and index part is the same as the endianness of the table.

Data Type	Description
uInt	offset of the index in this bucket. The high byte of the offset defines if row numbers are stored as 32 or 64 bits (0 = 32 bits, 1 = 64 bits). Note that this does not need to be the first byte, because that depends on the endianness.
byte[offset-4]	the data part
byte[N]	the index part as described below

Data rows can be added to a bucket until it is full, after which a new bucket is created. When a value in a bucket gets updated, it may not fit in the bucket anymore. In that case the bucket is split and a new bucket is created. When a row is removed, a bucket may get empty and added to the free bucket list.
Note that this storage manager cannot remove columns.

Data Part

The data part contains the values of all columns bound to this storage manager. Scalars and fixed shaped arrays of all data types (including strings) can be stored. However, a string cannot span buckets. Variable sized arrays are stored in the indirect array file; their offsets in that file are stored in the bucket's data part.

All values in the data part are stored consecutively. Bool arrays are stored as bits. The index part points to the correct data value. Note that the shape of a fixed shaped array is not stored since it is part of the table description. The shape of a variable sized array is stored with its data in the indirect array file.

Index Part

The bucket index defines the starting and end row of the data in a bucket. The index part in each bucket defines per column which rows have which values. Subsequent rows can have the same value, thus the index part defines the first and last row number in the bucket having that value. In fact, it only defines the first row number, because the first row number of the next value defines the last row number of this value. Note that these row numbers start at 0. The bucket index defines the actual row number; it also defines the last row number of the last value.

The index part contains per column the following information. It is stored consecutively for all columns.

Data Type	Description
uInt	number of values
uInt or Int64[nr]	first row number in bucket having the corresponding value. It starts at 0.
uInt[nr]	offset in data part for value given by the row number

- The row numbers are given relatively to the first row in the bucket. Thus if a bucket contains rows N till N+M, the values of the row numbers in the index part of that bucket is between 0 and M.
- The row numbers are stored as a 32-bit or 64-bit integers which depends on the highest (relative) row number in the bucket. The high byte in the offset value at the beginning of the bucket tells if it is 32 or 64 bits.
  Note that until 2018 Casacore stored the row numbers always as 32 bits. The new way of storing them as 32 or 64 bits is fully backwards compatible since the high byte of the offset was always 0.

Bucket Index

The bucket index defines which rows are contained in each bucket. It is written at the end of the file, thus right after the last bucket, as an AipsIO structure with the following fields.

Data Type	Description
AipsIO header	ISMIndex (version 1 or 2)
uInt	number of buckets
uInt or Int64[nr+1]	first row number in each bucket

- Version 1 is used if row numbers are stored as 32-bit integers, version 2 is used for 64-bit integers. The highest row number determines how they are stored.
- The last row number stored is the number of rows + 1; it can be seen as the first row number if a next bucket.

Free Buckets

A bucket is added to of the free bucket list once it does not contain any data anymore. The first free bucket is given in the header, the others form a list by maintaining the next bucket number in the first 4 bytes of the bucket in big endian format.

Actually, a free bucket is added to the head of the list to avoid having to update the last free bucket. Thus the current first free bucket is put (as the next free bucket) at the start of the new free bucket, which in its turn becomes the first free bucket to be stored in the header.

3.4.3 TiledStMan

The Tiled Storage Manager stores the data in a tiled way to achieve that access along the different axes is about equally fast. It is similar to the chunked storage of data arrays in HDF5. This storage manager can only be used for columns containing array with a fixed sized data type, thus scalar columns and string array columns cannot be stored. There are a few flavours.

- TiledColumnStMan
- TiledShapeStMan
- TiledCellStMan
- TiledDataStMan is obsolete.

- index - cell, column, shape stman

3.4.4 AipsIOStMan

3.4.5 Indirect Array File

Variable shaped arrays cannot be stored directly in most storage managers. Instead they are stored in a separate file. The offset in that file is stored by the storage managers.

3.5 Virtual Column Engines

3.5.1 ForwardColumnEngine

3.5.2 ScaledComplex

3.5.3 VirtualTaQLColumn

4 RefTable

5 ConcatTable

6 MultiFile Format

The MultiFile format has been designed to reduce the number of files. CTDS uses one or more files per storage manager which can result in quite a large number of files for a table. In particular a MeasurementSet and its subtables can consist of dozens of files. Often these files are quite small mapping poorly to more modern file systems such as Lustre, which use large IO blocks.
The MultiFile format has been designed to combine all these files into a single container file. Furthermore, MultiFile has the option to store a CRC value for each block to ensure data are read back correctly.

