Software through Pictures®
Core

Release 7.1

Object Management System

Software through Pictures Core Object Management System

Release 7.1

Part No. 10-MN100/ST7100-01298/001

December 1998

Aonix reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Trademarks

Aonix and the Aonix logo are trademarks of Aonix. ObjectAda is a trademark of Aonix. Software through Pictures is a registered trademark of Aonix. All rights reserved.

HP, HP-UX, and SoftBench are trademarks of Hewlett-Packard Inc. Sun and Solaris are registered trademarks of Sun Microsystems, Inc. SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc. Open Software Foundation, OSF, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. X Window System is a trademark of X Consortium, Inc. Windows NT is a trademark and Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and other countries. Adobe, Acrobat, the Acrobat logo, and PostScript are trademarks of Adobe Systems, Inc. Sybase and the Sybase logo are registered trademarks of Sybase, Inc. Adaptive Server, Backup Server, Client-Library, DB-Library, Open Client, PC Net Library, SQL Server, SQL Server Manager, SQL Server Monitor, Sybase Central, SyBooks, System 10, and System 11 are trademarks of Sybase, Inc.

World Headquarters
595 Market Street, 12th Floor
San Francisco, CA 94105
Phone: (800) 97-AONIX
Fax: (415) 543-0145
E-mail: info@aonix.com
http://www.aonix.com

Table of Contents

Chapter 2 Understanding the Persistent Data Model

Index

Preface

Software through Pictures (StP) is a family of multi-user integrated environments that supports the software development process, as well as maintenance and re-engineering of existing systems.

This manual is part of a complete StP documentation set of "Core" manuals, which includes: *Fundamentals of StP, Customizing StP, Query and Reporting System, Object Management System, StP Guide to Sybase Repositories*, and *StP Administration*. Basic information about the StP user interface is documented in *Fundamentals of StP*.

Object Management System describes the StP Object Management System (OMS), a set of standards and functions that supports each StP product. StP products consist of editors that generate information to the repository. The OMS provides the consistent, generic system structure for creating, updating, and deleting objects in the repository.

The OMS also provides capabilities for extending a particular application's view of the repository.

Intended Audience

The *Object Management System* manual is intended for StP users and independent software vendors (ISVs) who want to use the OMS to:

- Call the Application Program Interface (API) from a C program
- Manipulate data in the repository
- Extract data from the repository using the Query and Reporting System (QRS)

- Extend an application's view of the repository
- Integrate an application with StP

The information in this manual assumes that you are familiar with:

- Fundamentals of Software through Pictures, including using diagram editors, the Object Annotation Editor, and annotation templates
- Your operating system directory and file structures
- Fundamentals of C programming
- Working with abstract data types

Refer to the appropriate documentation if you have any questions on these topics.

Typographical Conventions

This manual uses the following typographical conventions:

Table 1: Typographical Conventions

Convention	Meaning
Palatino bold	Identifies commands**.**
`Courier`	Indicates system output and programming code.
`Courier bold`	Indicates information that must be typed exactly as shown, such as command syntax.
italics	Indicates pathnames, filenames, and ToolInfo variable names.
<angle brackets>	Surround variable information whose exact value you must supply.
[square brackets]	Surround optional information.

Contacting Aonix

You can contact Aonix using any of the following methods.

Reader Comments

Aonix welcomes your comments about its documentation. If you have any suggestions for improving *Object Management System*, you can send email to docs@aonix.com.

Technical Support

If you need to contact Aonix Technical Support, you can do so by using the following email aliases:

Table 2: Technical Support Email Aliases

Country	Email Alias
Canada	support@aonix.com
France	customer@aonix.fr
Germany	stp-support@aonix.de
United Kingdom	stp-support@aonix.co.uk
United States	support@aonix.com

Users in other countries should contact their StP distributor.

Websites

You can visit us at the following websites:

Table 3: Aonix Websites

Country	Website URL
Canada	http://www.aonix.com
France	http://www.aonix.fr
Germany	http://www.aonix.de
United Kingdom	http://www.aonix.co.uk
United States	http://www.aonix.com

Related Reading

Object Management System is part of a set of Software through Pictures documentation. For more information about OMS and related subjects, refer to the sources listed in Table 4.

Table 4: Further Reading

For Information About	Refer To
Installing StP	*StP Installation Guide*
Using the StP Desktop	*Fundamentals of StP*
Using standard StP editors and tools	*Fundamentals of StP*
Printing diagrams, tables, and reports	*Fundamentals of StP, Query and Reporting System*
Internal components of StP and how to customize an StP installation	*Customizing StP*
Using StP QRL Scripts	*Query and Reporting System*

Table 4: Further Reading (Continued)

For Information About	Refer To
Interacting directly with the StP storage manager Sybase Adaptive Server and SQL Server	*StP Guide to Sybase Repositories*
Managing an StP installation	*StP Administration*
Specific StP products information	The documentation provided with your StP product
Managing the database	The documentation provided with your relational database management system and *StP Administration*
A summary of changes in the current software release and last minute information that was not included in the manuals	Release Notes
Windows NT commands	Windows NT documentation

1 Introduction

This chapter introduces concepts and terms that are essential to understanding the Software through Pictures (StP) Object Management System (OMS), a set of standards and functions that defines the interface between application programs and the StP repository.

Topics covered are as follows:

Overview of StP

Software through Pictures (StP) comprises various client-server, multi-user products that share a common architecture built around a central repository. StP supports requirements analysis, information modeling, architectural and detailed program design, and a wide range of development activities.

Each StP product is comprised of tools, including:

- Graphical editors for drawing various types of diagrams
- Table editors for entering and formatting system-related data
- An annotation editor for extending information in diagrams

Each tool is an application that accepts and generates information. Together, these applications are concurrent processes that provide shared

access to the central StP repository in a client-server environment. The StP repository stores information "persistently," that is, the information outlives the process that created it.

To ensure that applications have a high degree of independence and that the tasks that use persistent data are straightforward and consistent, StP uses the Object Management System (OMS). The OMS provides services for creating, updating, and deleting objects in the StP repository efficiently and consistently regardless of how the data is stored. The OMS also provides services for creating new object types.

What Is the Object Management System?

The Object Management System (OMS) is a set of standards and functions that defines the interface between application programs and the Software through Pictures repository. The Object Management System comprises:

- The Persistent Data Model (PDM)
- The OMS Query Language
- The Application Programming Interface (API)
- Object Type Extensions

Persistent Data Model

StP repository data is "persistent data" and is stored according to the Persistent Data Model. This model provides a simple scheme for organizing and identifying objects in the StP repository while maintaining complete application independence. For details, see Chapter 2, "Understanding the Persistent Data Model."

OMS Query Language

The OMS provides a comprehensive query language that enables client programs to search the StP repository for persistent CASE objects. These

queries can be used in conjunction with various StP tools and applications. For details, see Chapter 3, “OMS Query Language.”

Application Program Interface

The OMS provides a high-level Application Program Interface (API), a library of functions with well-defined behavior that can be linked with C programs for extracting data from the StP repository. For more information, see Chapter 4, “Using the Application Program Interface.” Chapter 5, “Application Program Interface Types and Functions,” provides a reference section on each function.

Object Type Extensions

The OMS can extend the semantics of the data stored in the StP repository by adding new object types. For more information about defining application types and attributes, see *Customizing StP*.

How StP Stores Information

The models that a user creates with the StP editors are saved both in:

- ASCII system files, each of which represents an entire diagram, table, or set of annotations
- A database storage area called a repository, containing consolidated information about individual objects (rather than whole diagrams or tables)

Diagrams and tables must be syntactically correct for StP to store their contents in the repository. However, incorrect or incomplete diagrams and tables can be stored in the ASCII files. StP editors always load diagrams and tables from the ASCII files, rather than the repository. This allows a user to save incomplete work to be finished later, without jeopardizing the validity of the data in the repository.

System ASCII Files

Each ASCII file contains:

- An ASCII representation of a viewable diagram, table, or set of annotations
- Information about how a diagram, table, or set of annotations is connected to other diagrams and / or tables

If necessary, a damaged or missing repository can be completely restored or updated from the system files (for details, see *StP Administration*).

The StP Repository

The StP repository is a central store of project information supported by an underlying relational database. Because StP stores information in the repository according to the Persistent Data Model, it is completely independent of the particular application and storage manager in use.

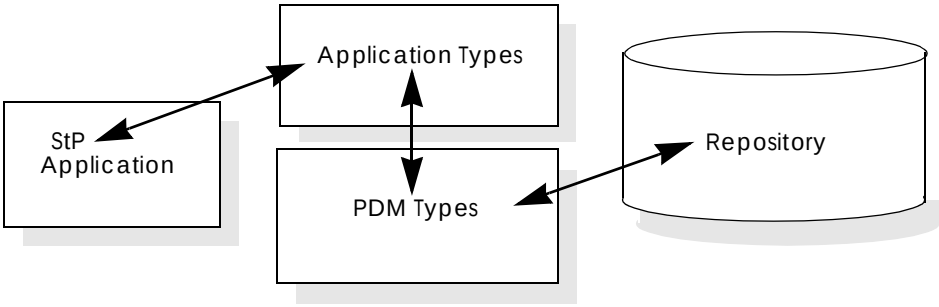
StP uses a commercial relational database, such as Sybase Adaptive Server or SQL Server, or Microsoft Jet, to implement the StP repository. Users and applications gain access to the StP repository through the Query Language and Reporting System (QRS), the OMS Query Language, or the API, rather than through any interface provided by the specific storage manager, such as embedded SQL or 4GL.

Theoretically, you can change StP data, such as diagrams and tables, through the underlying database's interface, but it is not a good idea to do so because you can inadvertently corrupt data. Always use StP itself to change StP data. You can, however, use the underlying database's utilities (such as Sybase's Sybase Central or SQL Server Manager, or Microsoft Jet's MS Access) to perform administrative tasks on the repository.

Applications and the Persistent Data Model

The link between applications and the Persistent Data Model (PDM) is provided by the Application Types (*app.types*) file. This is an extensible file that maps types recognized by applications (application types) to types recognized by the StP repository (PDM types).

Figure 1: The Application View of the StP Repository



Once information is in the repository, it is available to various applications for a wide range of activities, such as generating code and creating documents.

Terms

This section provides definitions for some of the terms used throughout this manual.

annotated object type	A category of data type in the Persistent Data Model. An annotated object is one that can have an annotation.
annotation	Explanatory notes about an object in the repository.
application	A program that interacts with the repository to accomplish a task.
Application Program Interface (API)	A library of functions that application programs can use to extract information from the repository.
attribute	A characteristic that is associated with a persistent data type.

auxiliary data type	A data type that helps to manage the persistent data types in the repository.
CASE type	A category in the Persistent Data Model that includes all persistent data types.
context (cntx) type	A data type in the Persistent Data Model that represents additional or attached information about a link.
cntx_ ref type	A data type in the Persistent Data Model that represents a reference to a context (cntx) in an StP file.
file type	A data type in the Persistent Data Model that represents a file known to StP applications.
file_hist type	A data type in the Persistent Data Model that represents an object that stores historical information about events pertaining to a file.
file_lock type	A data type in the Persistent Data Model that represents a file lock.
identifying attribute	An attribute that helps to identify an object as unique in the repository. Usually more than one attribute contributes to this unique identification.
instantiate	Create a specific instance of a general type.
item type	A data type in the Persistent Data Model that represents the item element of an object's annotation.
link type	A data type in the Persistent Data Model that represents a relationship or association between two node objects in an StP file.
link_ref type	A data type in the Persistent Data Model that represents a reference to a link (or symbol of a link) from a file.
node type	A data type in the Persistent Data Model that represents a unit in a file, exclusive of links or contexts.
node_ref type	A data type in the Persistent Data Model that represents a reference to a node from a file.

notation	A set of shapes with specific semantics within the context of an StP diagram editor.
note type	A data type in the Persistent Data Model that represents the note element of an object's annotation.
object	A discrete unit of information about the system design as it is stored in the repository. Each object has a set of attributes and can participate in relationships with certain other objects. These characteristics are determined by the Persistent Data Model.
persistent data	Any data that outlives the process that created it, as opposed to data that exists only in memory.
Persistent Data Model (PDM)	A paradigm that relates various abstract data types, their attributes and relationships to each other.
persistent data (PDM) type	An abstract data type that belongs to the persistent data model.
project	The directory name where systems are stored. "system" is a subdirectory of "project."
reference object	An object that stores information about a reference with semantic content in an StP file.
referential integrity	Constraints placed on relationships between objects in the repository that ensure that objects exist for each logical pointer to them from other objects.
relationship	The connection between objects
repository	The database where objects are stored.
Software Engineering (SE) object type	A category of data types in the Persistent Data Model that represents concepts used to build CASE models.
symbol	A drawing in a diagram to signify a concept or object.
system	The directory name where the editor files and project database are located. "system" is a subdirectory of "project."
viewpoint type	A data type in the Persistent Data Model that represents a diagram symbol that is the focal point for a certain type of information about a node object.

2 Understanding the Persistent Data Model

This chapter describes the objects that comprise the Persistent Data Model.

Topics covered are as follows:

What Is the Persistent Data Model?

The Persistent Data Model (PDM) is a conceptual scheme that defines the attributes and relationships of objects in the StP repository in terms of abstract, persistent data types. An abstract data type consists of a data structure and associated functions. Persistent data is any data that is

stored in the StP repository or files and can outlive the process that produced it.

The Persistent Data Model provides a uniform way for all applications to use the data in the StP repository. This uniformity makes working with or customizing applications a straightforward and consistent task.

Users and applications have access to the information in the StP repository using the:

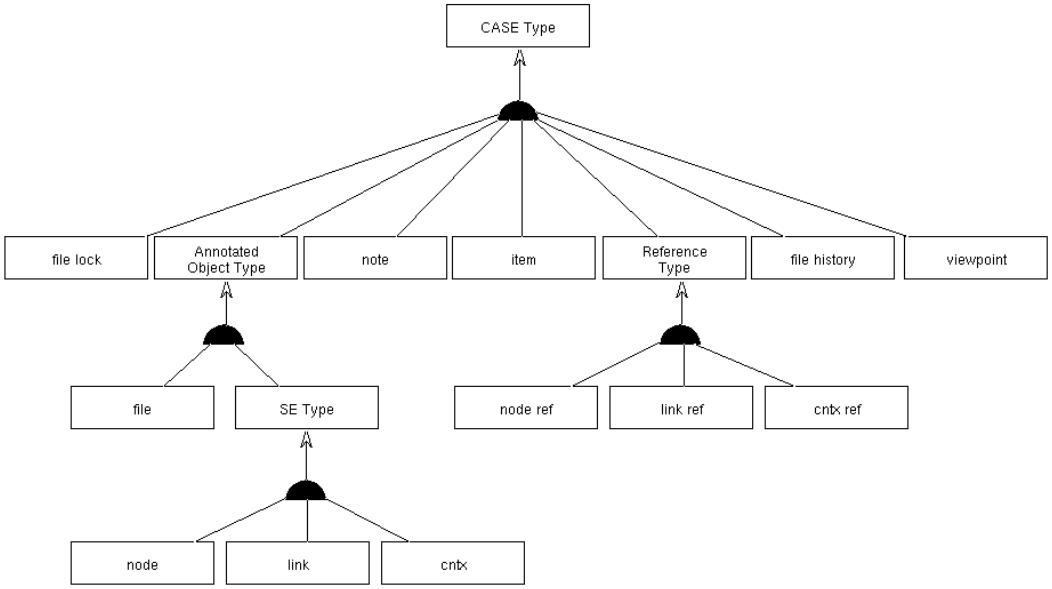
- OMS Query Language, described in Chapter 3, "OMS Query Language"
- Application Program Interface (API) a library of procedures described in Chapter 4, "Using the Application Program Interface"
- Query and Reporting System (QRS), described in *Query and Reporting System*

To construct OMS queries or use the API, you need a basic understanding of PDM types and their attributes.

