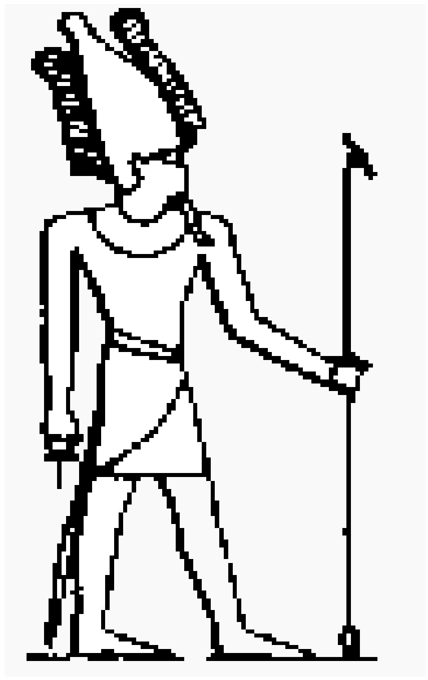

ONYX PROGRAMMER’S REFERENCE MANUAL

Version:	1.1
Date:	16 July, 2009
Author:	Nick Lloyd.

Contents:

Document History 5
Introduction 6
  Intended Audience 7
  Usage on Multiple Experiments 7
  HDF Requirements 7
  Onyx Compilation Requirements 8
Programming Overview 8
  Onyx Hierarchy 9
Onyx COM Objects 12
  Objects 12
  OnyxArray 13
  OnyxConfig 14
  OnyxContainer 15
  OnyxDataBase 16
  OnyxDataObject 17
  OnyxIterator 18
Onyx Structures and Enumerations 19
  enum ONYX_ACCESS_MODE 20
  enum VARENUM 21
  enum ONYX_COLLECTION_TYPE 23
  ONYXFIELDDESC 24
  ONYXCONTAINERDESC 26
  ONYX_CONFIGURATION_KEY 27
  ONYX_CONFIGURATION_EPOCHKEY 28
  ONYX_TABLE_ENTRY 29
Onyx COM Interfaces 30
IOnyxArray 31
  Methods 31
  IOnyxArray::GetArray 32
  IOnyxArray::ReleaseData 33
  IOnyxArray::SetNewSize 34
  IOnyxArray::SizeofElement 35
IOnyxCollection 36
  Methods 36
  IOnyxCollection::Clone 37
  IOnyxCollection::CopyFrom 38
  IOnyxCollection::CreateIterator 39
  IOnyxCollection::CreateRecord 40
  IOnyxCollection::Erase 41
  IOnyxCollection::Find 42
  IOnyxCollection::FindEntriesBounding 44
  IOnyxCollection::GetDescriptor 46
  IOnyxCollection::GetSortIndex 47
  IOnyxCollection::Insert 48
  IOnyxCollection::Reserve 49
  IOnyxCollection::SetSortIndex 50
  IOnyxCollection::Size 51
  IOnyxCollection::Sort 52
IOnyxConfig 53
  Methods 53

IOnyxConfig::ConfigSectionKeyValue 54
IOnyxConfig::CreateKeyRecord 55
IOnyxConfig::EpochSectionKeyValue....55
IOnyxConfig::FindSection 57
IOnyxConfig::GetFilename....58
IOnyxContainer 59
Methods....59
IOnyxContainer::Attach 60
IOnyxContainer::Detach....61
IOnyxContainer::Insert 62
IOnyxContainer::Load....63
IOnyxContainer::Name 64
IOnyxContainer::PartialLoad 65
IOnyxContainer::TableSize 66
IOnyxContainer::Write 67
IOnyxDatabase....68
Methods....68
IOnyxDatabase::AddContainerDefn....69
IOnyxDataBase::CreateContainer 70
IOnyxDatabase::CreateIndexObject....71
IOnyxDatabase::GetTablesInFile 72
IOnyxDataBase::LoadConfiguration 73
IOnyxDatabase::SetDatabaseClsid....74
IOnyxDataBase::SetDefaultLogger....75
IOnyxDataBase::SetMaxNumberOfOpenFiles 76
IOnyxDatabaseIndex 77
Methods....77
IOnyxDatabaseIndex::AllocateIndex 78
IOnyxDatabaseIndex::ConfigureAsIndirectIndex....79
IOnyxDatabaseIndex::CheckEpoch 80
IOnyxDatabaseIndex::SetGuardAndSort....83
IOnyxDataObject 84
Methods....84
IOnyxDataObject::IsKindOf 85
IOnyxDataObject::CopyCStruct....86
IOnyxDataObject::GetCStructPtr 87
IOnyxDataObject::Dereference 88
IOnyxDataObject::Clone 89
IOnyxDataObject::ReleaseData 90
IOnyxDataObject::KeyIndexIsLessThan....91
IOnyxFileLocator....92
Methods....92
IOnyxFileLocator::LocateFile 93
IOnyxIterator 95
Methods....95
IOnyxIterator::Eof....96
IOnyxIterator::Erase 97
IOnyxIterator::First 98
IOnyxIterator::Insert....99
IOnyxIterator::Last 100
IOnyxIterator::MoveTo 101
IOnyxIterator::Next....102
IOnyxIterator::Previous 103

IOnyxIterator::Value 104

Document History

V 1.0, November 9, 1998

Original draft version.

V 1.1, July 13, 1999

New specification requires re-compilation of all existing code based on V 1.0. This is not considered difficult since very few versions exist at the current time. The following modifications have been made:-

1. **IOnyxCollection::CreateCriteria** has been removed since it is not likely this method will be implemented in the foreseeable future. Should have no impact as its functionality was never defined.
2. **IOnyxDatabase::CreateIndex** has been added to provide support for database indexing within the same interface. No impact on existing software.
3. **IOnyxIndex** interface has been added. No impact on existing software.
4. **IOnyxFileLocator** interface has been added. No impact on existing software.
5. **IOnyxCollection::PartialLoad** has been added to support efficient joining of neighbouring files. No impact on existing software
6. Miscellaneous minor edits which have no impact on existing software

Introduction

ONYX is a software interface which provides access to scientific databases in a format suitable for subsequent analysis and processing by scientific programmers.

To most programmers the phrase "*database*" conjures up images of SQL, Access, Paradox, Oracle and other industry standard relational databases. This is not the same as scientific databases. Scientific databases are often very large, perhaps exceeding a terrabyte of storage, and often consist of thousands of individual files. The scientific database must be portable to many heterogeneous platforms. Finally the scientific programmer normallys wants "C" structures to represent individual records of data.

The ability to sort, filter and select various records in a scientific database is a concern but does not have the same degree of importance as in a relational database. In addition the ability to update individual records, independently of others, is also not a concern as most scientific database files are created in entirety within a single processing step. Once distributed to scientists/users the individual files are accessed in read-only mode.

For illustration we examine the case of OSIRIS, an optical spectrograph and Infra-Red imager, that is on-board the ODIN spacecraft. This instrument continuously collects data throughout the mission's lifetime and generates two basic, instrumental products: optical spectra and infra-red height profiles. The mission guidelines require that Level I data products be archived one orbit at a time. Hence the OSIRIS level I products will generate one set of files for each orbit. Given that OSIRIS will collect data on approximately 5000 orbits then the OSIRIS scientific database will consist of over 10,000 files, assuming one file per orbit for optical spectra and one file per orbit for infra-red data.

On a typical orbit OSIRIS will collect approximately 3000 to 6000 individual spectra from the optical spectrograph. Each exposure consists of a relatively small header and associated data and errors arrays. The C structure that represents the header is approximately 100 bytes while the data and error arrays consume over 13000 bytes. Hence the total size of the file required to store the data for one orbit is between 40 and 80 megabytes.

Speed of access is an important issue in scientific databases. ONYX takes a simple, middle-ground, approach to this problem. It splits the data record into two components: a small volume header and a large volume data/error component. The relatively small headers are written into one table within a file. The large volume data/error arrays are written to other "hidden" tables which are linked to the small volume header using special pointers called **IOnyxArray***.

The end result is that the user can efficiently read and cache all of the exposure headers for an entire orbit in a single read. Since the file has less than 10,000 records and the size of the header is 100 bytes less than 1 megabyte of memory is required to cache all of the headers. This is offers convenience as the user/scientist can quickly read in all of the headers, examine each one and then select only those exposures which are of value. ONYX then provides the mechanism to fetch the large volume data only for the selected records.

In this manner the user has complete freedom to match how much data is stored in the system's virtual memory. If they wish they can read an entire file, header and data, into virtual memory. Alternatively, more modest systems may choose to only read a few records at a time to avoid the disk-thrashing as the virtual memory swaps in and out of memory.

In summary, ONYX has the following features,

- It provides fast and efficient mapping from internal records to "C" structures used by scientific programmers.
- ONYX implicitly allows a database to be spraw out over thousands of files.
- ONYX works in cooperation with portable data formats such as HDF.
- ONYX allows rapid access to header records and links the header record to large volume data/error arrays using a special **IOnyxArray*** interface.

Finally ONYX is not an acronym but is simply a play on the phrase "**O**siris **Nicks** or **O-NYX**".

Intended Audience

This document make frequent references to COM, The Component Object Model as developed by Microsoft. It is strongly focussed on COM objects and the interfaces they expose to the outside world which is particularly well suited for C++ programming. This document assumes the reader understands COM and C++ programming and in particular the reader should understand,

- The difference between COM objects and the interfaces they expose.
- The **IUnknown** interface and its implications through the methods **QueryInterface**, **AddRef** and **Release**.
- COM aggregation.
- C++ including templates and a working knowledge of the ANSI Standard template library (STL).

Documentation on COM can be purchased from Microsoft Press. The definitive primer is usually referenced as Kraig Brockschmidt's *Inside OLE* which is no longer in press; a suitable substitution is *Inside COM* by Dale Rogerson, Microsoft Press, ISBN 1-57231-349-8. Documentation on the ANSI Standard Template Library is easily found on the Web by doing a search for STL or Standard Template Library. A suitable book is *STL Tutorial And Reference Guide* by Musser and Saini, Addison Wesley ISBN 0-201-63398-1.