The MultiFile format comes in 2 flavours:
1. The MultiFile which is a regular file with an header to denote all individual files in it.
2. The MultiHDF5 which is an HDF5 file containing a dataset per individual file.
They are described in more detail below.

6.1 MultiFile

A MultiFile is a binary file divided into (large) blocks of equal size. Each CTDS file is stored as a virtual file in one or more blocks in the MultiFIle. A header describes the MultiFile layout and the virtual files. It contains an index telling the name of the CTDS files and the block numbers containing their data. The header also contains a list of free blocks which arise when a CTDS file is deleted or truncated.

The MultiFile concept has evolved over time. The second version is more powerful and more robust than the first version. In both versions the header is maintained in memory and occasionally flushed to disk. It makes use of the AipsIO mechanism to serialize the header and to store it in the first block of the MultiFile. Continuation blocks are used if it is too large for a single block.

6.1.1 MultiFile version 1

Version 1 of the MultiFile can contain the files of a single CTDS table. It does not support nested MultiFiles as version 2 does. Neither does it support CRC values. Header continuation blocks are not stored in the MultiFile itself, but in a small separate file with the extra extension `_hdrext` in its file name.
The header is stored as follows:

Data Type	Description
Int64	header size in bytes
Int64	block size in bytes
Int64	counter keeping track how often the header was written. It is used to know if changes have been made by another process requiring the header to be reread.
AipsIO	AipsIO structure containing the entire MultiFile index

The MultiFile index is serialized in the following AipsIO structure.

Data Type	Description
AipsIO header	MultiFile (version 1)
Int64	Total number of blocks used in the MultiFile (including free blocks)
C-array of FileInfo	Vector of objects containing the info of each CTDS file in the MultiFile
C-array of Int64	Vector of free blocks

Each FileInfo entry contains:

Data Type	Description
String	CTDS file name
C-array of Int64	Vector of MultiFile block numbers used for this CTDS file
Int64	Total size in bytes of the CTDS file

6.1.2 MultiFile version 2

Version 2 of the MultiFile can contain multiple CTDS files (as version 1 does), but it also supports nested MultiFiles. Nested MultiFiles are used to store subtables in the MultiFile of the parent table to achieve that an entire MeasurementSet is stored in a single file. A nested MultiFile is the same as any other MultiFile, thus having its own header, index, block size, etc. It is stored as a single file in the parent MultiFile. However, a nested MultiFile does not have CRC values.
The header is stored as follows:

Offset	Data Type	Description
0	Int64	0 (to make it different from version 1)
8	Int64	blocknr of first continuation header block (0 = no continuation)
16	Int64	counter keeping track how often the header was written. It is used to know if changes have been made by another process requiring the header to be reread.
24	Int32	version (=2)
28	uInt32	CRC of entire header (using 0 for this CRC value)
32	Int64	header size in bytes
40	Int64	block size in bytes
48	Int64	Total number of blocks used in the MultiFile
56	char	use CRC?
57	char[7]	spare
64	AipsIO	AipsIO structure containing the MultiFile index
	nfile*FileInfo	general info per file
	nfile*Index	packed index per file
	Index	packed index of free blocks
	Int64	nCRC (0 if CRCs are not used)
	nCRC*uInt32	CRC value per block
	Int64	number of available blocks of first header continuation
	n*Int64	block numbers available for first header continuation
	Int64	number of available blocks of second header continuation
	n*Int64	block numbers available for second header continuation
	2*Int64	number of block actually used for first and second continuation

There are two header continuation blocks to improve robustness. They are used alternately when the header is flushed to disk. The first block (at offset 0 of the file) it written last. This ensures there is always a valid header in case of a crash. Note that new continuation blocks are added as needed. When added, the header is serialized again to contain those new block numbers. In case fewer continuation blocks are needed than there are available, the superfluous blocks are not added to the free list but are kept available.
The MultiFile index is serialized in the following AipsIO structure.

Data Type	Description
AipsIO header	MultiFile (version 1)
C-array of FileInfo	Vector of objects containing the info of each CTDS file in the MultiFile

Each FileInfo entry looks as:

Data Type	Description
String	CTDS file name
Int64	Total size in bytes of the CTDS file
Bool	Is this file a nested MultiFile?

A packed index is a compressed form of the index telling which blocks are used by a file. It is

compressed by storing a repeat count for consecutive block numbers. The repeat count is a negative value to make it different from the block numbers. For example:

```
10,-3,17   means block   10,11,12,17
```

6.2 MultiHDF5