Persistent Data Model Relationships

The abstract data types in the PDM (PDM types) belong to an exclusive subtype relationship, shown in Figure 1.

Figure 1: Persistent Data Model



In an exclusive subtype relationship, each entity (type) can be a supertype and / or subtype of other entities. Rectangles represent types. The exclusive subtype relationships are represented by the exclusive subtype symbol (the black semi-circle) and the arcs joining the types.

The CASE type is a supertype of all the types in the PDM model shown in Figure 1; all the types in the PDM are subtypes of the CASE type. Certain types, such as note and item, are direct subtypes of the CASE supertype. Other types are subtypes of subtypes. For example, file is a subtype of Annotated Object Type. In this relationship, Annotated Object Type is a supertype of file (and SE Type).

Supertypes (such as the Annotated Object Type) are categories of types and cannot be instances in the StP repository. In addition to the CASE type and Annotated Object Type, supertypes include Reference Type and Software Engineering type (SE Type). The subtypes represent types that can be instantiated and stored in the StP repository. Instances of types in the StP repository are called “objects.”

The following sections describe each PDM type. For information about PDM type attributes, see "Attributes of CASE Types" on page 2-6. Also, the OMS contains some abstract data types that cannot be stored in the StP repository and are therefore not included in the PDM. These abstract data types are described under "Auxiliary Types" on page 4-6.

CASE Types

There are three subtypes of the CASE supertype that are also supertypes:

- Annotated Object Type
- Reference Type
- Software Engineering type (SE Type)

Table 1 provides a description of each CASE supertype and subtype in the PDM. Each type is described fully in a subsequent section of this chapter.

Table 1: Overview of CASE Supertypes and Subtypes

Type	Description	Subtype of
Annotated Object Type	A supertype category that includes all object types that can have an annotation. See "Annotated Object Supertype" on page 2-13.	CASE Type
cntx	Corresponds to information that is attached to a link. See "Cntx Subtype" on page 2-19.	SE Type
cntx_ref	Information about a specific cntx symbol in a file. See "Cntx_ref Subtype" on page 2-32.	Reference Type
file	Corresponds to a file managed by the host machine's operating system containing StP diagrams, tables, annotations, documents, or source code. See "File Subtype" on page 2-13.	Annotated Object Type

Table 1: Overview of CASE Supertypes and Subtypes (Continued)

Type	Description	Subtype of
file_hist	Corresponds to historical information about events that happen to files. See "File_hist Subtype" on page 2-26.	CASE Type
file_lock	Corresponds to the lock placed by a user or an application or a file. See "File_lock Subtype" on page 2-24.	CASE Type
item	Corresponds to the name/value pair that is part of an annotation. See "Item Subtype" on page 2-22.	CASE Type
link	Corresponds to relationships or associations between nodes. See "Link Subtype" on page 2-18.	SE Type
link_ref	Corresponds to information about a specific link symbol in a file. See "Link_ref Subtype" on page 2-31.	Reference Type
node	Corresponds to an element in a model, such as a process, module, or external. See "Node Subtype" on page 2-17.	SE Type
node_ref	Corresponds to information about a specific node symbol in a file. See "Node_ref Subtype" on page 2-30.	Reference Type
note	Corresponds to the basic unit of an annotation. A note contains a set of items and a free-form description of arbitrary length. See "Note Subtype" on page 2-21.	CASE Type
Reference type	A supertype category that includes all references to an object from symbols in files. See "Reference Supertype" on page 2-28.	CASE Type
SE type	A supertype category that includes objects that are elements of Software Engineering. See "Software Engineering Supertype" on page 2-15.	Annotated Object Type

Table 1: Overview of CASE Supertypes and Subtypes (Continued)

Type	Description	Subtype of
viewpoint	Corresponds to the symbol that is designated as the view of an object. See "Viewpoint Subtype" on page 2-27.	CASE Type

Attributes of CASE Types

Each CASE type has a set of attributes that all objects of that type use. The OMS supports querying the StP repository for objects with specific attribute values. For example, the following CASE types, note and item, each have their own id, obj_id, and type attributes:

Table 2: Attributes Used by the Note and Item CASE Types

note subtype	item subtype
id	id
obj_id	obj_id
type	type
name	value
file_id	note_id
desc	

The OMS uses attributes to define referential integrity constraints between objects. For example, item objects belong to notes; hence, the note_id attribute under item links the item subtype to the note subtype. The file_id attribute under note links the note subtype to the file subtype. For more information on how OMS objects use referential integrity constraints, see "Referential Integrity" on page 2-33.

Application types (described in "Applications and the Persistent Data Model" on page 1-4) have all the attributes of the associated CASE type. For

example, a process application type has all the attributes of a node, which is its associated CASE type.

This section describes the attributes used in the PDM, independent of CASE type. Subsequent sections of this chapter describe CASE types and list their attributes.

Each attribute has a standard data type, as noted in Table 3 on page 2-7. These data types are described in detail in "Data Types of Attributes" on page 2-11.

Identifying Attributes

Each persistent CASE type has attributes that uniquely identify it. No other object in the StP repository can have the same values for all of these attributes. An application program can retrieve a particular object from the StP repository using the object's id. If the application program does not know the object's id, it can use the object's identifying attributes.

CASE Type Attributes Table

Table 3 describes the PDM's attributes and their data types. Several of the attributes apply to more than one CASE type. For information about individual CASE types, see the appropriate section following the table.

Table 3: Attributes of CASE Types

Attribute	Data Type	Description
accessors	character string	List of users that have access to a locked file.
annot_file_id	oms_object_id_tp	Uniquely identifies the file object that contains the annotation for an object.
appid	character string	String that uniquely identifies the reference to an SE object in a file.
cntx_id	oms_object_id_tp	Uniquely identifies a cntx.

Table 3: Attributes of CASE Types (Continued)

Attribute	Data Type	Description
desc	text	Free form textual description of arbitrary length.
destroyers	character string	List of users that can remove a file lock.
duration	oms_time_tp	Maximum time a file lock should be in effect.
fname	character string	File path of the object.
file_id	oms_object_id_tp	Uniquely identifies a relevant file.
from_node_id	oms_object_id_tp	Uniquely identifies a link's source node.
from_node_ref_id	oms_object_id_tp	Uniquely identifies the node reference that is the source of a link ref.
hostname	character string	String that uniquely identifies the machine where a relevant event occurred.
id	oms_object_id_tp	Uniquely identifies an object.
link_id	oms_object_id_tp	Uniquely identifies the link to which a cntx belongs.
link_ref_id	oms_object_id_tp	Uniquely identifies the link ref to which a related cntx ref is attached.
lmdfy_id	oms_object_id_tp	Uniquely identifies the most current change to a file.
lsync_id	oms_object_id_tp	Uniquely identifies the most current repository update for a file.
lview_id	oms_object_id_tp	Uniquely identifies the most current viewing of a file.

Table 3: Attributes of CASE Types (Continued)

Attribute	Data Type	Description
name	character string	Object name.
node_id	oms_object_id_tp	Uniquely identifies a relevant node object.
node_ref_id	oms_object_id_tp	Uniquely identifies the reference to a node.
note_id	oms_object_id_tp	Uniquely identifies the note to which an item belongs.
obj_id	oms_object_id_tp	Uniquely identifies the annotated object to which a note or item belongs.
pid	integer	Identifies the process that established a file lock.
rev	character string	Current revision of a file.
scope_node_id	oms_object_id_tp	Uniquely identifies the parent object of a nested object.
scope_node_ref_id	oms_object_id_tp	Uniquely identifies the node ref that serves as the scope of an object.
sig	character string	Uniquely identifies an object signature; application-supplied string used to differentiate two objects with otherwise equivalent identifying attributes.
svalue	character string	Value of an item. Same as the value attribute, except stored in all lower case letters.
time	oms_time_tp	Date and time a relevant event took place.
to_node_id	oms_object_id_tp	Uniquely identifies a link's destination node.

Table 3: Attributes of CASE Types (Continued)

Attribute	Data Type	Description
to_node_ref_id	oms_object_id_tp	Uniquely identifies the node reference that is the destination of an arc.
type	oms_app_type_tp	Object or event type; indicates application-level type information of an object or event.
user	character string	Login id of the user who performed a relevant action.
value	character string	Value of an item.
xcoord	integer	X coordinate in an application's coordinate system.
ycoord	integer	Y coordinate in an application's coordinate system.

Data Types of Attributes

An attribute's data type can be one of the following:

- Integer
- Character string
- Text
- Special OMS types

Integer Type

The PDM supports integers that correspond to the int data type in the C programming language.

Character String and Text Types

A character string is a varying length sequence of characters. Limits are imposed on:

- Character strings—maximum 220 characters
- Text—can be very long, depends on space allocated for repository

OMS Data Types

Some CASE types have attribute data types that are defined by the OMS:

- Object Id attribute type (oms_object_id_tp)
- Application Type attribute type (oms_app_type_tp)
- Time Type (oms_time_tp)
- Application Id attribute type (oms_appid_tp)

These OMS data types have special semantics. For a brief description of each OMS data type, see the sections which follow.

The Object Id Attribute: oms_object_id_tp

All persistent CASE types have an Object Id (oms_object_id_tp) attribute. The value of any Object Id attribute is assigned when the object is created. It is unique within a particular repository.

Applications should never examine this value directly; they should use the functions provided by the OMS (oms_object_id_equal) to determine whether one id is equal to another. An id is unique within only one repository, so applications that manipulate objects from more than one repository need to ensure that the objects remain separated.

The App Type Attribute: oms_app_type_tp

Many of the CASE types have an Application Type (oms_app_type_tp) attribute associated with them. This attribute gives a specific application type to nodes, links, files, cntxs, and other persistent CASE objects. For example, the value of the App Type attribute distinguishes nodes that represent processes from nodes that represent externals, cspecs, and other application types.

This attribute contains an integer value from a list of legal values specified in the application types (*app.types*) file. For more details, see *Customizing StP*.

The OMS does not enforce any semantic constraints on this attribute other than checking that the value is legal. While the OMS enforces the rule that a link must connect to two nodes, it does not enforce any rules concerning the types of links and nodes that can be connected. These semantics are considered part of the application program's domain.

The Time Type Attribute: oms_time_tp

The OMS stores oms_time_tp attributes in the database as 32-bit signed integers representing the number of seconds since January 1, 1970, Greenwich Mean Time.

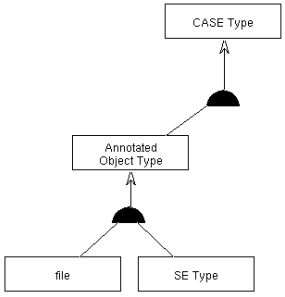
The Application Id Type Attribute: appid Suffix

Applications can use an attribute value to identify a reference within a file. This type of attribute has a name ending with the suffix "appid."

Annotated Object Supertype

The Annotated Object supertype has file and Software Engineering (SE) subtypes.

Figure 2: Annotated Object Supertype

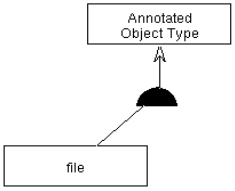


The subtypes can have information associated with them in the form of annotations. These consist of a set of user-specified notes that contain a set of user-specified items and a text description. Applications can search for objects and sort them in the StP repository based on information in the annotation.

File Subtype

File objects in the StP repository are used by applications to track the files that belong to a particular StP system, the file locations, and the file types.

Figure 3: File Subtype



Files play a key role in the StP environments. Diagrams, tables, annotations, documents, and source code are all stored in files that are managed by the host machine's operating system. However, operating systems do not track much of the information about files that applications need. You can specify in the file object's annotation any other information pertaining to files that you want to track.

Table 4 lists attributes of the file subtype. The "KEY" column indicates whether corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions, see "Attributes of CASE Types" on page 2-6.

Table 4: File Subtype Attributes

Attribute	KEY
id	N
name	Y
type	Y
rev	N
fname	N
lview_id	N
lmdfy_id	N
lsync_id	N
annot_file_id	N

Related Topics

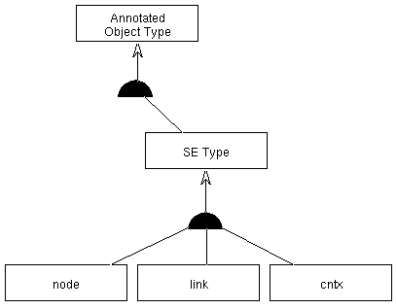
See also the following subtypes and functions:

- "File_hist Subtype" on page 2-26
- "File_lock Subtype" on page 2-24
- "oms_file_tp" on page 5-19
- "oms_file_hist_tp" on page 5-13
- "oms_file_lock_tp" on page 5-16

Software Engineering Supertype

The subtypes of the Software Engineering supertype (SE Type) represent and store concepts in the analysis and design domain that analysts, designers, and modelers use to build models. They may model these concepts using shapes in diagrams. Each concept represented in a diagram is stored as an SE object in the StP repository.

Figure 4: Software Engineering Supertype



For example, when users design with an entity-relationship editor, they are representing information in the real world and its inter-relationships. When the users create diagrams, the elements available to them are various application types, such as entities, relationships, and attributes. In the StP repository, these application types are stored as SE Type objects.

There are three subtypes of the SE Type:

- Node type
- Link type
- Cntx type

All persistent objects that belong to an SE subtype share the same structure. For example, all links share the same structure. One type of object is differentiated from another type by the value of the Application Type attribute (oms_app_type_tp). SE objects also have annotations that permit the user to store any related information about the object.

Identifying Attributes for SE Types

Two kinds of attributes provide values that help uniquely identify SE objects whose identity might be ambiguous:

- Scope attribute
- Signature attribute

The Scope Attributes

There are two scope attributes:

- scope_node_id
- scope_node_ref_id

When determining which object is being referred to by a symbol in the graphic editors, StP gets the object's name from the symbol's label and the type from the symbol's shape. However, sometimes the name and type are not sufficient to uniquely identify it.