Usage on Multiple Experiments

ONYX is designed so it has a core component that may be re-used on multiple experiments via COM aggregation. The experience of this author is that most scientific experiments would fit well within an ONYX framework.

The basic strategy when developing an ONYX database is to implement a small COM object (**OnyxDatabase** object) unique to each experiment. This object aggregates a common core OnyxDatabase object and then initializes it with all of the "C" structures specific to that experiment. The process is relatively simple and can be developed within a short period of time.

HDF Requirements

The current version of ONYX uses NCSA HDF, the Hierarchical Data Format, to store data to and from files. Full documentation and source code may be found at http://hdf.ncsa.uiuc.edu/. The current version of ONYX uses version 4.1 of HDF.

ONYX makes extensive usage of the HDF VDATA structure and in addition uses a feature new to version 4 that allows a user to append records to the end of the VDATA. Earlier versions of HDF did not permit multiple appends to the end of a VDATA and so ONYX must be built with version 4.1 or higher.

The interfaces exposed by ONYX are intentionally devoid of any reference to HDF and in theory implementations could be developed that do not use HDF. The basic concept of ONYX is one of multiple tables within a file. Clearly this paradigm easily translates to other storage mechanisms including relational databases, CDF and COM/OLE compound files. There are no plans at present to support any of these alternative storage mechanisms.

Onyx Compilation Requirements

ONYX is portable to multiple platforms. On Windows platforms it is generally implemented as Dynamic Link Libraries while on Unix platforms it is generally implemented as shareable objects. Since "C" structures are passed back and forth between ONYX and user programs it is important that both ONYX and user programs are compiled with the same compiler switches controlling structure alignment boundaries. Failure to do so will produce undesired effects.

ONYX uses three functions, **CoCreateInstance, CoInitialize** and **CoUninitialize** from the Microsoft COM API. These functions must be emulated on non-Microsoft operating systems. Suitable implementations have been developed for UNIX systems that support shareable objects.

Programming Overview

ONYX was designed to meet the database needs of the OSIRIS instrument on-board the Swedish ODIN spacecraft, due for launch in 2000 and provides a good example of the overall design goals. The database consists of several sets of primary data:-

1. The scientific data derived from the OSIRIS instrument are archived on an orbit by orbit basis. They typically consist of 2000 to 6000 exposures occupying between 5 and 50 MB.
2. Instrument calibration tables which change relatively infrequently and is very heterogeneous in nature.
3. Spacecraft position and velocity data. About 400 records stored on an orbit by orbit basis
4. Spacecraft attitude data. About 4000 records stored on an orbit by orbit basis
5. Spacecraft time conversion tables. One record per orbit.

It is anticipated that ODIN will collect data for 5000 to 10000 orbits

The OSIRIS scientists wanted the data to be accessible as "C" structures on multiple platforms as much of the actual scientific analysis is performed in C or C++. The portability is one of the main reasons for choosing HDF. The "C" structure is normally used to provide header information while the corresponding data are represented as arrays. A typical feature is that the header is significantly smaller than the scientific data arrays.

This overall structure of data is typical of many experiments. Other instruments might archive data on day by day, hour by hour or session by session but by necessity they almost always archive data in a systematic manner.

The overall paradigm used by ONYX is to read entire arrays of “C” structure headers into memory in one go while leaving the scientific data arrays on disk. This gets all of the information about the data set into memory where it can be quickly and efficiently manipulated, sorted and selected. ONYX however provides linkage mechanisms that automatically associate the header structure with the corresponding scientific data arrays.

ONYX also provides mechanisms that link ancilliary data records to the scientific data. On OSIRIS, for example, a frequent request is to acquire the spacecraft position at the time of a measurement. In essence this requires the software to find the spacecraft position file that corresponds to the requested time and then search within that file for the records surrounding that time. A specific algorithm must then interpolate between these records to the time of interest.

Finally ONYX links configuration information to the scientific data. ONYX expands on the text-based “INI” files used in Windows 3.1 to store various configuration information that changes relatively slowly in time. For example, the OSIRIS spectrograph stores the read out mode in the header as a number. At the beginning of the mission the value 3 may mean one thing but as the mission progresses may change its meaning. These values can easily stored in text based files.

In conclusion ONYX provides a paradigm for the creation of scientific databases. It does not pretend to be a relational database as it is trying to fit a niche where relational databases are relatively weak. On the other hand it is trying to build upon the foundation built by the HDF and HDF-EOS projects.

Onyx Hierarchy

In this section we give a brief overview of the ONYX hierarchy and a simple introduction to the programming interface.

ONYX uses a hierarchical paradigm. At the highest level is the OnyxDatabase object. This object coordinates all other access to the database for a given experiment. Generally a small OnyxDatabase object, aggregated on the larger core object, is created for each experiment and its main purpose is to define all of the mappings from user “C” structures to HDF Vdata fields.

Onyx Database

The only way to create an **OnyxDatabase** object is to call the COM API function **CoCreateInstance**. The CLSID passed to CoCreateInstance must be the CLSID of the required experiment. The **OnyxDatabase** object exposes the IOnyxDatabase* interface which is used as the primary entry point into this experiment.

Onyx Container

Given an **OnyxDatabase** object we can invoke method **CreateContainer** to create an empty OnyxContainer object. The OnyxContainer object’s primary function is provide input and output for database tables where each record of each table is ultimately stored in memory as a “C” structure. The implementation takes into account compiler alignment issues and handles all of the necessary conversions from the user’s “C” structure to the byte packed structures required by HDF.

The **IOnyxContainer*** interface is used to fill the OnyxContainer object with data. To read a table from an existing file, first **Attach** to an existing file and then **Load** the table. Alternatively to create a new table then call **Attach** to a new file. Then repeatedly call **CreateRecord** to create a new blank record, modify the "C" structure and then **Write** to save the record to disk. Finally a call to **Detach** flushes any cached records.

Onyx Collection

The OnyxContainer object also exposes the **IOnyxCollection** interface which is used to manipulate and manage the array of records stored in the table. The interface is closely modelled on the ANSI C++ Standard Template Library (STL). The interface can be used to copy, insert, erase, sort and iterate over the collection of records. Of all of these operations iteration is the most important as it is the only mechanism by which a user can access individual records.

Onyx Iterator

The method **IOnyxCollection::CreateIterator** is used to create an OnyxIterator object which may be used to iterate over the table. OnyxIterators are always created so they point to the first record of a table. The user can then use various operations, **Next, Previous, First, Last,** to move through the table and can test for the last record using **Eof**.

Once positioned at the appropriate record a call to method **Value** returns the OnyxDataObject used to represent individual record in a table.

OnyxDataObject

Each record in the table is represented as another COM object, OnyxDataObject which exposes the interface IOnyxDataObject. Each **OnyxDataObject** has its own memory allocated for the "C" structure associated with it. **IOnyxDataObject** provides the method **GetCStructPtr** to get a memory pointer to this "C" structure.

The **OnyxDataObject** also knows how to dereference any **HOPTR** that point to data/error arrays stored elsewhere in the file. Interface **IOnyxDataObject*** provides method **Dereference** specifically for this purpose.

In reality, the special **HOPTR** pointers are actually stored in the record's "C" structure as IOnyxArray* interfaces belonging to associated OnyxArray objects. The onus falls on the experimental database design since the database designer simply chooses which fields are stored right in the header table and which fields are stored as **HOPTR**. ONYX, itself, has no built-in limits as to how many HOPTR can be associated with a specific record. However HDF does limit the length of one record to less than 32 Kbytes and the database designer must ensure this constraint.

The **IOnyxArray*** interface provides method **GetArray** which provides the user with a pointer to the large volume data array. Other methods provide capability to change and query the size of the array. Once the user is finished with the large volume array they should call **IOnyxArray::ReleaseData** to release the memory associated with this array. Failure to do this may result in excessive demands on virtual memory if too many records have their large volume arrays created simultaneously.

Finally, ONYX uses the method **IOnyxDatabase::LoadConfiguration** to support the use of text based configuration files via the OnyxConfig object. It is expected that configuration files would contain low-volume calibration data, switches for various processing elements etc. etc.

The **OnyxConfig** object implements the configuration files as a collection of sections where each section is a collectionof keys. The **IOnyxConfig*** interface provides a method to “look-up” the collection of keys associated with a specific section in the file. The function returns a pointer to the **IOnyxCollection*** interface which may be used to efficiently locate keys within the collection.

Onyx COM Objects

The following is a list COM objects used in ONYX. A link to structures and enumerations used by ONYX is also included.

Objects

OnyxArray
OnyxConfig
OnyxContainer
OnyxDatabase
OnyxDataObject
OnyxIterator

OnyxArray

COM object used to represent a pointer to data arrays stored elsewhere in an ONYX file. The object is able to locate/store the associated data array inside the database file. The **OnyxArray** is created for all fields in "C" structures that have the **ONYX_FLAG_ISHOPTR** bit set in the ONYXFIELDDESC *flags* field. The corresponding field in the "C" structure must be of type **IOnyxArray***.

Interfaces Exposed

IOnyxArray*	Used to access and manipulate the associated array

Return to Onyx COM Objects

OnyxConfig

COM object used to fetch configuration information from text-based configuration files. to acquire “keyed” configuration information from text based files. The concept is based upon the ***.ini** files found on older versions of Microsoft Windows. These configuration files contain multiple sections and within each section are mutiple keys with assigned string values.

Example Configuration file

```
[Keyed Section Name A]
key1 = string value 1
key2 = string value 2
key3 = string value 3
…
[Keyed Section Name B]
key1 = string value 1
key2 = string value 2
key3 = string value 3
…
<Epoch Section Name>
1998-12-31 23:59:59 = string value 1
1999-05-15 00:00:00 = string value 2
1999-06-01 08:25:34 = string value 3
                         …
```

The **OnyxConfig** object supports two types of section: keyed and epoch. Keyed sections are identified by square brackets **[]** surrounding the section name while epoch sections are identified by angle brackets **< >** surrounding the section name. All section names must be unique and are case-sensitive. Leading and trailing whitespace is removed from the section name.

A keyed section is a collection of unique keys where each key is identified by a unique, case-sensitive string. The key is the string before the “equals” (=) sign with all leading and trailing whitespace removed. The value of the key is the string following the “equals” sign with leading and trailing whitespace removed.