Figure 5 shows a simple OMT diagram with two classes that each contain an operation named "swap." Although the labels for the operations are the same, the symbols represent two different operations. One represents an operation defined in Class 1 and the other represents an operation defined in Class 2. In this case, the name and type of an object are not enough to uniquely identify the node in the StP repository. To distinguish between the two node objects, StP needs to use a scope attribute.

Figure 5: "Swap" Operations Having Different Scopes

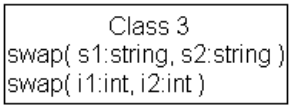
The scope_node_id attribute permits the nesting of some node objects inside of others. The value of the scope_node_id attribute for each of the operations in Figure 2 is the id of their respective parents. By using the scope, StP can distinguish one operation from another.

The Signature Attribute

The signature attribute provides a way for an application to distinguish objects with virtually identical identifying attributes from each other. For example, two or more nodes may have the same name, type and scope. By using the signature attribute, an application can uniquely identify each of these similar nodes. The OMS does not interpret the signature attribute.

Figure 6 shows another simplified OMT diagram. This one shows one class that contains two operations, both of which are named "swap." This is legal in languages such as Ada and C++ and is called "overloading." In this situation, the name, type, and scope of the two node objects that the operation symbols represent are the same. StP needs to use their signatures to distinguish them from one another.

Figure 6: "Swap" Operations Having Different Signatures



In programming languages, signatures are normally fabricated from information about the arguments. If one of the node objects has a signature value of "string s1, string s2" and the other has a signature of "int i1, int i2" applications can distinguish one node from the other by using the signature. The idea is not unique to programming languages. Signatures in the OMS are a general concept and can be used to differentiate between any two nodes that have the same name, type, and scope.

Node Subtype

The node subtype represents a wide variety of units in the design domain such as processes, externals, modules, data modules, packages, tasks, and states. The symbols that represent these objects in StP diagrams are usually node symbols. In tables, nodes often correspond to cells. Nodes do not include links or context (cntx) objects.

Table 5 lists attributes of the node subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 5: Node Subtype Attributes

Attribute	KEY
id	N
name	Y
type	Y
scope_node_id	Y
sig	Y
annot_file_id	N

Related Topics

See also the following subtypes and functions:

- "Node_ref Subtype" on page 2-30
- "Viewpoint Subtype" on page 2-27
- "oms_node_tp" on page 5-38
- "oms_node_ref_tp" on page 5-34

Link Subtype

The link subtype represents relationships or associations between node type objects. A link object always has a "from" node and a "to" node which represent the two objects that it relates. Link type objects can represent diverse relationships such as dataflow, component of, and subtype of.

Table 6 lists attributes of the link subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or

not (“N”). For descriptions of attributes, see “Attributes of CASE Types” on page 2-6.

Table 6: Link Subtype Attributes

Attribute	KEY
id	N
name	Y
type	Y
from_node_id	Y
to_node_id	Y
sig	Y
scope_node_id	N
annot_file_id	N

Related Topics

See also the following subtypes and functions:

- “Link_ref Subtype” on page 2-31
- “Cntx Subtype” on page 2-19
- “oms_link_tp” on page 5-30
- “oms_link_ref_tp” on page 5-27

Cntx Subtype

The cntx subtype represents information about links. A cntx object is attached to and cannot exist without a link object as its parent; it contains some aspect of that link’s information. Unlike node objects, which can be nested within each other to distribute information in a manageable way, link objects cannot be decomposed.

The subroutine call link in Ada is an example of a link that has a great deal of information. Links representing subroutine calls may have several

parameters associated with them. Information about these parameters can be conveniently stored as cntx objects.

Table 7 lists attributes of the cntx subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 7: Cntx Subtype Attributes

Attribute	KEY
id	N
name	Y
type	Y
sig	Y
link_id	Y
scope_node_id	N
annot_file_id	N

Related Topics

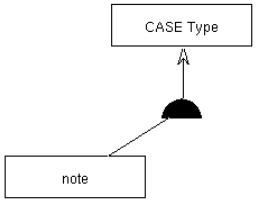
See also the following subtypes and function:

- "Cntx_ref Subtype" on page 2-32
- "Link Subtype" on page 2-18
- "oms_cntx_tp" on page 5-9
- "oms_cntx_ref_tp" on page 5-6
- "oms_link_tp" on page 5-30

Note Subtype

The note subtype represents one of the elements of an object's annotation (the other element is the item data type). An annotation enables you to extend the information about a particular CASE object in the StP repository. For example, you can specify a series of notes that are kept for each "process" node or each "call" link. Each note has a type and a name which uniquely identifies it within the context of a particular object.

Figure 7: Note Subtype



Each note also has a description field which consists of a character string of arbitrary length. The value of the description attribute might be a comment with some explanatory information about the note, source code or program design language (PDL) to describe the implementation aspect of this object, or any other appropriate information.

Table 8 lists attributes of the note subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 8: Note Subtype Attributes

Attribute	KEY
id	N
obj_id	Y
type	Y

Table 8: Note Subtype Attributes (Continued)

Attribute	KEY
name	Y
file_id	N
desc	N

Related Topics

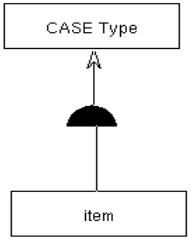
See also the following subtype and functions:

- “Item Subtype” on page 2-22
- “oms_note_tp” on page 5-42
- “oms_item_tp” on page 5-24

Item Subtype

The item subtype represents one of the elements of an object‘s annotation. Item objects belong to notes (see “Note Subtype” on page 2-21). They are name / value pairs that you specify along with each note type. You can add a note to an object, and based on the type of note, add individual items to that note. You select the type of item to add to the note and associate a value with it.

Figure 8: Item Subtype



For example, if you want to find all StP/IM relationships with a multiplicity of many in your system, you can query the StP repository for all relationships that have a Cardinality and Existence note with a Cardinality item set to "many."

An item's value is stored in the value attribute in any mix of upper or lower case letters, exactly as entered in the editor. When you specify the value attribute, StP creates a copy of that value in the svalue attribute, in all lower case letters. Therefore, by querying the repository for a specific svalue (specified in lower case letters), you can perform what is in essence a case insensitive query on an item's value.

Table 9 lists attributes of the item subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 9: Item Subtype Attributes

Attribute	KEY
id	N
node_id	Y
type	Y
value	Y
svalue	N
obj_id	N

Related Topics

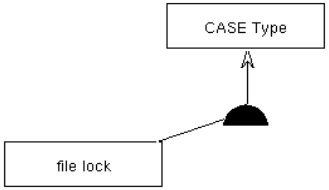
See also the following subtype and functions:

- "Note Subtype" on page 2-21
- "oms_item_tp" on page 5-24
- "oms_note_tp" on page 5-42

File_lock Subtype

The file_lock subtype represents a constraint placed on a file object. Any subsequent process that requires access to the file object can check to see if a lock exists for that file and, if so, take the appropriate action. If you try to load a diagram file into one of the editors and a lock has been placed on the corresponding file object, you can view the file but you cannot modify it.

Figure 9: File_lock Subtype



Each lock applies to one file object. Each of these contains a list of users who can access the file (accessors) and a list of users who can delete the lock (destroyers).

File locks are not enforced by the operating system, so each file is protected only if all applications that have access to the file check for locks in the StP repository and follow the established protocol.

When locking is enabled in an StP system, graphical editors place an "automatic" lock on a file when it is loaded. The OMS removes this lock whenever the editor terminates or loads a different file. A "permanent" lock can be placed on a file and will remain until you explicitly remove it or the duration of the lock is expired.

Table 10 lists attributes of the file_lock subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 10: File_lock Subtype Attributes

Attribute	KEY
id	N
file_id	Y
type	Y
user	N
accessors	N
destroyers	N
time	N
duration	N
hostname	N
pid	N

Related Topics

See also the following subtypes and function:

- “File Subtype” on page 2-13
- “File_hist Subtype” on page 2-26
- “oms_file_lock_tp” on page 5-16
- “oms_file_tp” on page 5-19
- “oms_file_hist_tp” on page 5-13

File_hist Subtype

File_hist (file history) objects store information about events that pertain to files. Creating, viewing, and modifying a file are examples of history events that can be tracked. Because file_hist objects store the time and date of the event, these objects can be used for dependency checking.

Figure 10: File_hist Subtype

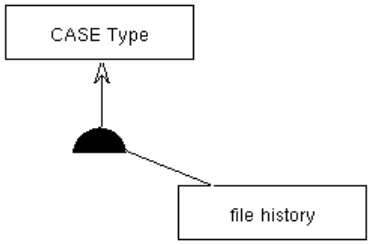


Table 11 lists attributes of the file_hist subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 11: File_hist Subtype Attributes

Attribute	KEY
id	N
type	Y
file_id	Y
time	Y
user	Y
hostname	Y

Related Topics

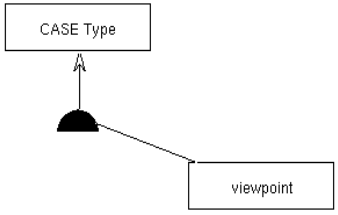
See also the following subtypes and functions:

- “File Subtype” on page 2-13
- “File_lock Subtype” on page 2-24
- “oms_file_hist_tp” on page 5-13
- “oms_file_tp” on page 5-19
- “oms_file_hist_tp” on page 5-13

Viewpoint Subtype

A viewpoint is a diagram symbol that you designate as a focal point for a certain type of information about a node object. Other symbols in the same system can show the same node, but only one symbol is a viewpoint.

Figure 11: Viewpoint Subtype



For example, an entity in an entity-relationship model can appear three times in three diagrams in a system. In two of these instances, the attributes connected to the entity are different. If the third instance is designated the “all attributes” viewpoint, all the attributes connected to the entity in the other two instances must also be connected to the viewpoint.

You can designate one symbol as multiple viewpoints. You cannot split a viewpoint among objects. It is not necessary that you designate every symbol on a diagram as a viewpoint.

You designate a symbol as a viewpoint by choosing from a list of viewpoint types furnished by the diagram editor. This information is stored in the StP repository.

StP can use viewpoint information to automatically navigate to the designated symbol and to validate diagrams.

Table 12 lists attributes of the viewpoint subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 12: Viewpoint Subtype Attributes

Attribute	KEY
id	N
node_id	Y
node_ref_id	Y
type	Y

Related Topics

See also the following subtype and function:

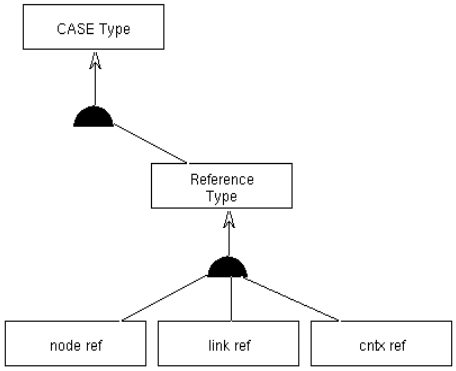
- "Node Subtype" on page 2-17
- "oms_viewpoint_tp" on page 5-49
- "oms_node_tp" on page 5-38

Reference Supertype

Where Software Engineering type objects store information about the things that symbols represent, reference type objects store information about the symbols themselves. Generally, one reference object appears in the StP repository for each symbol on a diagram. These objects permit StP

to know every place in the system where a reference to an SE object appears.

Figure 12: Reference Supertype



In addition to symbols, references to SE objects can also be in tables, source code, documents, or any other type of file.

Reference objects enable applications to:

- Perform consistency checking between diagrams and throughout the system without parsing diagram files
- Retrieve graphical information about the position of each symbol to aid reverse engineering applications and applications that need to sort output based on the graphical positioning of objects in a diagram
- Delete objects that are no longer referred to in any file in the system
- Make changes to every reference of an object
- Navigate to any reference to an object based on user-specified criteria

There are three subtypes of the reference supertype:

- Node_refs
- Link_refs
- Cntx_refs

Node_ref Subtype

The OMS stores one node_ref type object in the StP repository for each reference to a node object. A node_ref object stores the id of the file containing the reference and an integer identifier which uniquely identifies that reference within the file.

The meaning of this identifier is unknown to the OMS and varies according to the type of file that contains the reference. If the reference is a symbol in an StP diagram, the application that synchronizes the StP repository with the diagram uses the symbol id assigned by the graphical editors as the internal file identifier of the reference. Applications that generate references from source files might use a line number as the identifier. Other applications might use other types of internal identifiers that work within a file.

Table 13 lists attributes of the node_ref subtype. The "KEY" column indicates whether the corresponding attribute is an identifying attribute ("Y") or not ("N"). For descriptions of attributes, see "Attributes of CASE Types" on page 2-6.

Table 13: Node_ref Subtype Attributes

Attributes	KEY
id	N
node_id	Y
file_id	Y
appid	Y
scope_node_ref_id	N
xcoord	N
ycoord	N

Related Topics

See also the following subtype and functions:

- “Node Subtype” on page 2-17
- “oms_node_ref_tp” on page 5-34
- “oms_node_tp” on page 5-38

Link_ref Subtype

One link_ref type object exists in the StP repository for each reference to a link in the system. Link_refs represent references to links from any file type.

Link_ref objects, like node_ref objects, store the file_id of the file that contains the reference and the internal application id that uniquely identifies the link within a file.

Table 14 lists attributes of the link_ref subtype. The “KEY” column indicates whether the corresponding attribute is an identifying attribute (“Y”) or not (“N”). For descriptions of attributes, see “Attributes of CASE Types” on page 2-6.

Table 14: Link_ref Subtype Attributes

Attributes	KEY
id	N
link_id	Y
file_id	Y
appid	Y
from_node_ref_id	Y
to_node_ref_id	Y
xcoord	N
ycoord	N

Related Topics

See also the following subtypes and functions:

- “Link Subtype” on page 2-18
- “Cntx Subtype” on page 2-19
- “oms_link_ref_tp” on page 5-27
- “oms_link_tp” on page 5-30
- “oms_cntx_tp” on page 5-9

Cntx_ref Subtype

The OMS stores one cntx_ref object in the StP repository for each reference to a cntx object in the system. If a cntx symbol is attached to an arc symbol that may participate in more than one link, it refers to a different object for each link where the arc symbol is a member. Just as cntx objects must be attached to link objects, cntx_refs must be attached to link_refs.

Table 15 lists attributes of the cntx_ref subtype. The “KEY” column indicates whether the corresponding attribute is an identifying attribute (“Y”) or not (“N”). For descriptions of attributes, see “Attributes of CASE Types” on page 2-6.

Table 15: Cntx_ref Subtype Attributes

Attribute	KEY
id	N
cntx_id	Y
file_id	Y
appid	Y
link_ref_id	N
xcoord	N
ycoord	N

Related Topics

See also the following subtypes and function:

- “Link Subtype” on page 2-18
- “Link_ref Subtype” on page 2-31
- “oms_cntx_ref_tp” on page 5-6
- “oms_link_tp” on page 5-30
- “oms_link_ref_tp” on page 5-27

Referential Integrity

The OMS places referential integrity constraints on objects in the StP repository based on the relationships between data types in the Persistent Data Model. Relationships between objects in the StP repository are represented by one object having an attribute that contains the related object’s id value. For example, a node, B, is related to its scope node (parent), A, by virtue of B’s scope_node_id attribute value being identical to node A’s id attribute value.

An object must exist for each logical pointer from another object. No object can be stored in the StP repository unless all related objects are already in the repository. For example, a cntx object cannot be stored in the StP repository unless the link object to which the cntx object is attached is in the repository.

Referential Integrity Relationships

Figure 13 on page 2-35 illustrates referential integrity constraints imposed by the PDM. In the figure:

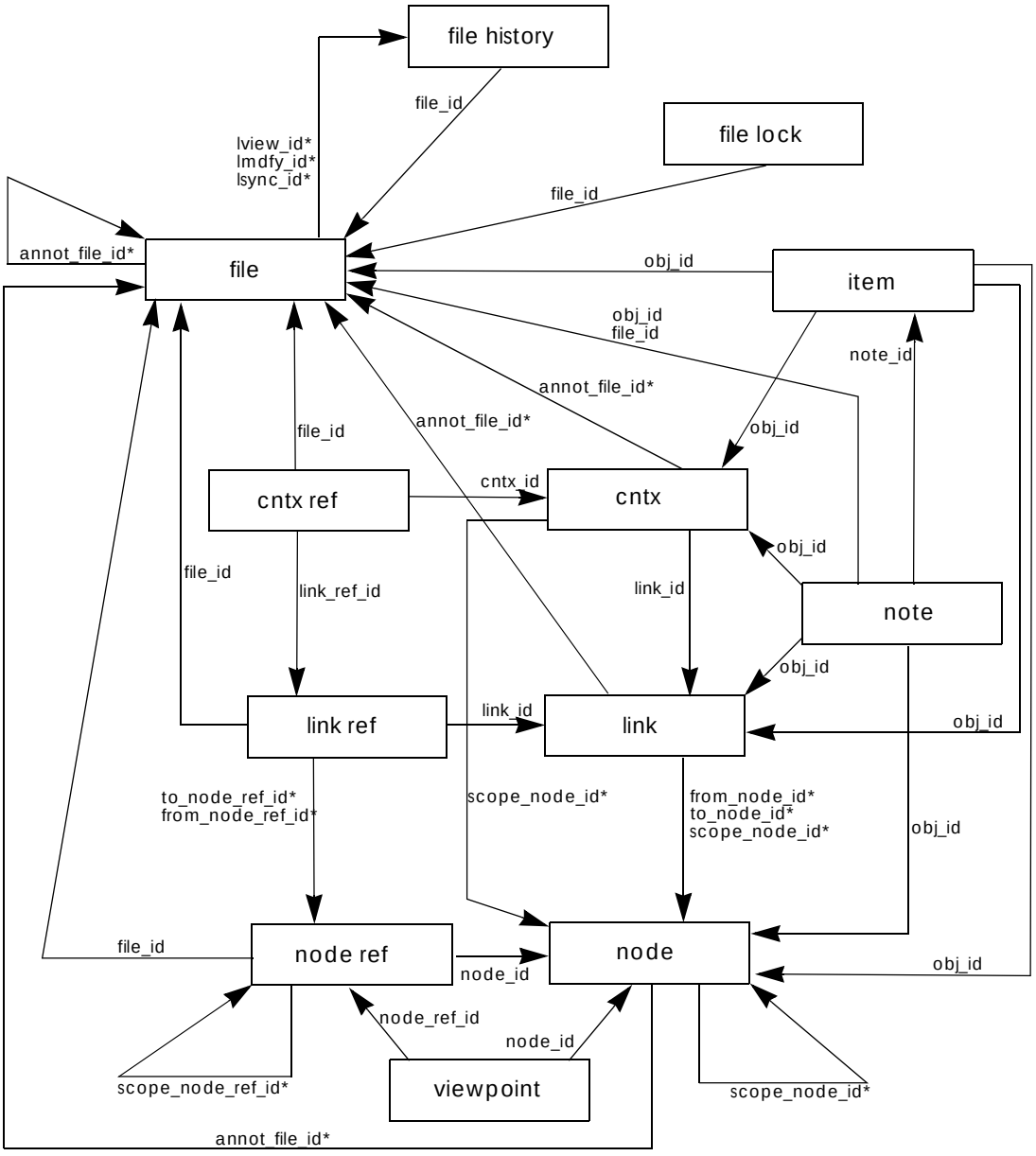
- Arcs connect the object type whose existence is a requirement (the independent object type) with the dependent object type. The arc originates from the dependent object type and points to the independent object type.
- Each arc label is the name of the dependent object type’s attribute that contains the related object’s id value.

- In general, any attribute whose name ends in "_id" represents another object in the StP repository which must exist before the current object can be stored.
- An asterisk indicates an attribute that can have the value 0. If the value of an attribute is 0, the OMS does not enforce the relative object's existence. For example, if a node has an annot_file, annot_file_id is non-zero and an annot_file must exist. If there is no annot_file, the annot_file_id is 0, and no annot_file exists.

Storing Nodes with Scope Parents

Most of the relationships between objects in the PDM occur between objects of different types. For example, links are connected to nodes, cntxs are attached to links, and reference objects correspond to SE objects. However, a scope relationship can exist between node objects. This means that before storing a node object which has a scope parent, the scope parent must be in the StP repository. Therefore, applications may have to order node objects according to their scope hierarchy before attempting to store them in the repository.

Figure 13: Referential Integrity Relationships



3 OMS Query Language

This chapter explains how to use the Object Management System query language to query repository data.

Topics covered are as follows:

What Is the OMS Query Language?

The Object Management System query language enables applications to query the StP repository for data in response to a wide range of attributes and relationship values. The query language is based on the Persistent Data Model and uses a simple syntax based on types, attributes, and relationships between types. Because the query language is an interpreted language, applications do not need every query at compile time.

A query can be issued for any of the persistent CASE object types, and the result is always a collection containing objects of that type.

Most of the tools in Software through Pictures use OMS queries. Examples include:

- Browsing the StP repository with the Repository Browser

- Creating documents using the Query and Reporting System

To retrieve data in the StP repository, you can:

- Use OMS queries in the Repository Browser (see *Fundamentals of StP*)
- Embed an OMS query in a Query and Reporting Language (QRL) statement (see *Query and Reporting System*)
- Embed an OMS query in an associative query function, which interprets the query and returns a collection satisfying the query's conditions (see Chapter 4, "Using the Application Program Interface")

Basic Syntax

The syntax of an OMS query is as follows, where **<pdm_type>**, **<restrictor>**, and **<attr_list>** are variables described in each subsection, and the plain (non-bold) square brackets indicate optional clauses:

<pdm_type> [**[<restrictor>]**] [**sort by <attr_list>**]

When you specify the restrictor, be sure to enclose it in brackets. For example, the following query searches for all files whose names begin with ow and sorts them by their id attributes:

```
file [name = 'ow*'] sort by id
```

Valid Values for pdm_type

The pdm_type part of an OMS query is equivalent to any valid PDM type:

- file_lock
- note
- item
- file_hist
- viewpoint
- file

- node_ref
- link_ref
- cntx_ref
- node
- link
- cntx

The Restrictor Clause

Objects can be selected based on the values of their attributes and their relationships with other objects. A relationship is an association of one object with another.

You specify selection criteria in the *restrictor*. A restrictor is one or more search conditions composed of:

- Zero or more attributes
- Zero or more relationships, specified as the type that participates in the relationship with the `pdm_type`
- Zero or more relational or boolean constructs

You can combine two or more search conditions with one another. You must enclose restrictors within brackets. You can enclose parts of a restrictor within parentheses to create an arbitrary grouping.

Queries Based on Attribute Values

You can construct a query based on one or more of a persistent CASE object's attribute values. If the attribute value is a character string, enclose it in single quotes. Otherwise no punctuation is required.

For example, this query searches for nodes whose type attribute value is "entity":

```
node [type = entity]
```

When using a type name you do not have to include `type =` . In this short form, the previous query is:

```
node[entity]
```

You must use attributes that are valid for the object type (as described in Chapter 2, “Understanding the Persistent Data Model” and in Table 2 on page 3-9). For example, valid attributes for node are “id,” “name,” “type,” “scope_node_id,” “sig,” and “annot_file_id.”

Queries Based on Relationships

You can construct a query based on the relationship between two or more persistent CASE objects. For example, the following query searches for cntxs that have relationships with links whose type attribute value is “dataflow”:

```
cntx [link[type=dataflow]]
```

or

```
cntx [link[dataflow]]
```

You must use only relationships that are valid for the given PDM type. For example, the valid relationships for cntx are “link,” “scope_node,” “annot_file,” “notes,” “items,” and “cntx_refs.”

Comparison Operators

Table 1 lists attribute comparison operators that you can use in OMS queries.

Table 1: Attribute Comparison Operators

Operator	Description
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Not equal to

Table 1: Attribute Comparison Operators (Continued)

Operator	Description
$	Match the following pattern (for string attributes only)

Sorting

The results of any query can be sorted using the **`sort by <attr_list>`** clause. The **`<attr_list>`** variable can be any list of attributes belonging to the PDM type, for example, “sort by id”.

Boolean Constructs and Pattern Matching

The OMS supports the Boolean constructs:

- Not (!)
- And (&&)
- Or (| |)

“Not” takes the highest precedence, followed by “And,” then ”Or.”

Regular expressions, such as *, are equivalent to those used for UNIX Bourne shell pattern matching.

This query returns entities whose names begin with “b”:

```
node [entity && name $ 'b*']
```

Quotation Marks

Always use single quotes for string or pattern matching.

Examples

The following OMS query examples progress from the simplest example to more complex ones. For details about constructing an OMS query,

including valid relationships for each persistent CASE object, see “Constructing an OMS Query” on page 3-7.

Query Based on Pdm_type

This shows a query based on a PDM type; it returns all nodes:

```
node
```

Query Based on a Single Attribute Value

This query shows the PDM type as node. Its restrictor specifies an attribute value, “process”:

```
node [type = process]
```

or

```
node [process]
```

Query Based on Combined Attribute Values

This example uses a restrictor that combines two attribute values. It returns all nodes whose type attribute is “process” and whose name attribute begins with “sc”:

```
node [type = process && name $ 'sc*']
```

or

```
node [process && name $ 'sc*']
```

Query Based on a Relationship

This example returns all nodes that have at least one related link that leaves that node:

```
node[out_links]
```

Query Based on the Attribute of a Relationship

This example returns all links that originate at a "process" node. It uses two restrictors: one specifies an attribute value ("process") and the other a relationship between two objects ("link" and "from node").

```
link [from_node[type=process]]
```

Query Based on Multiple Attributes and Relationships

This example returns all the links that have a note of type "base" with an item of type "author". To do this, it uses a PDM type of link and three restrictors: one specifying related objects (notes), one specifying objects related to notes (items) and the third specifying the value of items' attribute "type." This query uses the Boolean operator "and" (&&).

```
link [notes[type = GenericObject && items[type = Author]]]
```

Query Based on Boolean "Not"

This example applies the boolean operator "not" (!) to the previous example. It returns all links that do not have a note of type "base" with an item of type "author".

```
link[!notes[type = GenericObject && items[type = Author]]]
```

Sorting the Results of a Query

This example returns all cntx objects and sorts by name and signature:

```
cntx sort by name, sig
```

Constructing an OMS Query

When constructing an OMS query, follow these guidelines:

- A query must always begin with a pdm_type.
- A restrictor may include attributes and relationships.

- If a restrictor includes a relationship, that relationship must be defined for the pdm_type type to which the restrictor applies.

Table 2 provides a guide for constructing valid OMS queries. Each valid type (PDM type) is listed with its attributes and relationships and descriptions of the relationships. For example, the cntx type can own the notes and items types, and is referenced by cntx_refs. Also provided are the identifying attributes of the type (column IA) and the number of related types that can participate in a relationship with a single type.

For more information about types and attributes, see “CASE Types” on page 2-4.

Table 2: PDM Type with Attributes and Relationships

Type	Attribute	IA	Relationship	Cardinality	Description
cntx	id				
	name	Y			
	type	Y			
	sig	Y			
	link_id	Y	link	Single	is attached to
	scope_node_id		scope_node	Single	is scoped to
	annot_file_id		annot_file	Single	owns
			notes	Many	owns
			items	Many	owns
			cntx_refs	Many	is referenced by
cntx_ref	id				
	cntx_id	Y	cntx	Single	references
	file_id	Y	file	Single	is contained by
	appid	Y			
	link_ref_id		link_ref	Single	is attached to
	xcoord				
	ycoord				
file_hist	id				
	file_id	Y	file	Single	tracks events on
	type	Y			
	time	Y			
	user	Y			
	hostname	Y			

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
file_lock	id				
	file_id	Y	file	Single	locks
	type	Y			
	user				
	accessors				
	destroyers				
	time				
	duration				
	hostname				
	pid				
file	id				
	name	Y			
	type	Y			
	rev				
	fname				
	lview_id		lview_hist	Single	is tracked by
	lmdfy_id		lmdfy_hist	Single	is tracked by
	lsync_id		lsync_hist	Single	is tracked by
	annot_file_id		annot_file	Single	owns

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
file			notes	Many	owns
			node_refs	Many	contains
			link_refs	Many	contains
			cntx_refs	Many	contains
			file_hists	Many	is tracked by
			file_locks	Many	is locked by
			annot_node	Single	holds annotation for
			annot_link	Single	holds annotation for
			annot_cntx	Single	holds annotation for
			contains_notes	Many	has inside
item	id				
	note_id	Y	note	Single	is owned by
	type	Y			
	value	Y			
	svalue	Y			
	obj_id				
			file	Single	is owned by
			node	Single	is owned by
			link	Single	is owned by
			cntx	Single	is owned by

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
link_ref	id				
	link_id	Y	link	Single	references
	file_id	Y	file	Single	is contained by
	appid	Y			
	from_node_ref_id	Y	from_node_ref	Single	connects from
	to_node_ref_id	Y	to_node_ref	Single	connects to
	xcoord				
	ycoord				
			cntx_refs	Many	is attached by
link	id				
	name	Y			
	type	Y			
	from_node_id	Y	from_node	Single	connects from
	to_node_id	Y	to_node	Single	connects to
	sig	Y			
	scope_node_id		scope_node	Single	is scoped to
	annot_file_id		annot_file	Single	owns
			notes	Many	owns
			items	Many	owns
			link_refs	Many	is referenced by
			cntxs	Many	is attached by

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
node_ref	id				
	node_id	Y	node	Single	references
	file_id	Y	file	Single	is contained by
	appid	Y			
	scope_node_ref_id		scope_node_ref		is scoped to
	xcoord				
	ycoord				
			scoped_node_refs	Many	scoped by
			out_link_refs	Many	is source of
			in_link_refs	Many	is destination of
			viewpoints	Many	is designated by

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
node	id				
	name	Y			
	type	Y			
	scope_node_id	Y	scope_node	Single	is scoped to
	sig	Y			
	annot_file_id		annot_file	Single	owns
			notes	Many	owns
			items	Many	owns
			viewpoints	Many	is designated by
			node_refs	Many	is represented by
			scoped_nodes	Many	scopes
			scoped_links	Many	scopes
			scoped_cntxs	Many	scopes
			out_links	Many	is source of
			in_links	Many	is destination of

Table 2: PDM Type with Attributes and Relationships (Continued)

Type	Attribute	IA	Relationship	Cardinality	Description
note	id				
	obj_id	Y			
	type	Y			
	name	Y			
	desc				
			items	Many	owns
			file	Single	is owned by
			node	Single	is owned by
			link	Single	is owned by
			cntx	Single	is owned by
viewpoint	id				
	node_id	Y	node	Single	designates
	type	Y			
	node_ref_id		node_ref	Single	designates

Optimizing OMS Queries

An optimized OMS query reduces the number of times a program needs to search the StP repository. Guidelines for optimizing an OMS query include:

- Include all search criteria in a single query, if possible.
- Avoid nesting one OMS query in another, if possible.