An epoch section is a collection of keys identified by a timestamp. The timestamp is encoded as year-month-day hour:mins:secs. The keys need not be in chronological order within the file but they are properly ordered once read in by ONYX. The value of the key is the string following the “equals” sign with leading and trailing whitespace removed.

Interfaces Exposed

IOnyxConfig*

Return to Onyx COM Objects

OnyxContainer

COM object used to manipulate a single table within a file within the scientific database. The object exposes two interfaces: **IOnyxContainer** and **IOnyxCollection**. The first interface is primarily involved with record creation and file I/O while the latter interface is involved with iteration and sorting operations on the collection/array of records.

The **OnyxContainer** permits the user to choose a template for the type of container used to store the collection of records. These templates are based upon the standard collections available in the Standard Template Library: list, vector, set and multi-set. Most users reading files will find the vector collection very suitable for their needs. On the hand users creating new files will find the list collection most suitable as it easily grows in size.

Certain calibration information such as spacecraft attitude might be best suited to a set or multi-set collection as it is always sorted into ascending order when a new record in inserted. This allows specific records for specific instances in time to be quickly and efficiently located (Log(N) operation). See IOnyxCollection::Sort() for moe details about sorting requirements.

Interfaces Exposed

IOnyxCollection*
IOnyxContainer*

Return to Onyx COM Objects

OnyxDataBase

The **OnyxDataBase** COM object is the top level object in an ONYX database. By design each experiment/scientific database has its own specific **OnyxDatabase** object implementation. Each implementation initializes an aggregated ONYX **OnyxDatabase** object with definitions of all of the scientific “C” structures used in the database.

The **OnyxDatabase** object should be kept alive by the user for the entire duration of database usage. The **OnyxDatabase** object can only be created through a call to the COM API function **CoCreateInstance**. The user must call **CoInitialize** at least once in the program before calling **CoCreateInstance** and **CoUninitialize** after finishing with the database.

Interfaces Exposed

IOnyxDatabase*

Return to Onyx COM Objects

OnyxDataObject

A COM object that represents a single record within a scientific database table. Each **OnyxDataObject** allocates enough storage in memory to hold a single instance of the “C” structure associated with this object. The structure is allocated using **malloc** which guarantees proper memory alignment for any variable handled by the program.

The object exposes a single interface, **IOnyxDataObject*** that allows a user to access and manipulate the underlying “C” structure.

OnyxDataObject objects are normally accessed using the **IOnyxIterator*** interface of **OnyxIterator** objects created by **IOnyxCollection::CreateIterator**.

Interfaces Exposed

IOnyxDataObject*

Return to Onyx COM Objects

OnyxIterator

A COM object used to iterate over a collection of OnyxDataObject. The iterator object is created by **IOnyxCollection::CreateIterator** and is the only way to access the individual records of a **OnyxContainer** or section collection from **OnyxConfig**.

Interfaces Exposed

IOnyxIterator*

Return to Onyx COM Objects

Onyx Structures and Enumerations

enum VARENUM
enum ONYX_ACCESS_MODE
enum ONYX_COLLECTION_TYPE
ONYXFIELDDESC
ONYXCONTAINERDESC
ONYX_CONFIGURATION_KEY
ONYX_CONFIGURATION_EPOCHKEY
ONYX_TABLE_ENTRY

enum ONYX_ACCESS_MODE

Specifies file access modes for ONYX I/O routines

```
enum ONYX_ACCESS_MODE{
                                ONYX_READ,
                                ONYX_WRITE,
                                ONYX_APPEND
};
```

Entries

ONYX_READ

Open the file in Read access only. All table pointers are placed at the beginning of the first record.

ONYX_WRITE

Open the file in Write mode. Any existing file/table is erased if it already exists and a new empty file created. All table pointers are implicitly placed at the beginning of the each table.

ONYX_APPEND

Open the file in Append mode. If the file already exists it is opened otherwise a new file is created. All table pointers are placed beyond the last record in the table such that the next write will append to the end of the table.

Return to Onyx Structures and Enumerations

enum VARENUM

The VARENUM enumeration is used to tell ONYX what type of data is stored in each field of a “C” structure. VARENUM is based upon an enumeration of the same name found in the Windows Software Development Kit. This is particularly convenient when Windows-specific software is being developed.

For the most part ONYX follows the same definitions as used by the Windows implementation. Two exceptions are *VT_DATE* and *VT_BOOL*. ONYX uses Modified Julian Date (JD - 2400000.5) stored as a **double** while Windows uses a **double** based on another epoch. (1900 A.D. I think). ONYX uses **nxBOOL** as the associated data type for VT_BOOL while Windows uses **boolean**.

```
enum VARENUM{
                    VT_EMPTY            = 0,
                    VT_I2               = 2,
                    VT_I4               = 3,
                    VT_R4               = 4,
                    VT_R8               = 5,
                    VT_DATE             = 7,
                    VT_BOOL             = 11,
                    VT_I1               = 16,
                    VT_UI1              = 17,
                    VT_UI2              = 18,
                    VT_UI4              = 19,
                    VT_LPSTR            = 30
};
```

Entries

VT_EMPTY

Indicates no value. Not normally used by the user when defining “C” structures as translation to and from file formats is not supported. It is reserved for various error codes within the software. No associated “C” data type.

VT_I2

16 bit signed integer. Associated “C” data type is **nxSHORT**.

VT_I4

32 bit signed integer. Associated “C” data type is **nxLONG**.

VT_R4

32 bit floating point number. Associated “C” data type is **float**

VT_R8

64 bit floating point number. Associated “C” data type is **double**

VT_DATE

A “C” **double** used to store the modified Julian Date (Julian Date - 2400000.5). Associated “C” data type is **double**

VT_BOOL

Stores a boolean. Associated “C” data type is **nxBOOL**.

VT_I1
8 bit signed integer. Associated “C” data type is **nxCHAR**

VT_UI1
8 bit unsigned integer. Associated “C” data type is **nxBYTE**

VT_UI2
16 bit unsigned integer. Associated “C” data type is **nxWORD**

VT_UI4
32 bit unsigned integer. Associated “C” data type is **nxDWORD**

VT_LPSTR
Pointer to a null terminated string. Currently this is only used with the **IOnyxConfig*** interface which reads in elements which points to strings read from a file. Associated C data type is **[const] char ***. ONYX does not currently support reading and writing VT_LPSTR datatypes to and from HDF VDATA.

Return to Onyx Structures and Enumerations

enum ONYX_COLLECTION_TYPE

Used to tell ONYX what sort of collection is desired. All of the available collections are based upon the collections available in the ANSI Standard Template Library. ONYX provides four basic types of container and these are described below.

```
enum ONYX_COLLECTION_TYPE{
                              ONYX_DONT_CARE,
                              ONYX_LIST,
                              ONYX_VECTOR,
                              ONYX_SET,
                              ONYX_MULTISET
};
```

Entries

ONYX_DONT_CARE

Indicates that the caller has no preference and the software should use the default stored in the ONYXCONTAINERDESC.

ONYX_LIST

The collection is a linked list of elements. This is particularly useful when creating new tables from raw data as many records have to be repeatedly inserted at the the end of the collection.

ONYX_VECTOR

A collection closely related to a normal “array” of objects. It provides fast random access and allows for logarithmic sorting operations. This is the collection of choice when reading data records back from a file which do not normally need any special a-priori sorting.

ONYX_SET

A collection of uniquely keyed records. As far as I know STL implements this as a binary tree. This is the collection of choice when you can guarantee that the sort index is unique (non-unique values overwrite each other) and the data is well suited for a binary tree style algorithm.

ONYX_MULTISET

Similar to ONYX_SET except keys do not need to be unique.

Return to Onyx Structures and Enumerations

ONYXFIELDDESC

Describes each field of a “C” structure to ONYX. Usually implemented internally to an **OnyxDatabase** object. The structure provides a mapping from HDF byte packed arrays to “C” structure fields. The structure permits data to be stored in files at one resolution (*fieldtype*) and in memory at another (*datatype*). Typically this feature is used to store scientific data arrays as 32 bit **float** in files but process them in memory as 64 bit **double.**

```
typedef struct tagONYXFIELDDESC {
                        char*               fieldname;
                        enum VARENUM        fieldtype;
                        int                 flags;
                        long                fieldorder;
                        enum VARENUM        datatype;
                        int                 recordoffset;
} ONYXFIELDDESC;
```

Entries

fieldname
The name of the field as stored in the HDF VDATA.

fieldtype
The data type of the field stored in the file. If the *flags* field has bit ONYX_ISHOPTR set then this **ONYXFIELDDESC** field points to an array stored elsewhere in the file and *fieldtype* indicates the type of element stored in the array in file.

flags
Any special bit flags for this field. A value of 0 indicates no special flags. The only flag defined to-date is **ONYX_FLAG_ISHOPTR** (1). This means that this field is a pointer to a data array stored elsewhere in the file. If bit **ONYX_FLAG_ISHOPTR** is set then the corresponding field in the “C” structure must be IOnyxArray*. This association is implicitly assumed by ONYX and must be guaranteed by the user.

fieldorder
The number of elements stored in this field. This allows fixed length arrays to be stored in a corresponding “C” structure. Note that HDF limits the total size of a record to 32767 bytes and ONYX also reads in the entire table in one go. Hence it is not recommended to encode very large arrays using this method. Use the IOnyxArray* technique instead.

datatype
The data type of the field in the “C” structure. If the *flags* field has bit **ONYX_FLAG_ISHOPTR** set then this **ONYXFIELDDESC** field points to an array stored elsewhere in the file and *datatype* indicates the type of element stored in the array in memory. It is the database implementors responsibility to ensure that the *datatype* specified here maps to the corresponding variable type in the C structure. Inconsistency between the C structure variable types and the value of

datatype will lead to memory corruption problems. The descriptionof VARENUM indicates the mapping of each VARENUM to C variable.

recordoffset
The offset in bytes from the beginning of the C structure to this field

#define ONYX_FLAG_ISHOPTR 1

Return to Onyx Structures and Enumerations

ONYXCONTAINERDESC

This structure is used to define a “C” structure ONYX. It consists of a *classid*, which uniquely defines this “C” structure within the scientific database and an array of field descriptors, one for each field of the associated “C” structure.