Using Single Queries

One OMS query can extract information about any set of objects that is symbolically linked. By combining criteria in one query, you can reduce the number of times the program needs to search the StP repository. Also, a single query uses the power of the Persistent Data Model by having the StP repository devise the best search strategy.

For example, this single query extracts all nodes that are destinations of links that leave an entity:

```
node[in_links[from_node[Entity]]];
```

Avoiding Nested Queries

Generally, it is better to avoid nesting OMS queries. However, if objects share a name association, nested queries may be unavoidable. For example, this Query and Reporting Language (QRL) routine extracts any attribute that shares a name with another attribute:

```
query = 'node[Attribute]';
for_each_in_select (query, attribute)
{
   query = "node[Attribute && name ="+
      "'${attribute.name}' && id ="+
      "${attribute.id}]";
   for_each_in_select(query, duplicate)
   <qrl statements to process the query result>
}
```

For details about the QRL, see *Query and Reporting System*.

Using ID Lists

An id list is a temporary holder for the results of an OMS query that enables you to reuse the results in other queries. Rather than repeat a query throughout a routine, you can create an id list for the query's results, then refer to the id list throughout the routine. Id lists let you

simplify complex queries by breaking them up, storing the results, then reusing the id list in a higher-level query.

You can create an id list using the QRL id_list_create statement. For example:

```
void id_list_create(string query, string id_list_name);
```

When the routine is finished with the id list, free it using the id_list_free statement:

```
void id_list_free(string id_list_name);
```

You can also create an id list with an API function (see “Id List Functions” on page 4-26).

Examples

The examples in this section are embedded in the Query and Reporting Language. For details about this language, see *Query and Reporting System*.

This example is a QRL routine that uses an id list of all attributes in the current diagram.

```
//External arguments
external string diagram = "";
external_help = "Name of Diagram";
void
main()
{
string query;
node current_node;
```

Save the list of id’s of all attributes in the current diagram so the repository doesn’t have to refetch them:

```
query = "node[Attribute && node_refs[file[name = ’"+
diagram + "’]]]";
id_list_create(query, "attributes");
```

Select all attributes from the “attributes” id list that are connected to another symbol:

```
query = "node[attributes && (in_links || out_links)]";
for_each_in_select(query, current_node)
   print(current_node.type + " "+ current_node.name +
       " is connected" + " to another symbol.\n");
```

The name of the id list (“attributes”) is embedded in the query.

Select all attributes from the “attributes” id list that have a SybaseColumn note:

```
query = "node[attributes && notes[SybaseColumn]]";
for_each_in_select(query, current_node)
   print(current_node.type + " "+ current_node.name +
       " has a "+ "SybaseColumn note.\n");
```

Select all attributes from the “attributes” id list that are connected to a DataElement:

```
query = "node[attributes &&" +
in_links[from_node[DataElement]]]";
   for_each_in_select(query, current_node)
   print(current_node.type + " "+ current_node.name +
       " is connected "+ "to a DataElement.\n");
```

The queries are finished with this id list:

```
id_list_free("attributes");
```

}

4 Using the Application Program Interface

This chapter explains how to use the Application Program Interface (API) functions to manipulate objects in the repository.

Topics covered are as follows:

For detailed information on API functions, see Chapter 5, "Application Program Interface Types and Functions."

What Is the Application Program Interface?

The Application Program Interface (API) is a standard interface between C ++ application programs and the StP repository. You use it to simplify the application program's repository-related tasks. To manipulate repository data, StP tools rely on the OMS API rather than on repository-specific Embedded SQL or 4GL.

It is possible to incorporate calls to API functions in customized application programs to retrieve data from the StP repository. However, you may be able to accomplish the same task more easily by embedding OMS query language statements in a QRL routine, or by using QRL alone (see *Query and Reporting System*).

The API tools described in this manual let you manipulate data, but ***be very careful when doing so!*** Bear in mind that if you use the API to write to the repository, the changes are ***not*** reflected in the StP system files. This can result in inconsistencies between the system files and the repository. Changes made to the repository alone are overwritten by data in the ASCII files if:

- The repository is rebuilt from the system files
- The contents of a diagram or table loaded from an ASCII file are out of synch with the repository data, and the diagram or table is re-saved to the repository

API Requirements

To use the API you need:

- If you are using Sybase Adaptive Server or SQL Server, the Sybase OpenClient/C software
- If you are using Microsoft Jet, ODBC, SDK (Software Developer's Kit), and DAO (Data Access Objects) SDK
- A supported C++ compiler

You can choose the option to install the OpenClient/C software when you install the Sybase Adaptive Server. If the directory in which Sybase has been installed contains *include/* and *lib/* subdirectories, Sybase OpenClient/C has already been installed.

Supported Compilers

The following compilers have been tested with these operating systems only:

Table 1: Tested compilers

Operating System	Compiler
NT 4.0	MS Visual C++ 5.0

Compiling and Linking OMS API Programs

The following sample program and makefile show the necessary include files and libraries you need to successfully compile and link an OMS API program.

To successfully link and run an OMS API program, you must also include the file *stpapp.obj* in the compile and link line. You can find this file in *<install_dir>/lib/<arch>* with the other OMS API include files (see the sample program below).

Sample Program

This sample program (*oms_test.C*) connects to the default system in the default project as set by the *system* and *projdir* ToolInfo variables. For more information, refer to the *StP Administration Guide*. It then prints out and counts the number of nodes, links, and cntx objects in the repository. Before running this program, make sure that the *system*, *projdir*, and other ToolInfo environment variables are set.

```
#include "oms_api.h"
#include <stdio.h>
#include <stdlib.h>
// Sample OMS API program to count the number of node, link and cntx
// objects in a system. */

// This code assumes that the environment variables IDE_PROJDIR,
// IDE_SYSTEM, and ToolInfo are all appropriately set.

int
main (int argc, char **argv)
{
   oms_coll_tp *oms_coll;
   int          node_count, link_count, cntx_count;
   // Open the repository for the default project and system specified
   //by the relevant environment or ToolInfo variables. */
   if (oms_sys_repos_open((char *) NULL, (char *) NULL) !=OMS_SUCCEED)
   {
      fprintf(stderr, "failed to open repository\n");
      exit(1);
   }
   oms_coll = oms_node_coll_create("node");
   node_count = oms_coll_count(oms_coll);
   oms_coll_free_all(&oms_coll);
   oms_coll = oms_link_coll_create("link");
   link_count = oms_coll_count(oms_coll);
   oms_coll_free_all(&oms_coll);
   oms_coll = oms_cntx_coll_create("cntx");
   cntx_count = oms_coll_count(oms_coll);
   oms_coll_free_all(&oms_coll);
   oms_repos_close();

   printf ("System has %d node, %d link, and %d cntx objects.\n",
   node_count, link_count, cntx_count);
   return 0;
}
```

Building the Sample Program

To build the sample program using Developer's Studio 5.0:

1. Create a new "Win32 Console Application" project and add the *oms_test.C* file.
2. Set the following project settings:

Table 2: Project Settings

	Category	Setting
C++/C	General	Project Options - add "**/TP**"
	Code Generation	Select "**Multithreaded DLL**" for runtime library
	Preprocessor	Add preprocessor definition "**W32NTX86**" Add additional include directory: ***<install_dir>\include***
Link	Input	Add additional "Object/Library modules": ***stptools.lib***, ***stpoms.lib***, and ***stpapp.obj*** Add additional library path: ***<install_dir>\lib\W32NTX86***

3. Build and execute the project.

   Be sure to have *<install_dir>\bin\W32NTX86* in your path at runtime to successfully find the required *.dll* files.

Overview of API Functions

The API provides functions that:

- Create and retrieve data
- Manage memory
- Manage the StP repository
- Perform operations on collections in the StP repository

- Interpret OMS queries
- Manipulate time strings
- Process StP repository transactions

The tables in this chapter list and describe each type of function. The "Function" column provides the generic name of each function (for example, oms_<pdm_type>_create). <pdm_type> indicates PDM data type, as in the specific function name (for example, oms_cntx_ref_create).

For details about PDM data types, see "CASE Types" on page 2-4. The examples in this chapter use attributes that are listed in Chapter 5, "Application Program Interface Types and Functions." For details about specific functions, see that chapter.

Auxiliary Types

The OMS uses auxiliary types, abstract data types that are not part of the Persistent Data Model. Auxiliary type objects help manage the process of creating, reading, updating, and deleting the persistent objects in the StP repository. Objects that are instances of these auxiliary types cannot be stored in the repository, so be careful when you use the auxiliary type functions. The auxiliary types are:

- Collection type
- Big Collection type
- Transaction type
- Id lists
- Repository type

Collection Type

Collection types group objects into a single unit. OMS routines that need to return a group of objects will do so in the form of a collection. The OMS provides operations to count, remove, store, and randomly access the members of a collection.

Big Collection Type

Use Big Collection types to retrieve a group of objects from the repository that is too large to be loaded into memory. The Big Collection type allows users to iterate through objects from the repository one at a time so that large groups of objects can be examined.

Transaction Type

The transaction type gives applications control over when a transaction begins and whether or not the actions performed on the StP repository are committed or aborted. The transaction type uses the following three operations:

- begin, which marks the starting point of the transaction
- commit, which completes, or rolls foward, the transaction and writes the results to the repository
- abort, which cancels the transaction and restores the values in the repository to their original state

If the begin operation is not called before a routine that affects the database is invoked, then that routine is treated as an individual transaction and is committed immediately upon success.

Nested calls to the begin operation are legal; they increment an internal counter. Calls to the commit operation have no effect until an equal number of calls to both operations have been made. Calls to the abort operation cause all the nested transactions to be aborted immediately.

Calls to functions that modify the StP repository may fail for many reasons, including deadlock and duplication of identifying attributes. You should carefully monitor errors to ensure that all operations that make up a logical transaction have completed successfully before committing a transaction to the database.

Transactions can lock data and cause other programs to block and wait for the transaction to complete.

Id Lists

An id list is a temporary holder for the results of an OMS query that enable you to reuse the results in other queries within the application. Rather than repeat a query throughout a routine, you can create an id list for the query's results, then refer to the id list throughout the routine. Id lists let you simplify complex queries by breaking them up, storing the results, then reusing the id list in a higher-level query. For more information, see "Id List Functions" on page 4-26.

Repository Type

The repository type contains information about the location and state of an StP repository. The location of a repository is indicated by a server name and a system-level or system-specific name. The state of a repository is either open or closed. A process can only have one repository open at any given time.

Creating and Retrieving Data

Creation and retrieval functions perform fundamental repository-specific operations on objects in the StP repository. Table 3 describes creation and retrieval functions.

Table 3: Creation and Retrieval Functions

Function	Description
oms_<pdm_type>_create	Searches the repository for an object having identifying attributes equal to this function's parameters. If one is found, a pointer to a newly allocated copy of the object is returned. If no object is found, a new one is created in memory and a pointer to it is returned.

Table 3: Creation and Retrieval Functions (Continued)

Function	Description
oms_<pdm_type>_find	Searches the current repository for an oms_<pdm_type>_tp object with the same identifying attributes as the parameters. If a matching object is found, it is loaded into a newly allocated oms_<pdm_type>_tp structure and a pointer to the new structure is returned. If no such object exists or an error occurs, a NULL pointer is returned.
oms_<pdm_type>_find_by_id	Searches the current repository for an oms_<pdm_type>_tp object with an id equal to the argument. If a matching object is found, it is loaded into a newly allocated oms_<pdm_type>_tp structure and a pointer to that structure is returned. If no oms_<pdm_type>_tp object is found with that id value or if an error occurs, a NULL pointer is returned.
oms_<pdm_type>_find_by_query	Searches the current repository for an oms_<pdm_type>_tp object that satisfies the conditions specified by the query parameter. If more than one object is found, a pointer to a newly allocated copy of an arbitrary object in the set is returned. If no such object is found, a NULL pointer is returned.

Example

This is a sample function to call creation and retrieval functions for a node object.

```
int sample_func_create_retrieve(
    const char *name, oms_app_type_tp type, oms_object_id_tp scope,
    const char *signature)

/* This function uses the following four node names, and provides a buffer
space of 1024 K. */
{
oms_node_tp *node1, *node2, *node3, *node4;
char buf[1024];

/* Assume that there are no nodes in the StP repository that have
identifying attributes that match the input ones. oms_node_find returns
null when it cannot find a node with the same identifying attributes: */

node1 = oms_node_find(name,type,scope,signature);
    if (node1) {
    printf("Found a node when we should not have\n");
    }

/* If the object doesn't exist in the StP repository, oms_node_create
creates a new object in memory:*/

node1 = oms_node_create(name,type,scope,signature);
if (!node1) {
    printf("Unable to create a new node\n");
    return -1}

/* The object's id is assigned a unique value and its identifying
attributes are assigned the values of the parameters.oms_node_find finds
the existing node and allocates a new structure for it: */

node2 = oms_node_find(name,type,scope,signature);
if (!node2) {
    printf("Unable to find existing node\n");
    return -1}

/* oms_node_find_by_id finds the existing node and allocates a new
structure for it: */

node3 = oms_node_find_by_id(oms_node_id(node2));
```

```
if (!node3) {
   printf("Unable to find existing node\n");
   return -1}

/* At this point, all three node pointers contain copies of the same
object. oms_node_find_by_query looks through the database for any nodes
that satisfy the query:*/

sprintf(buf, "node[name='%s' && type=%d &&
   scope_node_id=%ld and signature='%s']", name, type, scope,
   signature);

/* Now, it looks for the node based on the query: */

node4 = oms_node_find_by_query (buf);

return 0;
}
```

Using Utility Functions

Several functions are provided to help manage memory and perform other useful functions. Table 4 describes utility functions.

Table 4: Utility Functions

Function	Description
oms_<pdm_type>_copy	Allocates a new oms_<pdm_type>_tp structure and copies all of the attribute values from the argument to the new structure. The function returns a pointer to the newly allocated copy.

Table 4: Utility Functions (Continued)

Function	Description
oms_<pdm_type>_equal	Compares the identifying attributes of two oms_<pdm_type>_tp structures. If both sets of identifying attributes are equal then the function returns OMS_TRUE otherwise OMS_FALSE is returned. If both objects are represented by NULL pointers, then OMS_TRUE is returned.
oms_<pdm_type>_free	Takes the address of a pointer to an oms_<pdm_type>_tp structure and frees all memory associated with the structure and any of its attributes. NULL is assigned to the pointer after all memory has been freed. It has no effect on the repository.
oms_<pdm_type>_print	Takes the address of a pointer to an oms_<pdm_type>_tp structure and prints out the name of each attribute along with its value to a specified file.
oms_<pdm_type>_print_image_asgn	Replaces the default format string used by oms_<pdm_type>_print with the fmt argument. If fmt is NULL, this function resets the format string to the default. The format string is a standard format string used by printf as defined in the standard C library.

Example

This is a sample function to call utility functions for a node object.

```
int sample_func_utility(const char *name,oms_app_type_tp type,
   oms_object_id_tp scope,const char *signature,FILE*fp)

/* This function uses the following two node names. oms_boolean_tp will
provide an OMS_TRUE or OMS_FALSE value. */
{
oms_node_tp *node1, *node2;
oms_boolean_tp oms_return;

/* Next, use oms_node_create to create a node called node1: */

node1 = oms_node_create(name,type,scope,signature);

/* Now use oms_node_copy to copy node1 to a new node, node2. If it cannot
create it, it returns an error message. */

node2 = oms_node_copy(node1);
if (!node2) {
   printf("Unable to copy a new node\n");
   return -1}

/* Use oms_node_equal to check whether the nodes are equivalent:*/

oms_return = oms_node_equal(node1,node2);
if (oms_return == OMS_FALSE) {
   printf("Nodes not equivalent when they should have been\n");
   }

/* oms_node_print prints node1's attributes to fp: */

oms_node_print(node1,fp);

/* Now that you have completed the comparison, use oms_node_free to free
the memory allocated for the nodes: */

oms_node_free(&node1);
oms_node_free(&node2);

return 0:
}
```

Manipulating Attributes of Objects

Attribute access and assign functions provide a programmatic interface to attributes of objects in the StP repository. Generally, you should always use the appropriate function to access and assign values to OMS objects. Table 5 describes attribute access and assign functions.

Table 5: Attribute Access and Assign Functions

Function	Description
oms_<pdm_type>_<attribute>(oms_<pdm_type>_tp *obj)	Retrieves the value of an object from the StP repository.
oms_<pdm_type>_<attribute>_copy (oms_<pdm_type>_tp *obj)	Copies the value of an attribute from the StP repository (available for string attributes only).
oms_<pdm_type>_<attribute>_asgn (oms_<pdm_type>_tp *obj, <attribute_type> attribute)	Assigns a new value to an object's attribute.

Examples

This function retrieves the value of the cntx_id attribute from a pointer to a cntx_ref object:

```
oms_object_id_tp oms_cntx_ref_cntx_id (oms_cntx_ref_tp *cr)
```

This function returns a copy of the value of cntx_ref appid from a cntx object:

```
char   *oms_cntx_ref_appid_copy (oms_cntx_ref_tp *cr)
```

The next six function examples assign values to the attributes of cntx_ref:

```
void   oms_cntx_ref_cntx_id_asgn (oms_cntx_ref_tp *cr,
       oms_object_id_tp cntx_id);

void   oms_cntx_ref_file_id_asgn (oms_cntx_ref_tp
       *cr, oms_object_id_tp file_id);

void   oms_cntx_ref_appid_asgn
       (oms_cntx_ref_tp *cr,
       const char *appid);

void   oms_cntx_ref_link_ref_id_asgn
       (oms_cntx_ref_tp *cr,
       oms_object_id_tp link_ref_id);

void   oms_cntx_ref_xcoord_asgn
       (oms_cntx_ref_tp *cr,
        int xcoord);

void   oms_cntx_ref_ycoord_asgn
       (oms_cntx_ref_tp *cr,
       int ycoord);
```

Managing the StP Repository

Repository management functions perform repository-specific operations on objects in the StP repository. These functions are the only OMS functions that actually change data in the repository. Table 6 describes StP repository management functions.

Table 6: StP Repository Management Functions

Function	Description
oms_<pdm_type>_check	Checks to make sure an object passes validation checks. If any referential integrity constraints are violated, or an error occurs, it returns OMS_FAIL.
oms_<pdm_type>_update	Inserts or updates objects in the StP repository. It first searches the repository for a matching object (an object that has the same PDM type and id or the same identifying attributes as the current object). If one is found, the function attempts to update any attributes that are different. If no matching object is found, the function attempts to insert a new object. If any referential integrity constraints are violated or if an error occurs, the function returns OMS_FAIL; otherwise, it returns OMS_SUCCEED.
oms_<pdm_type>_delete	Deletes a persistent object from the StP repository. If an object exists in the repository with the same PDM type and id, and referential integrity is not violated, it is deleted. If no such object exists or if the delete operation succeeds, OMS_SUCCEED is returned. If deleting the object causes any referential integrity constraints to be violated or if an error occurs, it returns OMS_FAIL.

Example

This is a sample function to call StP repository management functions for a node object.

```
int    sample_func_rep_mgmt(const char *name, oms_object_type_tp type,
oms_object_id_tp scope,const char *signature, oms_object_id_tp annot)

/* This function uses the following node, node1. oms_status_tp will
provide an OMS_TRUE or OMS_FALSE value. */

{
oms_node_tp *node1;
oms_status_tp oms_return;

/* Use oms_node_create to create node1: */

node1 = oms_node_create(name,type,scope,signature);

/* Use oms_node_check to make sure node1 can be put into the StP
repository: */

oms_return = oms_node_check(node1);
if (oms_return != OMS_SUCCEED) {
   printf("Object cannot be placed into the respository\n");
   return -1;}

/* Use oms_node_update to put node1 into the StP repository if it does not
exist: */

oms_return = oms_node_update(node1);
if (oms_return != OMS_SUCCEED) {
   printf("Update failed\n");
   return -1;}

/* oms_node_update now modifies the StP repository after a change to a
non-identifying attribute: */

oms_node_annot_file_id_asgn(node1, 0);
oms_return = oms_node_update(node1);
if (oms_return != OMS_SUCCEED) {
   printf("Update failed\n");
   return -1;}
```

```
/* Next, oms_node_delete deletes the object from the StP repository: */

oms_return = oms_node_delete (node1);
if (oms_return != OMS_SUCCEED) {
   printf("Delete failed\n");
   return -1}
return 0:
}
```

Using Typed Collections

Typed collection functions perform PDM-oriented operations on homogenous collections of objects in the StP repository. There are two types of collections:

- Collections (coll), used for groups of objects that can be loaded easily into memory
- Big collections (bigcoll), used for groups of object too large to be loaded into memory

Both types of collections have routines that are general and routines that are pdm_type specific. Table 7 describes typed collection functions. Table 8 describes typed big collection functions.

Table 7: Typed Collection Functions

Function	Description
oms_<pdm_type>_coll_create	Creates a collection of oms_<pdm_type>_tp objects. Objects in the repository that satisfy the criteria specified by the query argument are loaded into the memory associated with the collection.
oms_<pdm_type>_coll_get	Returns the *n*th object of a collection. If the *n*th object does not exist, a NULL pointer is returned.

Table 7: Typed Collection Functions (Continued)

Function	Description
oms_<pdm_type>_coll_add	Adds an object of oms_<pdm_type>_tp to a collection.
oms_<pdm_type>_coll_add_unique	Adds an object of oms_<pdm_type>_tp to a collection only if the set does not already contain an object with the same identifying attributes.
oms_<pdm_type>_coll_delete	Searches through the collection for an object with the same memory address as the argument. If one is found it is deleted and OMS_TRUE is returned; otherwise OMS_FALSE is returned.
oms_<pdm_type>_coll_delete_ equivalent	Looks for an object with the same identifying attributes in collection. If one is found, it is deleted from the collection and all memory associated with it is freed. If an equivalent object is found and deleted, OMS_SUCCEED is returned; otherwise OMS_FAIL is returned.
oms_coll_print	Calls oms_<pdm_type>_print for each object in the collection.

Table 8: Typed Big Collection Functions

Function	Description
oms_<pdm_type>_bigcoll_create	Creates a big collection of oms_<pdm_type>_tp objects that satisfy the conditions specified by the query parameter. Big Collections do not load any objects into memory until the ide_<pdm_type>_bigcoll_next function is called.

Table 8: Typed Big Collection Functions (Continued)

Function	Description
oms_<pdm_type>_bigcoll_next	Frees the memory associated with the current object, creates a newly allocated copy of the next object and returns a pointer to the newly created copy. If there are no more objects in the collection, then a NULL pointer is returned. Only one iteration through a set is possible at a time. Repeated iterations can be accomplished by making calls to the oms_bigcoll_reset routine.
oms_bigcoll_reset	Causes a bigcoll to reevaluate the query associated with it, and to begin loading objects from the repository that satisfy the conditions of the query.
oms_bigcoll_print	Calls oms_<pdm_type>_print for each object in the collection.
oms_bigcoll_free	Frees all memory and repository resources associated with the big collection.

Example

This is a sample function to call typed collection functions for a node object.

```
int       sample_func_type_collections(const char *name,
oms_object_type_tp type, oms_object_id_tp scope, const char *signature)

/* This function uses the following nodes, node1 and node2, and creates a
typed collection, node_coll, in which to store them. oms_boolean_tp will
provide an OMS_TRUE or OMS_FALSE value. An integer, i, also will be used
later in the example.*/

{
oms_node_tp *node1, *node2;
oms_coll_tp *node_coll;
```

```
oms_boolean_tp oms_return;
int       i;

/* Next, use oms_node_create to create the two nodes, node1 and node2,
with different identifying attributes: */

node1 = oms_node_create(name,type,scope,signature);
strcat(name,"_xxx");
node2 = oms_node_create(name,type,scope,signature);

/* oms_node_coll_create allocates the collection structure with null
query: */

node_coll = oms_node_coll_create(NULL);

/* oms_node_coll_add_unique adds node1 and node2 to the collection: */

oms_return = oms_node_coll_add_unique(node_coll, node1);
if (oms_return != OMS_SUCCEED) {
       printf("Collection add failed\n");
       return -1}
oms_return = oms_node_coll_add_unique(node_coll, node2);
if (oms_return != OMS_SUCCEED) {
       printf("Collection add failed\n");
       return -1}

/* Now, print each node one by one: */

i = 0;
while (node1 = oms_node_coll_get(node_coll, i++)) {
   oms_node_print(node1, stdout);
   }

/* Print all the nodes at once: */

oms_coll_print(node_coll, stdout);

/* After printing, delete node2 from the collection: */

oms_return = oms_node_coll_delete_equivalent(node_coll, node2);
if (oms_return != OMS_SUCCEED) {
       printf("Collection delete failed\n");
       return -1}
return 0;
}
```

Using Auxiliary Functions

The OMS provides auxiliary functions that help manage the environment:

- Time functions
- Transaction functions
- Id lists
- Error Handling

Time Functions

Time functions manage and manipulate time variables. Table 9 describes time functions.

Table 9: Time Functions

Function	Description
oms_time_input_format_asgn(const char *in_format)	Sets the input format of time strings to the value of “in_format” and returns the old input format (type char*). The input format is used to transform a character string to an oms_time_tp data object. Character string representations of time are used in the OMS query language and in the oms_time_tp functions that translate string representation to the internal representation. This function makes its own copy of in_format and the return value needs to be freed by the calling function.

Table 9: Time Functions (Continued)

Function	Description
oms_time_output_format_asgn(const char *out_format)	Sets the output format of time strings to the value of “out_format” and returns the old output format (type char*). The output format is used wherever a string representation is output. This function makes its own copy of out_format and the return value needs to be freed by the calling function.
oms_time_set_lctime_env(const char *lang)	Sets time to local environment formats. This function sets the LANG environment variable to the value of lang and calls setlocale with the LCTIME category. It returns a copy of the current value of LANG (if any; type char*).
oms_time_to_text(oms_time_tp time)	Converts a time value to a character string representation using the out_format string. A newly allocated character string containing the text representation is returned.
oms_time_text_to_time(const char *time)	Uses the in_format string to convert a character string into an oms_time_tp value.
oms_time_now( )	Returns the current time as an oms_time_tp value.

Transaction Functions

Transaction functions manage StP repository transaction processing. Table 10 lists and describes transaction functions.

Table 10: Transaction Functions

Function	Description
oms_txn_begin();	Begins a transaction and returns oms_status_tp. All modifications to the StP repository after this call has been made and before a corresponding call to oms_txn_commit or oms_txn_begin are part of one transaction. Calls to oms_txn can be nested. When calls to oms_txn_begin are nested, an internal counter is incremented. When calls to oms_txn_commit are made the counter is decreased. Whenever an equivalent number of calls have been made, the transaction is committed and the modifications are made permanently in the StP repository.
oms_txn_commit();	Commits a transaction to the StP repository and returns oms_status_tp. It only has an effect when an equal number of calls have been made to the commit and the begin function.
oms_txn_abort();	Aborts a current transaction, restores the values to their original state, and returns oms_status_tp. No subsequent calls to oms_txn_commit are valid until the next call to oms_txn_begin has been made.

Examples

This is a sample function to call transaction functions for node object.

```
int sample_func_transaction(const char *name, oms_object_type_tp type,
oms_object_id_tp scope, const char *signature)

/* This function uses the following nodes, node1 and node2. oms_boolean_tp
has valid values OMS_TRUE and OMS_FALSE. oms_status_tp contains valid
return values of the OMS update operations. */
{
oms_node_tp *node1, *node2;
oms_boolean_tp oms_bool;
oms_status_tp oms_return;

/* oms_txn_begin signals the start of the transaction: */

oms_txn_begin();

/* Use oms_node_create to create two nodes, node1 and node2, with node2
having a different name from node1: */

node1 = oms_node_create(name,type,scope,signature);
strcat (name, "_xxx")
node2 = oms_node_create(name,type,scope,signature);

/* Next, use oms_node_update to update the StP repository: */

oms_return = oms_node_update(node1);
oms_return = oms_node_update(node2);

/* If node2 is the same as node1, abort the transaction. */

oms_bool = oms_node_equal(node1,node2);
if (oms_bool == OMS_TRUE) {
       oms_txn_abort();
} else {
   oms_txn_commit();}
return 0;
}
```

Id List Functions

An id list is a temporary holder for the results of an OMS query that enable you to reuse the results in other queries within the same application. Table 11 describes the id list functions.