The first field descriptor in the array *fields* is used as the default “sort” field by ONYX. This requirement is especially important for collections based upon ONYX_SET or ONYX_MULTISET as they almost always use the first field to determine how to insert data into the ONYX collection. ONYX does not support sorting of **IOnyxArray*** fields.

```
typedef struct tagONYXCONTAINERDESC {
                                HOCONT                     classid;
                                enum ONYX_COLLECTION_TYPE
                                                           defaultcollection;
                                int                        userrecordlength;
                                int                        numfields;
                                ONYXFIELDDESC*             fields;
} ONYCONTAINERDESC
```

Entries

classid

A number that uniquely identifies this **ONYXCONTAINERDESC** from any other within the scope of the experiment. Different experiments/database may use common numbers. The database implementor is responsible for allocating unique numbers. Failure to do so could cause potentially serious problems within ONYX as it will associate field offset and sizes with the wrong C structure.

defaultcollection

The default type for the collection to be used when a database implementor requests ONYX_DON’T_CARE

userrecordlength

The length of the associated C structure in bytes. This should be equivalent to the value returned by **sizeof(***my structure***) .**

numfields

The number of valid **ONYXFIELDDESC** entries in the array *fields*.

fields

A contiguous array of *numfields* ONYXFIELDDESC entries. Each entry describes on field in the C structure. It is assumed that the entire “C” structure is properly represented by this list although this is a responsibility of the database implementor. The ordering of the **ONYXFIELDDESC** entries does not have to be in the same order as the fields in the C structure.

Return to Onyx Structures and Enumerations

ONYX_CONFIGURATION_KEY

The ONYX_CONFIGURATION_KEY structure is used to represent one record of “keyed” information stored in an **OnyxConfig** section. This is the “C” structure of the **OnyxDataObject** objects stored in the collection returned from a call to **IOnyxConfig::FindSection.**

```
struct ONYX_CONFIGURATION_KEY{
                const char*             key;
                const char*             value;
};
```

Entries

key

The case-sensitive “key” string associated with this entry in the corresponding configuration section

value

The string value associated with this key. The OnyxConfig object is responsible for string memory allocation and deletion and the user should not attempt to overwrite these values.

#define CONT_ONYX_CONFIGURATION_KEY -1

Return to Onyx Structures and Enumerations

ONYX_CONFIGURATION_EPOCHKEY

The ONYX_CONFIGURATION_EPOCHKEY structure is used to represent one record of "epoch based keys" stored in an OnyxConfig section. This is the "C" structure of the OnyxDataObject objects stored in thecollection returned from a call to IOnyxConfig::FindSection**.**

```
struct ONYX_CONFIGURATION_EPOCHKEY{
                              double              mjd;
                              const char*         value;
};
```

Entries

mjd

The Modified Julian Date of this entry in the collection.

value

The string value associated with this key.

#define CONT_ONYX_CONFIGURATION_EPOCHKEY -2

Return to Onyx Structures and Enumerations

ONYX_TABLE_ENTRY

Structure used by **IOnyxDatabase::GetTablesInFile** to report details about structures store in an ONYX database file.

```
struct ONYX_TABLE_ENTRY{
                    const char*             name;
                    nxLONG                  classid;
};
```

Entries

name

Pointer to a null terminated string that is the name of the table in the file. The pointer is valid while the parent **IOnyxCollection*** is valid.

classid

The class id of the “C” structure of this table. This only has meaning in the context of the database object that created the file.

#define CONT_ONYX_TABLE_ENTRY -3

Return to Onyx Structures and Enumerations

Onyx COM Interfaces

This is the list of documented interfaces supported by ONYX. ONYX objects may implement other interfaces but they are either documented elsewhere or are not supported. Unsupported interfaces may be changed without notice in future releases and upgrades of ONYX and so their use is strongly discouraged.

IOnyxArray
IOnyxCollection
IOnyxConfig
IOnyxContainer
IOnyxDatabase
IOnyxIndex
IOnyxDataObject
IOnyxFileLocator
IOnyxIterator

IOnyxArray

Interface used to access arrays associated with OnyxArray objects. These arrays are normally fairly large volume entities. ONYX automatically creates **OnyxArray** objects for all fields in the underlying C structure that have the **ONYX_FLAG_ISHOPTR** bit set in the associated ONYXFIELDDESC *flags*. ONYX inserts the **IOnyxArray*** interface into the C structure. This is the only mechanism for creating **OnyxArray** objects.

Methods

GetArray	Fetch a pointer to the start of the array
ReleaseData	Release the memory associated with the actual array.
SetNewSize	Change the size of the array
SizeOfElement	Get the size of one array element in bytes.

Return to Onyx interfaces

IOnyxArray::GetArray

Fetches details about the data array controlled by this interface. The data array is specified as a 2-D array with *numcolumns* as the number of elements along one row, i.e. contiguous in memory and *numrows* as the number of rows in one data array.

HRESULT GetArray(

byte**	*dataptr*,
int*	*numcolumns*,
int*	*numrows*,
enum VARENUM*	*datatype*

);

Parameters

dataptr

Returns a pointer to the start of the array. The array is allocated by **malloc** as a single contiguous chunk. This guarantees proper memory alignment for any data type. May return NULL. This pointer should be typecast by the user to the appropriate datatype as specified by *datatype.*

numcolumns

Returns the number of elements along one row of the data array. These elements are contiguous in memory and represent the fastest changing element. Hence in C they represent one row but in Fortran they represent one column. If *numcolumns* is NULL then this value is not returned.

numrows

Returns the number of rows in this data array. This represents the slowest changing element in memory. If *numrows* is NULL then this value is not returned

datatype

Returns the data type of each element. All elements in the array are of the same type. If *datatype* is NULL then this value is not returned.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes

Return to IOnyxArray

IOnyxArray::ReleaseData

Release the memory allocated for data storage. This does not release the object only the data storage. It is important to release data for high volume data as it will simply consume virtual memory until it runs out.

HRESULT ReleaseData( );

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxArray

IOnyxArray::SetNewSize

Changes the size of the array. This destroys any previous contents ad invalidates any pointers returned by **GetArray**. It is not possible to change the data type of an array by design.

```
HRESULT SetNewSize(
                int             numcolumns,
                int             numrows,
                byte**          dataptr
);
```

Parameters

numcolumns
: The number of columns in the array. This is the fastest changing index in the array.

numrows
: The number of rows in the array.

dataptr
: Returns a pointer to the newly allocated memory.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxArray

IOnyxArray::SizeofElement

Retrieves the ize of an array element in bytes

```
HRESULT SizeofElement(
                    int*                    nbytes
);
```

Parameters

nbytes
    Returns the size of one element in bytes.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxArray

IOnyxCollection

Interface used to manipulate the collection of records stored in the **OnyxContainer** object or similar objects

Methods

BinaryFind	Search a sorted collection using a binary tree, logarithmic algorithm.
Clone	Make a copy of this collection
CopyFrom	Copies/appends a collection from another collection
CreateIterator	Create an iterator object to iterate this collection
CreateRecord	Create a **OnyxDataObject** compatible with collection
Erase	Erase the entire collection
Find	Find a specific member in the collection
FindEntriesBounding	Find entries bounding a range in a sorted collection
GetDescriptor	Get the descriptor of associated C structure
GetSortIndex	Get the index of the field used for sort operations
Insert	Inserts an element into the collection
Reserve	Reserves space for N elements in the collection
Size	Returns the number of elements in the collection
SetSortIndex	Define which field is used for sorting
Sort	Sort the collection

Return to Onyx interfaces

IOnyxCollection::Clone

Makes a separate copy of this collection. All fields of each **OnyxDataObject** object in the collection is copied to new memory in the new collection. The only exception is **IOnyxArray*** fields. In this case only the **IOnyxArray*** pointer is copied and an extra call to **IOnyxArray::AddRef()** is made. In this way only pointers to the high volume data fields are copied rather than the data itself.

A useful application is to change the type of storage collection from one mode to another, e.g. ONYX_LIST to ONYX_VECTOR, or ONYX_VECTOR to ONYX_SET etc. etc.

```
HRESULT Clone (
                IOnyxCollection**               newcollection,
                enum ONYX_COLLECTION_TYPE       usercoltype
                BOOL                            create_empty
);
```

Parameters

newcollection

Returns a pointer to the **IOnyxCollection*** interface of the newly created collection. The user must call **Release()** when finished with this collection. If the procedure fails then this pointer may return NULL.

usercoltype

Specifies the type of collection that should be created. The choices are based upon the ANSI Standard Template Library containers. This value should be ONYX_LIST, ONYX_VECTOR, ONYX_SET or ONYX_MULTISET.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::CopyFrom

Copies all of the elements from the collection represented by *other* and inserts them into this collection. This does not physically copy the memory of each element but only copies the **IOnyxDataObject** pointer. The procedure calls **IOnyxDataObject::AddRef()** for each copied element. In this case changes made to elements in the original collection are also reflected in the copies stored in this collection.

```
HRESULT CopyFrom(
                IOnyxCollection*        other
);
```

Parameters

other
Interface of the other collection from which elements are copied.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::CreateIterator

Creates an object which can be used to iterate over this collection of objects. Note that inserting and erasing elements from the collection while the iterator is active may have strange effects. It is not recommended to change the number of elements in the collection while the iterator is active. The user must call the iterator's **Release()** method when finished with the iterator.

```
HRESULT CreateIterator(
                                IOnyxIterator**            iterator
);
```

Parameters

iterator

Returns the **IOnyxIterator*** interface of the iterator object. If the procedure fails this may return NULL.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::CreateRecord

Creates a new, blank **OnyxDataObject** object that is compatible with this collection. Note that the object is not automatically inserted into this collection and the user must call **IOnyxCollection::Insert()** to do this. The user should call **IOnyxDataObject::Release()** when finished with the object.

```
HRESULT CreateRecord(
                    IOnyxDataObject**      dataobject
                    );
```

Parameters

dataobject

Returns the **IOnyxDataObject*** interface of the **OnyxDataObject** object. May return NULL if the procesure fails.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Erase

Erases all of the elements in this collection. The **Release()** member function is called for each element stored in the collection. After return from this function the collection is empty.

HRESULT Erase();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Find

Used to find a specific record in the collection. The algorithm is a linear search for ONYX_VECTOR and ONYX_LIST collections. It searches for a matching record using just the current sort field. A typical application is for searching specific keys in a collection belonging to a section from a configuration file. In this case the user creates a compatible **IOnyxDataObject**, fetches its C structure, fills out the sort field and calls **IOnyxCollection::Find** to locate the object in the collection with the same sort field.

IOnyxCollection::Find will allways find the first instance in the collection that matches the requirement. To find subsequent entries a user would have to iterate through the container. **IOnyxCollection::Find** uses the definition of equality as (!(a < b) && !(b < a)) as the **IOnyxDataObject** interface only provides a “Less Than” operator.

```
HRESULT Find(
                         IOnyxDataObject*       value,
                         IOnyxDataObject**      record
);
```

Parameters

value

The **IOnyxDataObject*** interface of the “value” to be found. This object is provided by the user. The “value” object only needs to have a valid value in the sort field as all other fields are ignored.

record

Returns the **IOnyxDataObject*** interface of the object found in the collection. If no mathc is found then *record* returns NULL.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::BinaryFind

Uses a binary search algorithm to find a specific record in the collection. The algorithm matches records using the current sort field. If the collection is base dupon ONYX_VECTOR, ONYX_SET or ONYX_MULTISET then it search the collection using abinary search algorithm. The user must ensure that the collection is properly sorted otherwise the algorithm will not work correctly. For ONYX_LIST collections the algorithm simply performs a linear search.

```
HRESULT BinaryFind(
                IOnyxDataObject*        value,
                IOnyxDataObject**       record
);
```

Parameters

value

The **IOnyxDataObject*** interface of the “value” to be found. This object is provided by the user. The “value” object only needs to have a valid value in the sort field as all other fields are ignored.

record

Returns the **IOnyxDataObject*** interface of the object found in the collection. If no mathc is found then *record* returns NULL.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::FindEntriesBounding

Used to find bounding records in a collection which is implicitly assumed to be sorted. This routine is typically used to find the records immediately before and after a specific time. For example it can be used to “look-up” the attitude of a spacecraft for a given instrument measurement. Uses the STL function **find_lower** and **find_upper**. The function uses the default sort field to determine which records are less than *value* and which records are greater.

```
HRESULT FindEntriesBounding(
                                IOnyxDataObject*        value,
                                IOnyxIterator**         lower,
                                IOnyxIterator**         upper,
                                boolean                 extendarray
);
```

Parameters

value

The **IOnyxDataObject*** interface of the “value” to be found. This object is provided by the user. The “value” object only needs to have a valid value in the sort field as all other fields are ignored.
The user should be very careful not to pass in the **IOnyxDataObject** interface of an object which does not have the same C structure as this collection, e.g. passing a instrument measurement to a spacecraft attitude collection. This usually means using a small interface function which creates a blank record with the collection object’s **IOnyxContainer::CreateRecord** method and filling in the sort field with the appropriate value extracted from the incompatible **IOnyxDataObject*** object.

lower

Returns a pointer to the **IOnyxIterator*** interface of an iterator object that points to the record in the collection immediatley before or equal to the location specified by the *value* object. If *extendarray* is TRUE then this function will return a pointer to the first record in the collection even if the *value* object is before the first record. If *extendarray* is FALSE then if the *value* object is before the first record in the collection then the procedure returns the value NULL in *lower*. If the function fails then *lower* will return NULL.

upper

Returns a pointer to the **IOnyxIterator*** interface of an iterator object that points to the record in the collection immediatley after the location specified by the *value* object. If *extendarray* is TRUE then this function will return a pointer to the last record in the collection even if the *value* object is after the last record. If *extendarray* is FALSE then if the *value* object is after the last record in the collection then the procedure returns the value NULL in *upper*. If the function fails then *upper* will return NULL.

extendarray

If TRUE then always return a sensible value in *lower* and *upper* even if *value* is outside the bounds of the collection. If FALSE then return NULL in *lower* or *upper* if *value* is outside the bounds of the collection.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::GetDescriptor

Returns the descriptor that defines the underlying C structure. For informational purposes only.

HRESULT GetDescriptor(
const ONYXCONTAINERDESC** *desc*
);

Parameters

desc
Returns a pointer to the descriptor describing the underlying C structure. User should be aware that the pointer is only guaranteed valid while the container exists.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::GetSortIndex

Retrieves the field index that is currently used for sort operations. The value returned can be used to index the array returned by **GetDescriptor**.

```
HRESULT GetSortIndex(
                        int*                      sortindex
                        );
```

Parameters

sortindex
Returns the index of the field used in sort operations.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Insert

Inserts the object *newelement* into this collection. Only the pointer to the object is stored in the collection. The physical C structure is not copied but instead only the **IOnyxDataObject*** interface pointer is copied. In addition the element's **AddRef()** member is called. The element's **Release()** method is called when the element is removed from the collection.

```
HRESULT Insert (
                              IOnyxDataObject*        newelement
);
```

Parameters

newelement

The **IOnyxDataObject*** interface of the lement to be inserted into this collection. Insertion depends upon the type of collection. ONYX_LIST and ONYX_VECTOR will insert at the end while ONYX_SET and ONYX_MULTISET will insert using the current sort field to work out its location.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Reserve

Reserves an additional *nelem* elements for this collection. This effects the efficiency of collections which are allocated as ONYX_VECTOR. For all other collections it has no effect. This does not actually change the size of the collection but merely provides an opportunity for more efficient memory allocation for anticipated future insertions.

```
HRESULT Reserve(
                    int                    nelem
);
```

Parameters

nelem
    Number of extra elements to reserve

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::SetSortIndex

Specifies whch field in the underlying C structure should be used for "sort" operations on the collection. This field is implicitly used for ONYX_SET and ONYX_MULTISET collections. The sort field for these two collections can only be changed if the collection is empty. The field is explicitly used for methods **Sort** and **FindEntriesBounding**. The index refers to the zero-based index of the **ONYXFIELDDESC** entry in the collection's **ONYXCONTAINERDESC::fields** member. The default value for the sort index is 0. The designer's of ONYXCONTAINERDESC structures should try to ensure that the first field is a sensible sort field.

Sorting operations will fail on C structure fields based on **IOnyxArray***.

```
HRESULT SetSortIndex(
                    int                    fieldindex
);
```

Parameters

fieldindex
: The zero based index of the field in **ONYXCONTAINERDESC::fields** to be used for subsequent sort operations.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Size

Fetches the number of elements in the collection.

```
HRESULT Size(
                int*                    numrecords
);
```

Parameters

numrecords
Returns the number of records in this collection.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxCollection::Sort

Sorts the collection using the current sort field. See **SetSortIndex** to change the field used for sorting. It is not possible sort ONYX_SET and ONYX_MULTISET collections as they are already sorted! However this function will succeed for these collections.

HRESULT Sort();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxCollection

IOnyxConfig

The **IOnyxConfig** interface provides a mechanism for accessing, low-volume, text based configuration information. The configuration object is created by **IOnyxDatabase::LoadConfiguration** which associates and/or loads a text based configuration file with this object. The current **IOnyxConfig** paradigm assumes the text based file consists of multiple, unique, sections where each section is an array of text based keys. The format is very similar but not identical to the initialization files (*.ini) used on Windows operating systems.

One feature of the **IOnyxConfig** paradigm is that it allows epoch based sections. In these sections the key is a timestamp (year-month day hr:min:secs) and the value string is the value associated with that time. This permits easy maintenance and portability of low volume, epoch driven, configuration tables associated with the database.

The current implementation identifies sections as key driven when the section name in the configuration file is surrounded by square brackets **[ ]**. Epoch driven sections are identified by angle brackets **< >** surrounding the section name in the configuration file. It is permissible to freely mix epoch and key driven sections within one configuration file.

Methods

ConfigSectionKeyValue	retrieve value of a key in a section
CreateKeyRecord	Create a blank record compatible with a specific section
EpochSectionKeyValue	Retrieve value of previous epoch key in a section
FindSection	Find the keys in a specific section
GetFilename	Get the name of the associated configuration file.

Return to Onyx interfaces

IOnyxConfig::ConfigSectionKeyValue

Retrieves the string *value* associated with a specific key in a specific section in this configuration.

```
HRESULT ConfigSectionKeyValue(
                const char *        section,
                const char *        keystring,
                int                 maxchar,
                char*               value
);
```

Parameters

section
Name of section to search. String is null-terminated and case sensitive.

keystring
The name of the key to search for. String is null-terminated and case-sensitive.

maxchar
The maximum number of characters that can be written to *value*.

value
Points to a string buffer of *maxchar* characters. The string value of the matched key is placed into this buffer. Up to *maxchar* characters are written. Algorithm always writes a NULL character '\0' into the last available space to ensure the string is NULL terminated. If no matching key is found then string returns as a blank string (“”);

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxConfig

IOnyxConfig::CreateKeyRecord

Creates a blank **OnyxDataObject** record compatible with the specified section

```
HRESULT CreateKeyRecord(
                                IOnyxDataObject**       blankrecord,
                                const char *            sectionname
                                );
```

Parameters

blankrecord

Returns the **IOnyxDataObject*** interface of a blank record compatible with the requested section specified by *sectionname*. If the section does not exist then return NULL. The blank record must be deleted by a call to **Release().** If the section is "epoch-based" then the blank record is based upon the C structure ONYX_CONFIGURATION_EPOCHKEY**.** If it is not epoch based then it is based upon ONYX_CONFIGURATION_KEY.

sectionname

The name of the requested section. The name is case-sensitive, including whitespace and non-printable characters.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxConfig

IOnyxConfig::EpochSectionKeyValue

Retrieves the string value associated with the epoch driven key in the specified section that occurs before or at the time specified by *mjd.*

```
HRESULT EpochSectionKeyValue(
                                const char *            section,
                                double                  mjd,
                                int                     maxchar,
                                char*                   value
);
```

Parameters

section

Name of section to search. String is null-terminated and case sensitive.

mjd

The time for which information is requested. Algorithm will search for the latest key that occurs before or at the time specified by *mjd*.

maxchar

The maximum number of characters that can be written to *value*.

value

Points to a string buffer of *maxchar* characters. The string value of the matched key is placed into this buffer. Up to *maxchar* characters are written. Algorithm always writes a NULL character ‘\0’ into the last available space to ensure the string is NULL terminated. If no matching key is found then string returns as a blank string (“”);

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxConfig

IOnyxConfig::FindSection

Retrieves the collection of section keys associated with the section, *sectionname*, in the configuration file. The *isepochbased* flag is an additional part of the moniker provided by the user which must match the properties of the requested section.

```
HRESULT FindSection(
                                const char*             sectionname,
                                boolean                 isepochbased,
                                IOnyxCollection**       collection
);
```

Parameters

sectionname
    The name of the section requested by the user. The name is case-sensitive.
isepochbased
    Must be TRUE if the associated section is epoch based. Must be FALSE if the associated container if key based.
collection
    The **IOnyxCollection*** interface of the collection of keys associated with the requested sections. If the section is not found the this value returns NULL. The user must call the **IOnyxCollection::Release()** member function when finished with the interface. All **IOnyxConfig** collections are based upon ONYX_SET and hence are intrinsically sorted unique keys.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxConfig

IOnyxConfig::GetFilename

Retrieves the name of the associated configuration filename.