Table 11: Id List Functions

Function	Description
oms_idlist_create (oms_pdm_tp pdm_type, const char *name, const char *query))	Creates an id list of pdm_type with a specified name, for reusing the results of an OMS query.
oms_idlist_free (const char *name)	Frees all the memory and storage engine resources that are associated with an id list.
oms_idlist_free_all ()	Frees all memory and storage engine resources that are associated with all id lists that have been created by an application.

Each id list is scoped to one application and has a name that is assigned to it by that application. The assigned name can only be used by queries that are processed by that application. It is not visible from any other application.

Error Handling Function

The OMS handles error conditions by printing an error number and an error message to the standard output. You can change this behavior by setting an oms_error_handler function in the API.

The function oms_set_oms_err_handler(void (*err_handler) ()) can be called by an application to tell the OMS which function to call when an error occurs.

Define the err_handler function as:

```
void err_handler(oms_boolean_tp is_error, int num, const
char *msg)
```

When an error occurs, the OMS calls the function:

```
oms_error(oms_boolean_tp is_error, int num, const char *msg)
```

This function sets the global variable oms_error_val to num and calls the err_handler routine specified by the client. If none is specified, it calls a default routine that prints the number and message to standard error output.

The error number stored in the global variable oms_error_val represents a general class of errors. The message may give more specific information about the error. Table 12 lists errors generated by the OMS for OMS class messages. Table 13 lists errors relating to the OMS storage manager.

Table 12: General OMS Messages

Message Number	Message
1001	Unable to allocate memory
1002	Attempt to store <PDM type> object with invalid type: <type id>
1003	Attempt to store string attribute with too many characters
1004	Attempt to perform database operation on <PDM type> object from different repository
1005	Error while parsing query
1006	Attempt to create an idlist with an illegal name
1007	Failed to retrieve nodes for scope text
1008	Unable to connect to repository manager
1009	Unable to initialize repository manager
1010	Attempt to use attribute type of "text" in query
1012	SYBASE environment variable is not set

Table 12: General OMS Messages (Continued)

Message Number	Message
1013	Can't open Sybase system file: <path to Sybase interfaces file>. Is your SYBASE environment variable set correctly?
1014	Unable to open System Repository in : <system repository>

Table 13: Storage Manager Messages

Message Number	Message
9001	Internal error: Attempt to re-insert existing <PDM type> object: <object id>
9005	Error while processing results
9006	Unexpected return value from dbresults
9007	Failed to set Sybase DB-Library option DBROWCOUNT
9008	Internal error: Attempt to update <PDM type> object <id> that doesn't exist in the database
9009	Internal error: Attempt to delete <PDM type> object <id> that doesn't exist in the database
9010	Attempt to save object not in cache
9011	This StP installation cannot open systems with repository type <type>
10000	System repository or log is full
10003	Unbalanced call to oms_txn_commit ()
10004	Unbalanced call to oms_txn_abort ()
10005	Unable to locate server in Sybase interfaces file
10006	Not a valid login / password

Table 13: Storage Manager Messages (Continued)

Message Number	Message
10007	You are not a valid user in the current system
10008	Unable to open current repository
10009	Can't connect to Sybase (too many users?)
10010	Too many relationships used in query
10011	User "guest" does not have create table privileges in tempdb—have you executed rmansetup
10017	Internal error: Error while executing SQL: <failed SQL statement>
10018	Unable to allocate new object id
10019	System needs to be upgrade to Core 2.0 before it can be used with this version of StP software
11000	(The message that appears for this error depends on the ODBC error message being generated. See the ODBC documentation for more information.)
11001	Tried to open ODBC database on UNIX
11002	This StP installation cannot open systems of type ODBC

5 Application Program Interface Types and Functions

This chapter provides details about the Application Program Interface (API) library of functions. These functions provide a consistent interface between application programs and the StP repository.

Topics covered are:

- "Data Type / Data Structure Correspondence" on page 5-2
- A description of each function, listed in alphabetical order by data structure name as follows:

oms_app_type_tp	oms_file_tp	oms_object_id_tp
oms_app_type_method_tp	oms_idlist_tp	oms_pdm_type_tp
oms_bigcoll_tp	oms_item_tp	oms_repos_tp
oms_boolean_tp	oms_link_ref_tp	oms_status_tp
oms_cntx_ref_tp	oms_link_tp	oms_time_tp
oms_cntx_tp	oms_method_tp	oms_txn_tp
oms_coll_tp	oms_node_ref_tp	oms_viewpoint_tp
oms_file_hist_tp	oms_node_tp	
oms_file_lock_tp	oms_note_tp	

For information on how to use API functions to manipulate objects in the repository, see Chapter 4, "Using the Application Program Interface."

Data Type/Data Structure Correspondence

Most of the data structures in this chapter correspond to PDM data types, auxiliary data types, or special OMS data types, as described in Chapter 2, “Understanding the Persistent Data Model.”

Table 1 lists the correspondence between these types and the data structures.

Table 1: Data Type/Data Structure Correspondence

PDM Type	Category	Data Structure
	attribute type	oms_app_type_tp
	auxiliary	oms_bigcoll_tp
	auxiliary	oms_boolean_tp
cntx_ref	PDM	oms_cntx_ref_tp
cntx	PDM	oms_cntx_tp
	auxiliary	oms_coll_tp
file_hist	PDM	oms_file_hist_tp
file_lock	PDM	oms_file_lock_tp
file	PDM	oms_file_tp
id list	auxiliary	none
item	PDM	oms_item_tp
link_ref	PDM	oms_link_ref_tp
link	PDM	oms_link_tp
node_ref	PDM	oms_node_ref_tp
node	PDM	oms_node_tp
note	PDM	oms_note_tp
	attribute type	oms_object_id_tp

Table 1: Data Type/Data Structure Correspondence (Continued)

PDM Type	Category	Data Structure
	auxiliary	oms_pdm_type_tp
	auxiliary	oms_repos_id_tp
	auxiliary	oms_repos_tp
	auxiliary	oms_scope_cmp_tp
	auxiliary	oms_status_tp
	attribute type	oms_time_tp
	auxiliary	oms_txn_tp
viewpoint	PDM	oms_viewpoint_tp

For general information about categories of functions and how to call functions from application programs, see Chapter 4, “Using the Application Program Interface.”

oms_app_type_tp

The functions in this section manipulate the application type attribute.

The oms_app_type_to_text function takes a valid PDM object and returns the name of the type.

The oms_app_type_datatype function returns the type of data that the oms_app_type_tp type can contain.

The oms_app_type_from_text function takes the name of a valid PDM application type and returns the oms_app_type_tp value.

The oms_app_type_method function takes a valid PDM object and a command name and an oms_app_type_method_tp object, which contains information about invoking a method on a particular object.

Table 2: oms_app_type_tp Functions

Type	Function
char *	oms_app_type_annot_filename(oms_app_type_tp type)
oms_app_type_tp	oms_app_type_code_get(oms_pdm_type_tp pdm_type, int index)
char *	oms_app_type_code_to_text(oms_app_type_tp type)
char *	oms_app_type_datatype(oms_app_type_tp type)
char *	oms_app_type_comment(oms_app_type_tp type)
oms_app_type_-method_tp	oms_app_type_method_ex(oms_app_type_tp type, const char *name)
oms_pdm_type_tp	oms_app_type_pdm_type(oms_app_type_tp pdm_type)
char *	oms_app_type_printstring(oms_app_type_tp type)
char *	oms_app_type_text_get(oms_pdm_type_tp pdm_type, int index)
char *	oms_app_type_text_get_copy(oms_pdm_type_tp pdm_type, int index)
char *	oms_app_type_text_to_code(oms_pdm_type_tp pdm_type, const char *text)

oms_app_type_method_tp

The functions in this section manipulate the method of the application type attribute.

oms_app_type_method_method accesses the method attribute of type oms_method_tp from the oms_app_type_method_tp structure.

oms_app_type_method_cmd accesses the cmd attribute of type char * of the oms_app_type_method_tp structure.

oms_app_type_method_free frees all resources and memory associated the oms_app_type_method_tp object.

Table 3: oms_app_type_method_tp Functions

Type	Function
oms_method_tp	oms_app_type_method_method(oms_app_type_ method_tp *atm_
char*	oms_app_type_method_cmd(oms_app_type_method_- tp *atm)
void	oms_app_type_method_free(oms_app_type_mthod_tp *atm)

oms_bigcoll_tp

The functions in this section manipulate big collection objects.

Table 4: oms_bigcoll Functions

Type	Function
void	oms_bigcoll_free(oms_bigcoll_tp **coll)
void	oms_bigcoll_print(oms_bigcoll_tp *coll, FILE *fp)
void	oms_bigcoll_reset(oms_bigcoll_tp *coll)

oms_boolean_tp

A number of API functions use the oms_boolean_tp data type. Possible values are false (`OMS_FALSE`) and true (`OMS_TRUE`).

oms_cntx_ref_tp

The functions in this section manipulate cntx ref objects.

Table 5: oms_cntx_ref_tp Creation and Retrieval Functions

Type	Function
oms_cntx_ref_tp *	oms_cntx_ref_create(oms_object_id_tp cntx_id, oms_object_id_tp file_id, const char *appid)
	oms_cntx_ref_find(oms_object_id_tp cntx_id, oms_object_id_tp file_id, const char *appid)
	oms_cntx_ref_find_by_id(oms_object_id_tp id)
	oms_cntx_ref_find_by_query(const char *query)

Table 6: oms_cntx_ref_tp Utility Functions

Type	Function
oms_cntx_ref_tp *	oms_cntx_ref_copy(oms_cntx_ref_tp *cr)
oms_boolean_tp	oms_cntx_ref_equal(oms_cntx_ref_tp *cr1, oms_cntx_ref_tp *cr2)
void	oms_cntx_ref_free(oms_cntx_ref_tp **cr)
	oms_cntx_ref_print(oms_cntx_ref_tp *cr, FILE *fp)
	oms_cntx_ref_print_image_asgn(const char *image)

Table 7: oms_cntx_ref_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_cntx_ref_id (oms_cntx_ref_tp *cr)
oms_object_id_tp	oms_cntx_ref_cntx_id (oms_cntx_ref_tp *cr)
oms_object_id_tp	oms_cntx_ref_file_id(oms_cntx_ref_tp *cr)
char *	oms_cntx_ref_appid (oms_cntx_ref_tp *cr)
oms_object_id_tp	oms_cntx_ref_link_ref_id (oms_cntx_ref_tp *cr)
int	oms_cntx_ref_xcoord (oms_cntx_ref_tp *cr)
int	oms_cntx_ref_ycoord (oms_cntx_ref_tp *cr)
char *	oms_cntx_ref_appid_copy (oms_cntx_ref_tp *cr)
void	oms_cntx_ref_cntx_id_asgn (oms_cntx_ref_tp *cr, oms_object_id_tp cntx_id)
void	oms_cntx_ref_file_id_asgn (oms_cntx_ref_tp *cr, oms_object_id_tp file_id)
void	oms_cntx_ref_appid_asgn(oms_cntx_ref_tp *cr, const char *appid)
void	oms_cntx_ref_link_ref_id_asgn (oms_cntx_ref_tp *cr, oms_object_id_tp link_ref_id)
void	oms_cntx_ref_xcoord_asgn (oms_cntx_ref_tp *cr, int xcoord)
void	oms_cntx_ref_ycoord_asgn (oms_cntx_ref_tp *cr, int ycoord)

Table 8: oms_cntx_ref_tp Repository Management Functions

Type	Function
oms_status_tp	oms_cntx_ref_update(oms_cntx_ref_tp *cr)
	oms_cntx_ref_delete(oms_cntx_ref_tp *cr)

Table 9: oms_cntx_ref_tp Typed Collection Functions

Type	Function
oms_coll_tp*	oms_cntx_ref_coll_create(const char *query)
oms_cntx_ref_tp*	oms_cntx_ref_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_cntx_ref_coll_add(oms_coll_tp *coll, oms_cntx_ref_tp *cr)
	oms_cntx_ref_coll_add_unique(oms_coll_tp *coll, oms_cntx_ref_tp *cr)
	oms_cntx_ref_coll_delete(oms_coll_tp *coll, oms_cntx_ref_tp *cr)
	oms_cntx_ref_coll_delete_equivalent (oms_coll_tp *coll, oms_cntx_ref_tp *cr)

Table 10: oms_cntx_ref_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_cntx_ref_bigcoll_create(const char *query)
oms_cntx_ref_tp*	oms_cntx_ref_bigcoll_next(oms_bigcoll_tp *bcoll)

Table 11: oms_cntx_ref_tp Navigation Functions

Type	Function
oms_cntx_tp*	oms_cntx_ref_cntx(oms_cntx_ref_tp *cr)
oms_file_tp*	oms_cntx_ref_file(oms_cntx_ref_tp *cr)
oms_link_ref_tp*	oms_cntx_ref_link_ref(oms_cntx_ref_tp *cr)

oms_cntx_tp

The functions in this section manipulate cntx objects.