HRESULT GetFilename(
    int *nchar*,
    char* *filename*
);

Parameters

nchar
    The maximum number of characters pointed to by buffer *filename*.

filename
    A pointer to a buffer of *nchar* characters tha receives the name of the configuration filename. The string is truncated to fit into the specified buffer if necessary but it it always NULL terminated.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxConfig

IOnyxContainer

Initial entry-point interface for processing **OnyxContainer** objects. The interface provides create new, blank records, iterators for the collection of records and to perform I/O functions to and form file.

Methods

Attach	Attach (open) to a specified file
Detach	Detach from the specified file.
Insert	Insert a data object into this container
Load	Load this container from the attached file
Name	Get the name of this container
PartialLoad	Loads part of a table into a container
TableSize	Returns number of records in a given table.
Write	Write an OnyxDataObject to the attached file

Return to Onyx interfaces

IOnyxContainer::Attach

Attaches the current container to the file specified by *filename*. The code will automatically open the file in the specified mode and if successful will open the table associated with this container. If the mode is ONYX_WRITE then a new file is created. It both permitted and expected that several **OnyxContainer** objects will open the same file simultaneously. ONYX keeps an internal list of active files and if that file currently exists then it is returned rather than creating a new instance. The only requirement in these cases is that the file must be opened in the same *mode.*

```
HRESULT Attach(
                                const char*                 filename,
                                enum  ONYX_ACCESS_MODE      mode
);
```

Parameters

filename

The filename of the file to be opened. This is the full path to the file. The string is zero terminated string and must not be NULL.

mode

The mode in which to open the file. The supported modes are ONYX_READ, ONYX_WRITE and ONYX_APPEND. ONYX_WRITE will erase any existing file and create a new copy while ONYX_READ and ONYX_APPEND will open existing files. ONYX_APPEND will create a new file if the file does not already exist.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::Detach

Detaches from a currently attached file. Closes any connection this object has with the attached file. This is important as the **IOnyxContainer** interface often caches records as this makes the underlying HDF VDATA less fragmented. The **OnyxContainer** object will automatically detach when it is destroyed although the user should not rely on this mechanism.

HRESULT Detach();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::Insert

Inserts a new **OnyxDataObject** member into the container. The insertion point depends upon the type of **OnyxContainer**. Containers based upon ONYX_SET or ONYX_MULTISET are place objects into an sorted list while ONYX_LIST and ONYX_VECTOR place the new record at the end. See **IOnyxCollection** for more details about sorting **OnyxDataObject** objects

```
HRESULT  Insert(
                              IOnyxDataObject*         newobject
);
```

Parameters

newobject

The new **OnyxDataObject** to be inserted into the container. The container places an **AddRef()** onto the object and calls its **Release()** member when the object is removed from the collection. It is important to realize that changes made to the **OnyxDataObject** by the user are also reflected in the object stored in the container. This is particularly important if container is based upon ONYX_SET or ONYX_MULTISET as the collection's sort order is potentially corrupted if the sorted field is changed after insertion into the container.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::Load

Loads all of the records from the appropriate table in the currently attached file. All **IOnyxArray*** fields are loaded as indirect pointers to the data stored in the hidden file tables. It is implicitly assumed that all of the records can be loaded into virtual memory.

HRESULT Load();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::Name

Returns the name of this container.

```
HRESULT  Name(
                        const char**            name
);
```

Parameters

name

Returns a pointer to the name of this container. The returned string is null terminated. User should be aware that the pointer is only valid while the container exists.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::PartialLoad

Identical to method Load except it only loads part of a table. This is useful for situation where the caller only requires a few records from the container. The caller may use method **TableSize** to determine the number of records in the table.

```
HRESULT  PartialLoad(
                    int                 startrecord,
                    int                 numrecords
);
```

Parameters

startrecord
: The zero based record at which to start reading records from the table stored in the file.

numrecords
: The number of records to read from the table stored in the file.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxContainer::TableSize

Returns the total number of records in the table that the container is attached to.

```
HRESULT  TableSize (
    int*                    numrecords
);
```

Parameters

numrecords

Returns the total number of records in the table that the container is attached to.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

IOnyxContainer::Write

Writes the record associated with the object *objct* to the attached file. The file must be opened in mode ONYX_APPEND or ONYX_WRITE. The action of writing the object implicitly inserts a copy of the object into the container. Usually the container does not physically write the record to disk until the container is detached from the file but since the container has its own copy the user is free to modify the C structure associated with the **OnyxDataObject** after this call.

The call will call the objects method **ReleaseData**. This was provided since many users were forgetting to release memory and running out of virtual memory.

HRESULT Write**(**
IOnyxDataObject* *objct*
);

Parameters

objct
The **IOnyxDataObject*** interface of the **OnyxDataObject** that will be written to file.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxContainer

IOnyxDatabase

The **IOnyxDatabase** interface is the top most interface for accessing ONYX scientific databases. The most common method is **CreateContainer()** which Users can call to traverse further down the database hierarchy..

Methods

AddContainerDefn	Add a new C struct definition to the database
CreateContainer	Create a new OnyxContainer object
CreateIndexObject	Create a new OnyxDatabaseIndex object.
LoadConfiguration	Load a text-based configuration file
SetDefaultLogger	Define where errors are logged
SetMaxNumberOfOpenFiles	Set upper limit to maximum number of open files
GetTablesInFile	Get list of tables in this file
SetDatabaseClsid	Set the GUID of the ONYX creator.

Return to Onyx interfaces

IOnyxDatabase::AddContainerDefn

Adds a new “C” structure definition to the database. This in primarily intended for use by the COM database object specific to each experiment. This object during initilization loads the core component with a list of C structure definitions. However the method is available for use by general users.

```
HRESULT AddContainerDefn(
                    const ONYXCONTAINERDESC*   contdescriptor
);
```

Parameters

contdescriptor

A pointer to the container descriptor. It is implicilty assumed by ONYX that the pointer is valid for the entire duration of the database as ONYX does not make its own copies of these structures. Normally these definitions are static variables declared in the COM object specific to each experiment. Must not be NULL.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

See Also

ONYXCONTAINERDESC
Return to IOnyxDataBase

IOnyxDataBase::CreateContainer

Creates a new, empty container that can hold the "C" structures associated with *classid*. The interface requested in *riid* is returned in *interfce*. The user should call **Release()** once it is finished with the container.

```
HRESULT CreateContainer (
                    const char*                  name,
                    HOCONT                       classid,
                    REFIID                       riid,
                    void**                       interfce,
                    enum  ONYX_COLLECTION_TYPE          colltype
);
```

Parameters

name
Null terminated string specifying the name of the container. Must not be NULL.

classid
Numeric identifier specifying the format of the underlying "C" structure. The numeric identifiers are specific to each experiment and should be contained in that documentation.

riid
Identifier of the requested interface.

interfce
The requested interface. This will be NULL if the function does not return S_OK

colltype
Type of container requested by user. This allows the user to choose between a linked list, vector array, unique key binary tree, or multi-key binary tree.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

See Also

ONYX_COLLECTION_TYPE, HOCONT

Return to IOnyxDataBase

IOnyxDatabase::CreateIndexObject

Create a new **OnyxDatabaseIndex** object which may be used for indexing this database. Note that the object must be initialized by the user. The user must call the object's **Release()** when finished.

HRESULT **CreateIndexObject(**

```
                    IOnyxDatabaseIndex**        newindex
);
```

Parameters

newindex

Returns the **IOnyxDatabaseIndex*** interface of the new object. The user must call **Release()** when finished with this interface. The new object must be properly initialized by the caller before it an be properly used. If the creation fails then *newindex* will return a NULL pointer.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

See Also

ONYX_COLLECTION_TYPE, HOCONT

Return to IOnyxDataBase

IOnyxDatabase::GetTablesInFile

Retrieves the CLSID of the database that created the specified file and a collection of **IOnyxDataObject** that store “C” structure **ONYX_TABLE_ENTRY**. Each entry stores the name of a table in the file and the class id of the associated “C” structure. The class id of the “C” structuire only has meaning in the context of the creator database identified by *onyxdbase*. This function is intended to support generic browsing applications.

The function will only return information about tables which have “C” structure class id’s greater than 0. Zero and negative values are reserved for internal usage and hidden tables.

```
HRESULT GetTablesInFile(
                        const char *          filename,
                        CLSID*                onyxdbase,
                        IOnyxCollection**     table_collection
);
```

Parameters

filename
  The name of the file to be queried

onyxdbase
  Returns the CLSID of the ONYX database that created this file. If no entry is found this will return the value **CLSID_OnyxCoreDatabase.**

table_collection
  Returns a collection of **OnyxDataObject**. Each object in the collection is based upon the “C” structure **ONYX_TABLE_ENTRY**. There is one **OnyxDataObject** for each table found in the file. The user should dispose of the collection with **IOnyxCollection::Release()**. The strings referenced in the **ONYX_TABLE_ENTRY** structures are only valid for the lifetime of the collection.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataBase

IOnyxDataBase::LoadConfiguration

Creates a IOnyxConfig object and loads it with the specified text-based, configuration file. Configuration files are normally used to save small volume, system wide, database information. Processing options, search directories and other bits of informations are ideal candidates for this interface. The configuration files can also handle relatively small volumes of epoch dependent configuration.

```
HRESULT LoadConfiguration(
                    const char *            filename,
                    boolean                 cachecontents,
                    IOnyxConfig**           config
);
```

Parameters

filename
The full path-name of the configuration file.

cachecontents
If TRUE then the specified file is loaded and cached in memory. If not TRUE then the file is not loaded until user actually requests configuration information.

config
Returns the OnyxConfiguration object used to store configuration information from the file. The user should call **Release()** when he is finished with this configuration object. If the procesure fails then NULL is returned.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

See Also

InxLog

Return to IOnyxDataBase

IOnyxDatabase::SetDatabaseClsid

Initializes the core ONYX database with the CLSID of a specific ONYX scientific database. This CLSID information is stored in each database file. This function is not intended for use by the general user but is provided for the scientific database implementor.

HRESULT SetDatabaseClsid**(**
REFCLSID *onyxdbase*
);

Parameters

onyxdbase
The CLSID of this particular sciuentific database. The value is retrieved from the the file using **IOnyxDatabase::GetTablesInFile**.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataBase

IOnyxDataBase::SetDefaultLogger

Loads a new error logging object into the database. All subsequent errors encountered during ONYX processing are logged to the new logger object. When first created an ONYX database has no default logger.

HRESULT SetDefaultLogger**(**
IUnknown* *loggerobject*
);

Parameters

loggerobject
A pointer to a COM object supporting the **InxLog** interface. The database will lock the object until it is either replaced with anew logger or the database object is destroyed.

*** NOTE *** Beta version assumes *loggerobject* is a pointer to a C++ class nxLog* as I have not yet implemented the InxLog interface. This has minimal impact and will be remedied for final release software.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

See Also

InxLog

Return to IOnyxDataBase

IOnyxDataBase::SetMaxNumberOfOpenFiles

```
HRESULT SetMaxNumberOfOpenFiles(
                         int                    nfiles
);
```

Parameters

nfiles

The maximum number of files that ONYX can physically hold open at any instant. Normally ONYX will assume it can hold open up to MAX_FILE files where MAX_FILE is defined within the HDF software. Note ONYX is not limited in functionality by this parameter. If ONYX runs out of valid HDF file handles it will internnaly close the least recently used files and automatically reopen them when required. Generally the user will not need access to this function. The value specified should not exceed the value of MAX_FILE defined in hdf include files.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataBase

IOnyxIndex

Interface used to index Onyx database files. The indexing paradigm supports both direct an dindirect indexing. Direct indexing is where the entire database table is stored in one file and can be loaded into a single **IOnyxCollection.** It is anticipated that a direct indexing would be applied to tables which occupy less than a few MegaBytes of system memory.

Indirect indexing is where the the Onyx database table is distributed over many separate files (or granules). requested epoch driven records are spread over multiple files where the start and end time of each file is controlled by a master index. The activity is similar to a hash lookup table. For example the ODIN spacecraft position and velocity are stored in one file per orbit. A master index holds the start time of each orbit. The logical operation is to look up the orbit which corresponds to a specific time and then look up the record in the file that corresponds to that specific time.

The default **IOnyxIndex** tracks the two records that straddle a given instant in time. i.e. the start and end record. However the class can be configured so it reads *N* records before and *M* records after the record that starts an epoch; the default of two records is *N=0* and *M=1*. The class handles indirect indexing so they are continuous across master index boundaries, there is a weak constraint that it will only look at records from the previous/following entry as defined by the start times in the master index.

It is possible to create a hierarchy so a direct or indirect indexed array is the master index for another indirect indexed array. Most applications will use a simple direct indexed array as the master index of an indirect indexed array. Multi-threading is not supported.

By default it is assumed that records are stored in file in proper ascending order. The code will, by default, use the first VT_DATE field found in the container's ONYXCONTAINERDESC fields. These settings may be overidden by a call to **SetGuardAndSort**

Methods

SetGuardAndSort
DefineIndexParams
ConfigureAsIndirectIndex
Lookup
GetCollection

Return to Onyx interfaces

IOnyxIndex::DefineIndexParams

Allocates internal storage information necessary for the **OnyxIndex** object to locate appropriate records for the user. Once created, the internal allocation persists until the object is destroyed through a call to **Release()**. This method must be called once and only once before a call to **CheckEpoch()**. This call only allocates internal structures but does not actually load any files until required. Any calls to **ConfigureAsIndirectIndex** and **SetGuardAndSort** must occur before calling this method.

HRESULT DefineIndexParams **(**

const char*	*name,*
HOCONT	*cid*,
boolean	*isconfiguration,*
IOnyxFileLocator*	*locator*

);

Parameters

name

If the *isconfiguration* flag is FALSE then *name* is name of the container that will be read in from an ONYX data file. If *isconfigration* is TRUE then *name* is the name of an epoch based section in an text-based *ini* file.

cid

If the *isconfiguration* flag is FALSE then *cid* is the class identifier of the Onyx container that will be read in from file. If *isconfigration* is TRUE then *cid* is ignored.

isconfiguration

If FALSE then the epoch driven tables are derived from binary tables which are accessed using the the **IOnyxContainer** interface. If TRUE then the epoch driven tables are accessed using the **IOnyxConfig** interface and is typically used to access text based *ini* files.

locator

The **IOnyxFileLocator*** interface of an object used to locate files. This object will be kept alive for the entire duration of the given **IOnyxDatabaseIndex** object.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDatabaseIndex

IOnyxDatabaseIndex::ConfigureAsIndirectIndex

Instructs the code to treat this object as an indirect index. This requires that a master index exists. The indirect index requires the times in the master index to define how to locate the file that contains the specific record for the period of interest. This method must be called before a call to method **AllocateIndex**.

HRESULT **ConfigureAsIndirectIndex (**

```
                              IOnyxDatabaseIndex*   masterindex,
                              int                   sizeof_masterindex
);
```

Parameters

masterindex
Is the interface of the master index object. The master index will be held in memory via a call to **AddRef()** until this object is destroyed. If *masterindex* is NULL then this object is a direct index.

sizeof_masterindex
This is the size in bytes of the OnyxDataObject records in the master index. If the master index is based upon the **IOnyxConfig** interface then *sizeof_masterindex* must be ???.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDatabaseIndex

IOnyxDatabaseIndex::CheckEpoch

Fetches the appropriate database records corresponding to the time specified in *mjd.* If successful the records will match those defined in a call to method **AllocateIndex()**.

To be successful the method must fetch all of the required records surrounding a given instant in time. For example if we have configured the index so it fetches 1 point before the associated record and 1 after (total 3 points as we have to include the actual record as well, 3=1+1+1) then the function only succeeds if it can fetch all 3 points.

If the index is indirect then the method will use the master index to automatically *"join"* neighbouring files. Note that the algorithm will always read the previous file as there are normally no guarantees that records within a given indexing file are synchronized to the records in the master index.

```
HRESULT CheckEpoch(
                    double          mjd,
                    int             recordsize,
                    void***         records,
                    int*            numrecords,
                    boolean*        haschanged
);
```

Parameters

mjd
: The Universal Time epressed as a modified Julian Date for which index records are required.

recordsize
: The size in bytes of the OnyxDataObject records in this database index. If the index is based upon the **IOnyxConfig** interface then *recordsize* must be ???.

records
: Returns the array of records surrounding the requested time. The parameter returns a pointer to a contguous array of void*. Each element of the void* array really points to the "C" structure of each record. The pointers are guaranteed to remain valid until the next call to this object's **CheckEpoch()** method. The number of elements in the array is returned in **numrecords.*

numrecords
: Returns the number of void* elements in the **records* array.

haschanged
: Returns TRUE if the records have changed since the last call. Otherwise it returns FALSE which indicates the records are the same as the last call.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDatabaseIndex

IOnyxDatabaseIndex::GetCollection

Fetches the **IOnyxCollection*** interface of the current collection associated with this index. If the index is direct then the collection is the entire table. If the index is indirect then the collection is the indirect component currently loaded.

```
HRESULT GetCollection(
                IOnyxCollection**       collection,
);
```

Parameters

collection

Returns the collection currently used by the index. The value will return NULL if the index has not yet cached any records. This will happen if the index has not ben used for a data lookup or if the indirect collection is not available for whatever reason (e.g. not on-line).

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDatabaseIndex

IOnyxDatabaseIndex::SetGuardAndSort

Allows the user to define various parameters used during indexing: the number of records retrieved around a given instant, the index of the record elements used to sort records and a flag requesting that **IOnyxConfig** indices extend there first and last records to -/+ infinity.

By default the code assumes database records are stored in file in a sorted format. If this is not the case then this function must be called to force an explicit sorting of records (which incurs a fairly expensive computational overhead especially on indirect indices).

```
HRESULT SetGuardAndSort (
                         int          numbefore,
                         int          numafter,
                         int          sortindex,
                         boolean      sortrecords,
                         boolean      extendbounds
);
```

Parameters

numbefore
The number of records to be retrieved before the record associated with a given instant in time. The record associated with a given instant in time is the closest record occurring either at or before to the specified instant in time. The default value, which is also the minimum, is to retrieve 0 records before the associated record.

numafter
The number of records to be retrieved after the record associated with a given instant of time. The default value, which is also the minimum, is 1: the routine will always retrieve as a minimum the two records straddling a given instant in time.

sortindex
Specifies the index of the time field in the container's ONYXCONTAINERDESC used to sort records. Note that the *sortrecords* flag should be set to TRUE if the user cannot guarantee that the records as stored in file are in the properly sorted order. A value of -1, which is the default, instructs the code to use the first VT_DATE field found in the container's ONYXCONTAINERDESC.

sortrecords
If TRUE forces the code to explicitly sort records using the index specified by *sortindex.* This operation can be produce heavy I/O for indirect indices as they must not only sort the current file but also the previous and following files. It is assumed that most configuration files will already be stored in proper chronological order and that this flag provides a convenience on rarely used indices.

extendbounds

This allows indices based upon **IOnyxConfig** to extend the first and last records to infinity. This can be useful on occasion. The option is not available for indices based upon **IOnyxContainer.**

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDatabaseIndex

IOnyxDataObject

Interface used to interact with **OnyxDataObject** objects. The methods provide methods to manipulate the associated C structure including copying, moving and comparing.

Methods

Clone	Make a new **OnyxDataObject** with identical information
CopyCStruct	Copy the underlying C structure to a user buffer
Dereference	Dereference all of the **OnyxArray** objects in the C structure
GetCStructPtr	Get a pointer to the underlying C structure
IsKindOf	Returns the number identifying the underlying C structure
KeyIndexIsLessThan	See if this C structure is "*less than*" another C structure
ReleaseData	**ReleaseData** for all **OnyxArray** objects in C structure

Return to Onyx interfaces

IOnyxDataObject::IsKindOf

Retrieves a number that uniquely identifies this C structure within the scope of this scientific database.

HRESULT IsKindOf**(**
HOCONT* *typeident*
);

Parameters

typeident
Returns the number that uniquely identifies the underlying C structure.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataObject

IOnyxDataObject::CopyCStruct

Copies the underlying C structure to or from a user supplied buffer. All fields are physically copied from the source to the destination. Fields based on **IOnyxArray*** copy the pointer from the destination to the source and make an additional call to **IOnyxArray::AddRef().** Hence all **OnyxArray** objects are shared between C structure copies.

HRESULT CopyCStruct**(**

int	*nbytes*,
void*	*cstruct*,
boolean	*tothisobject*

);

Parameters

nbytes
: The size in bytes of the C structure, *cstruct*, passed by the user. This size must match the size of this object's C structure otherwise the function will fail.

cstruct
: A pointer to the callers C structure. Although it is a pointer to void the user must ensure the C structure has proper memory alignment. This must not be NULL.

tothisobject
: If FALSE then copy from this object to *cstruct*, otherwise copy from *cstruct* to this object.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes

Return to IOnyxDataObject

IOnyxDataObject::GetCStructPtr

Retrieves a pointer to the underlying “C” structure for use by the caller. This is significantly more efficient than **CopyCStruct** as only pointers are transferred.

```
HRESULT GetCStructPtr(
                    int          nbytes,
                    void**       cstructptr
);
```

Parameters

nbytes
: The size of the “C” structure expected by the caller. This number, provided by the caller, must match the size of the the underlying structure stored in the **OnyxDataObject**.

cstructptr
: Retrieves a pointer to the underlying C structure which the caller must type-cast to the appropriate structure.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataObject

IOnyxDataObject::Dereference

Releases all of the data array memory associated with **IOnyxArray*** fields stored in the underlying C structure. All **IOnyxArray*** fields are identified by the **ONYX_FLAG_ISHOPTR** bit in the *flags* field in the **ONYXFIELDDESC** structure.

HRESULT Dereference()

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes

Return to IOnyxDataObject

IOnyxDataObject::Clone

Creates a new **OnyxDataObject** which is a copy of this object. Algorithm creates the new **OnyxDataObject** and then invokes CopyCStruct. The new object is returned in *copy*. Note that **CopyCStruct** only copies **IOnyxArray*** pointers but shares the data arrays that they point to.

```
HRESULT Clone(
                    IOnyxDataObject**        copy
);
```

Parameters

copy

Returns a pointer to the newly created **OnyxDataObject**. May return NULL if the function fails. The user must call **Release()** on the new object when finished with the object.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataObject

IOnyxDataObject::ReleaseData

Releases all of the memory associated wih all **IOnyxArray*** fields stored in the underlying “C” structure. Failure to call this function can result in excessive demands on system memory as more new objects are created. In turn this may cause serious degradation in performance as the virtual memory starts to swap to disk.

HRESULT ReleaseData ();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxDataObject

IOnyxDataObject::KeyIndexIsLessThan

This function is used to compare this **OnyxDataObject** to another. It is used in all sorting operations and is implicitly used in **OnyxContainer** objects based upon ONYX_SET or ONYX_MULTISET.

The implementation only compares one field of the underlying structure, which is specified by *fieldindex*. This is the "key" field. ONYX can compare all of the supported VARENUM datatypes but cannot compare **IOnyxArray*** fields

```
HRESULT KeyIndexIsLessThan(
                    int                   fieldindex,
                    IOnyxDataObject*      other
);
```

Parameters

Returns

Returns S_OK if this **OnyxDataObject** object is "less than" the *other* **OnyxDataObject**. Returns S_FALSE if greater than or equal. May return E_FAIL and other COM error codes.

Return to IOnyxDataObject

IOnyxFileLocator

An interface typically used by **IOnyxDatabaseIndex** interfaces to locate files associated with a master index. The interface encapsulates all of the details necessary to locate files for a given experiment on a given operating system. No implementation is provided within the core ONYX libraries as the details are too specific for each different experiment.

Typical **OnyxFileLocator** objects are closely related to associated **OnyxDatabaseIndex** objects and normally part of the experiments database design.

Methods

LocateFile	Finds a file that is consistent with a given C structure. Typically used when looking up indirect indices.

Return to Onyx interfaces

IOnyxFileLocator::LocateFile

This method takes a “C” structure from the caller, who is normally an **IOnyxDatabaseIndex** object, decodes the “C” structure and locates the file associated with the “C” structure. The full path to the file is returned to the caller.

The OSIRIS instrument uses this technique to locate the orbit duration files. The **OnyxDatabaseIndex** object is configured to use the orbit definition table as the master index and the **OnyxFileLocator** objects are configured to decode the orbit definition entries, which includes the orbit number, and then locate the corresponding file. The details of searching directories and other operating system dependent activities are kept internal to the implementation of the **OnyxFileLocator** object.

HRESULT **LocateFile(**

void*	*masterrecord*,
int	*recordsize*,
int	*offset*,
char*	*buffer*,
int	*maxbuflen*

);

Parameters

masterrecord

A pointer to the master record “C” structure. This structure, normally acquired from an **OnyxDatabaseIndex** object, will contain all of the information necessary to locate the requested file. The parameter will be NULL when the **OnyxDatabaseIndex** object is a direct index, this is the proper behaviour. It should not be NULL when the **OnyxDatabaseIndex** object is an indirect index.

recordsize

The size in bytes of the *masterrecord*. This value is provided so the method may verify it has received as a structure of the correct size. I.E. it provides a quick check to make sure everything is sensible.

offset

This value may be used to locate files that logically come before or after the file indicated by *masterrecord.* A value of 0 returns the file indicated by *masterrecord*. A value of -1 returns the file before and +1 returns the file after. There is no intrinsic limit on the size of offset. hegative numbers get files before *masterrecord* and positive numbers get files after *masterrecord*.

buffer

Pointer to a user supplied buffer of *maxbuflen* characters (including trailing null terminator). The method will write the full pathname of the file to this buffer. It is suggested that the user supplied buffer be at least MAX_PATH+1 characters long. The pointer may be NULL in which case no data are written.

maxbuflen

The maximum number of characters including the null terminator that may be written to *buffer*. The value may be 0 or less in which case no data are written. A value of 1 will result in just the null terminator being returned!

Returns

S_OK if succesful and the file was found. Returns S_FALSE if the function worked but the file was not found. All other errors E_FAIL and other COM error codes.

Return to IOnyxFileLocator

IOnyxIterator

An object created by interface **IOnyxCollection***. Its sole purpose is give a user access to records controlled by the parent **IOnyxCollection*** interface. It also provides a few methods for precise insertion and deletion of records by the user.

Methods

Eof	Indicates we are at the end of the collection
Erase	Erase and remove record at current location
First	Moves to the first element in the collection
Insert	Inserts a record at current location
Last	Moves to the last element in the collection
MoveTo	Moves to element *n* in the collection
Next	Points to the next record in the collection.
Previous	Moves to the previous element in the collection
Value	Retrieves the record at current location.

Return to Onyx interfaces

IOnyxIterator::Eof

Indicates whether the current iterator is pointed beyond the end of the collection. This function is normally used to terminate an iteration over the collection.

HRESULT Eof();

Returns

S_OK if at end of collection. S_FALSE if not at end. Returns E_FAIL and other COM error codes if an error exists.

Return to IOnyxIterator

IOnyxIterator::Erase

Erases the record at the current location. This process involves calling the record's **IOnyxDataObject::Release** member and removing the **IOnyxDataObject*** pointer from the collection.

HRESULT Erase();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::First

Moves to the first element in the collection

HRESULT First();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::Insert

Inserts a record. If the collection is based upon ONYX_LIST or ONYX_SET then the insertion will occur before the current location. ONYX_SET and ONYX_MULTISET will insert the record in the proper sorted location.

HRESULT Insert(
IOnyxDataObject* *newobject*
);

Parameters

newobject
The new record to insert into the associated collection. The *newobject* must be compatible with the collection. The collection will automatically call **AddRef** on the new object to ensure its lifetime is consistent with the collection's lifetime.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::Last

Moves to the last element in the collection

HRESULT Last();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::MoveTo

Moves to the *n*'th element in the collection. Very efficient for collections based upon ONYX_VECTOR as these allow randon access. All other implementations for other collections start at the beginning and manually step forward *n* times.

HRESULT MoveTo(
int *n*
);

Parameters

n

The new, zero-based, position in the array. If new position is out of collection bounds then no action is taken.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::Next

Step to the next record in the collection. ONYX_VECTOR and ONYX_LIST collections are traversed in a linear fashion while ONYX_SET and ONYX_MULTISET collections are traversed in a sorted order.

HRESULT Next();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::Previous

Moves to the previous element in the collection

HRESULT Previous();

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator

IOnyxIterator::Value

Retries the **OnyxDataObject** record at the current location.

HRESULT Value**(**
IOnyxDataObject** *dataobject*
);

Parameters

dataobject
Returns the **IOnyxDataObject*** interface of the **OnyxDataObject** at the current location pointed to by this iterator. The caller must call **Release** on the **IOnyxDataObject*** interface when finished with the record. May return NULL if the pointer is invalid or other error occurs.

Returns

S_OK if succesful otherwise E_FAIL and other COM error codes.

Return to IOnyxIterator