Table 12: oms_cntx_tp Creation and Retrieval Functions

Type	Function
oms_cntx_tp*	oms_cntx_create(const char *name, oms_app_type_tp type, const char *sig, oms_object_id_tp link_id)
	oms_cntx_find (const char *name, oms_app_type_tp type, const char *sig, oms_object_id_tp link_id)
	oms_cntx_find_by_id(oms_object_id_tp id)
	oms_cntx_find_by_query(const char *query)

Table 13: oms_cntx_tp Utility Functions

Type	Function
oms_cntx_tp*	oms_cntx_copy(oms_cntx_tp *cntx)
oms_boolean_tp	oms_cntx_equal(oms_cntx_tp *cntx1, oms_cntx_tp *cntx2)

Table 13: oms_cntx_tp Utility Functions (Continued)

Type	Function
void	oms_cntx_free(oms_cntx_tp **cntx)
	oms_cntx_print(oms_cntx_tp *cntx, FILE *fp)
	oms_cntx_print_image_asgn(const char *image)

Table 14: oms_cntx_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_cntx_id (oms_cntx_tp *cntx)
char *	oms_cntx_name (oms_cntx_tp *cntx)
oms_app_type_tp	oms_cntx_type(oms_cntx_tp *cntx)
char *	oms_cntx_sig(oms_cntx_tp *cntx)
oms_object_id_tp	oms_cntx_link_id(oms_cntx_tp *cntx)
oms_object_id_tp	oms_cntx_scope_node_id (oms_cntx_tp *cntx)
oms_object_id_tp	oms_cntx_annot_file_id (oms_cntx_tp *cntx)
char *	oms_cntx_name_copy (oms_cntx_tp *cntx)
char *	oms_cntx_sig_copy (oms_cntx_tp *cntx)
void	oms_cntx_id_asgn (oms_cntx_tp *cntx, oms_object_id_tp id)
void	oms_cntx_name_asgn (oms_cntx_tp *cntx, const char *name)
void	oms_cntx_type_asgn(oms_cntx_tp *cntx, oms_app_type_tp type)
void	oms_cntx_sig_asgn (oms_cntx_tp *cntx, const char *sig)

Table 14: oms_cntx_tp Attribute Access and Assign Functions (Continued)

Type	Function
void	oms_cntx_link_id_asgn (oms_cntx_tp *cntx, oms_object_id_tp link_id)
void	oms_cntx_scope_node_id_asgn (oms_cntx_tp *cntx, oms_object_id_tp scope_node_id)
void	oms_cntx_annot_file_id_asgn (oms_cntx_tp *cntx, oms_object_id_tp annot_file_id)

Table 15: oms_cntx_tp Repository Management Functions

Type	Function
oms_status_tp	oms_cntx_update(oms_cntx_tp *cntx)
	oms_cntx_delete(oms_cntx_tp *cntx)
	oms_cntx_check(oms_cntx_tp *obj)

Table 16: oms_cntx_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_cntx_coll_create(const char *query)
oms_cntx_tp*	oms_cntx_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_cntx_coll_add(oms_coll_tp *coll, oms_cntx_tp *cntx)
	oms_cntx_coll_add_unique(oms_coll_tp *coll, oms_cntx_tp *cntx)
	oms_cntx_coll_delete(oms_coll_tp *coll, oms_cntx_tp *cntx)
	oms_cntx_coll_delete_equivalent(oms_coll_tp *coll, oms_cntx_tp *cntx)

Table 17: oms_cntx_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp*	oms_cntx_bigcoll_create(const char *query)
oms_cntx_ref_tp *	oms_cntx_bigcoll_next(oms_bigcoll_tp *bigcoll)

Table 18: oms_cntx_tp Navigation Functions

Type	Function
oms_link_tp *	oms_cntx_link(oms_cntx_tp *cntx)
oms_node_tp *	oms_cntx_scope_node(oms_cntx_tp *cntx)
oms_coll_tp *	oms_cntx_notes(oms_cntx_tp *cntx)
oms_coll_tp *	oms_cntx_cntx_refs(oms_cntx_tp *cntx)

oms_coll_tp

Typed Collection functions use the oms_coll_tp structure. The data members of this type should not interpreted by an application program.

The oms_coll_add function adds any PDM object to coll. The routine returns OMS_SUCCEED if the object was successfully added, or OMS_FAIL otherwise.

The oms_coll_delete function compares the pointer to the pointers in coll. If an identical pointer is found, the pointer is deleted from the collection. It does not free any memory related to the object and references to the object are still valid after it has been deleted from the collection.

The oms_coll_free function frees all resources and memory associated with a collection and sets the pointer to the collection to NULL. It does not free any memory associated with the objects it contains. oms_collection_free_all frees all memory associated with each object in

the collection and then frees all memory associated with the collection itself.

Table 19: oms_coll_tp Functions

Type	Function
int	oms_coll_count(oms_coll_tp *coll)
void	oms_coll_free(oms_coll_tp **coll)
void	oms_coll_free_all(oms_coll_tp **coll)
void	oms_coll_print(oms_coll_tp *coll, FILE *fp)

oms_file_hist_tp

The functions in this section manipulate file history objects.

To remove file history objects that have accumulated over time, use the SysTruncHist system function, described in *Query and Reporting System*.

Table 20: oms_file_hist_tp Creation and Retrieval Functions

Type	Function
oms_file_hist_tp *	oms_file_hist_create(oms_app_type_tp type, oms_object_id_tp file_id, oms_time_tp time, const char *user, const char *hostname)
	oms_file_hist_find(oms_object_id_tp file_id, oms_app_type_tp type, oms_time_tp time, const char *user, const char *hostname)
	oms_file_hist_find_by_id(oms_object_id_tp id)
	oms_file_hist_find_by_query(const char *query)

Table 21: oms_file_hist_tp Utility Functions

Type	Function
oms_file_hist_tp *	oms_file_hist_copy(oms_file_hist_tp *fh)
oms_boolean_tp	oms_file_hist_equal(oms_file_hist_tp *fh1, oms_file_hist_tp *fh2)
void	oms_file_hist_free(oms_file_hist_tp **fh)
	oms_file_hist_print(oms_file_hist_tp *fh, FILE *fp)
	oms_file_hist_print_image_asgn(const char *image)

Table 22: oms_file_hist_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_file_hist_id (oms_file_hist_tp *fh)
oms_object_id_tp	oms_file_hist_file_id (oms_file_hist_tp *fh)
oms_app_type_tp	oms_file_hist_type (oms_file_hist_tp *fh)
oms_time_tp	oms_file_hist_time (oms_file_hist_tp *fh)
char *	oms_file_hist_user (oms_file_hist_tp *fh)
char *	oms_file_hist_hostname (oms_file_hist_tp *fh)
char *	oms_file_hist_hostname _copy(oms_file_hist_tp *fh)
char *	oms_file_hist_user_copy(oms_file_hist_tp *fh)
void	oms_file_hist_id_asgn (oms_file_hist_tp *fh, oms_object_id_tp id)
void	oms_file_hist_file_id_asgn (oms_file_hist_tp *fh, oms_object_id_tp *file_id)
void	oms_file_hist_type_asgn(oms_file_hist_tp *fh, oms_app_type_tp type)

Table 22: oms_file_hist_tp Attribute Access and Assign Functions (Continued)

Type	Function
void	oms_file_hist_time_asgn (oms_file_hist_tp *fh, oms_time_tp time)
void	oms_file_hist_user_asgn (oms_file_hist_tp *fh, const char *user)
void	oms_file_hist_hostname_asgn (oms_file_hist_tp *fh, const char *hostname)

Table 23: oms_file_hist_tp Repository Management Functions

Type	Function
oms_status_tp	oms_file_hist_update(oms_file_hist_tp *fh)
	oms_file_hist_delete(oms_file_hist_tp *fh)
	oms_file_hist_ check(oms_file_hist_ tp *obj)

Table 24: oms_file_hist_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_file_hist_coll_create(const char *query)
oms_file_hist_tp *	oms_file_hist_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_file_hist_coll_add(oms_coll_tp *coll, oms_file_hist_tp *fh)
	oms_file_hist_coll_add_unique(oms_coll_tp *coll, oms_file_hist_tp *fh)
	oms_file_hist_coll_delete(oms_coll_tp *coll, oms_file_hist_tp *fh)
	oms_file_hist_coll_delete_equivalent (oms_coll_tp *coll, oms_file_hist_tp *fh)

Table 25: oms_file_hist_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_file_hist_bigcoll_create(const char *query)
oms_file_hist_tp *	oms_file_hist_bigcoll_next(oms_bigcoll_tp *bigcoll)

Table 26: oms_file_hist_tp Navigation Function

Type	Function
oms_file_tp *	oms_file_hist_file(oms_file_hist_tp *fh)

oms_file_lock_tp

The functions in this section manipulate file lock objects.

Table 27: oms_file_lock_tp Creation and Retrieval Functions

Type	Function
oms_file_lock_tp *	oms_file_lock_create(oms_object_id_tp file_id, oms_app_type_tp type)
	oms_file_lock_find(oms_object_id_tp file_id, oms_app_type_tp type)
	oms_file_lock_find_by_id(oms_object_id_tp id)
	oms_file_lock_find_by_query(const char *query)

Table 28: oms_file_lock_tp Utility Functions

Type	Function
oms_file_lock_tp *	oms_file_lock_copy(oms_file_lock_tp *fl)
oms_boolean_tp	oms_file_lock_equal(oms_file_lock_tp *fl1, oms_file_lock_tp *fl2)
void	oms_file_lock_free(oms_file_lock_tp **fl)
	oms_file_lock_print(oms_file_lock_tp *fl, FILE *fp)
	oms_file_lock_print_image_asgn(const char *image)

Table 29: oms_file_lock_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_file_lock_id (oms_file_lock_tp *fl)
oms_object_id_tp	oms_file_lock_file_id (oms_file_lock_tp *fl)
oms_app_type_tp	oms_file_lock_type (oms_file_lock_tp *fl)
char *	oms_file_lock_user (oms_file_lock_tp *fl)
char *	oms_file_lock_accessors (oms_file_lock_tp *fl)
char *	oms_file_lock_accessors_copy (oms_file_lock_tp *fl)
char *	oms_file_lock_destroyers (oms_file_lock_tp *fl)
char *	oms_file_lock_destroyers_copy (oms_file_lock_tp *fl)
char *	oms_file_lock_user_copy (oms_file_lock_tp *fl)
void	oms_file_lock_id_asgn (oms_file_lock_tp *fl, oms_object_id_tp id)
void	oms_file_lock_file_id_asgn (oms_file_lock_tp *fl, oms_object_id_tp *file_id)

Table 29: oms_file_lock_tp Attribute Access and Assign Functions (Continued)

Type	Function
void	oms_file_lock_type_asgn(oms_file_lock_tp *fl, oms_app_type_tp type)
void	oms_file_lock_user_asgn (oms_file_lock_tp *fl, const char *user)
void	oms_file_lock_accessors_asgn (oms_file_lock_tp *fl, const char *accessors)
void	oms_file_lock_destroyers_asgn (oms_file_lock_tp *fl, const char *destroyers)

Table 30: oms_file_lock_tp Repository Management Functions

Type	Function
oms_status_tp	oms_file_lock_update(oms_file_lock_tp *fl)
	oms_file_lock_delete(oms_file_lock_tp *fl)
	oms_file_lock_check(oms_file_lock_tp *obj)

Table 31: oms_file_lock_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_file_lock_coll_create(const char *query)
oms_file_lock_tp *	oms_file_lock_coll_get(oms_coll_tp *coll, int i)

Table 31: oms_file_lock_tp Typed Collection Functions (Continued)

Type	Function
oms_status_tp	oms_file_lock_coll_add(oms_coll_tp *coll, oms_file_lock_tp *fl)
	oms_file_lock_coll_add_unique(oms_coll_tp *coll, oms_file_lock_tp *fl)
	oms_file_lock_coll_delete(oms_coll_tp *coll, oms_file_lock_tp *fl)
	oms_file_lock_coll_delete_equivalent (oms_coll_tp *coll, oms_file_lock_tp *fl)

Table 32: oms_file_lock_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_file_lock_bigcoll_create(const char *query)
oms_file_lock_tp *	oms_file_lock_bigcoll_next(oms_coll_tp *bigcoll)

Table 33: oms_file_lock_tp Navigation Function

Type	Function
oms_file_tp *	oms_file_lock_file(oms_file_lock_tp *fl)

oms_file_tp

The functions in this section manipulate file objects.

Table 34: oms_file_tp Creation and Retrieval Functions

Type	Function
oms_file_tp *	oms_file_create(const char *name, oms_app_type_tp type)
	oms_file_find(const char *name, oms_app_type_tp type)
	oms_file_find_by_id(oms_object_id_tp id)
	oms_file_find_by_query(const char *query)

Table 35: oms_file_tp Utility Functions

Type	Function
oms_file_tp *	oms_file_copy(oms_file_tp *file)
oms_boolean_tp	oms_file_equal(oms_file_tp *file1, oms_file_tp *file2)
void	oms_file_free(oms_file_tp **file)
	oms_file_print(oms_file_tp *file, FILE *fp)
	oms_file_print_image_asgn(const char *image)

Table 36: oms_file_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_file_id (oms_file_tp *file)
char *	oms_file_name (oms_file_tp *file)
oms_app_type_tp	oms_file_type (oms_file_tp *file)
char *	oms_file_rev (oms_file_tp *file)

Table 36: oms_file_tp Attribute Access and Assign Functions (Continued)

Type	Function
char *	oms_file_fname (oms_file_tp *file)
char *	oms_file_fname_copy (oms_file_tp *file)
oms_object_id_tp	oms_file_lview_id (oms_file_tp *file)
oms_object_id_tp	oms_file_lmdfy_id (oms_file_tp *file)
oms_object_id_tp	oms_file_lsync_id (oms_file_tp *file)
oms_object_id_tp	oms_file_annot_file_id (oms_file_tp *file)
char *	oms_file_name_copy (oms_file_tp *file)
char *	oms_file_rev_copy (oms_file_tp *file)
void	oms_file_id_asgn (oms_file_tp *file, oms_object_id_tp id)
void	oms_file_name_asgn (oms_file_tp *file, const char *name)
void	oms_file_type_asgn(oms_file_tp *file, oms_app_type_tp type)
void	oms_file_rev_asgn (oms_file_tp *file, const char *rev)
void	oms_file_fname_asgn (oms_file_tp *file, const char *fname)
void	oms_file_lview_id_asgn (oms_file_tp *file, oms_object_id_tp lview_id)
void	oms_file_lmdfy_id_asgn (oms_file_tp *file, oms_object_id_tp lmdfy_id)
void	oms_file_lsync_id_asgn (oms_file_tp *file, oms_object_id_tp lsync_id)
void	oms_file_annot_file_id_asgn (oms_file_tp *file, oms_object_id_tp annot_file_id)

Table 37: oms_file_tp Repository Management Functions

Type	Function
oms_status_tp	oms_file_update(oms_file_tp *file)
	oms_file_delete(oms_file_tp *file)
	oms_file_check(oms_file_tp *obj)

Table 38: oms_file_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_file_coll_create(const char *query)
oms_file_tp *	oms_file_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_file_coll_add(oms_coll_tp *coll, oms_file_tp *file)
	oms_file_coll_add_unique(oms_coll_tp *coll, oms_file_tp *file)
	oms_file_coll_delete(oms_coll_tp *coll, oms_file_tp *file)
	oms_file_coll_delete_equivalent (oms_coll_tp *coll, oms_file_tp *file)

Table 39: oms_file_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_file_bigcoll_create(const char *query)
oms_file_tp *	oms_file_bigcoll_next(oms_coll_tp *bigcoll)

Table 40: oms_file_tp Navigation Functions

Type	Function
oms_file_hist_tp *	oms_file_lview_hist(oms_file_tp *file)
oms_file_hist_tp *	oms_file_lmdfy_hist(oms_file_tp *file)
oms_file_hist_tp *	oms_file_lsync_hist(oms_file_tp *file)
oms_file_tp *	oms_file_annot_file(oms_file_tp *file)
oms_coll_tp *	oms_file_notes(oms_file_tp *file)
	oms_file_node_refs(oms_file_tp *file)
	oms_file_link_refs(oms_file_tp *file)
	oms_file_cntx_refs(oms_file_tp *file)
	oms_file_file_hists(oms_file_tp *file)
	oms_file_file_locks(oms_file_tp *file)

oms_idlist_tp

The functions in this section manage id lists.

Table 41: oms_idlist_tp Functions

Type	Function
oms_status_tp	oms_idlist_create(oms_pdm_tp pdm_type, const char *name, const char *query
void	oms_idlist_free (const char *name)
void	oms_idlist_free_all ()

oms_item_tp

The functions in this section manipulate item objects.

Table 42: oms_item_tp Creation and Retrieval Functions

Type	Function
oms_item_tp *	oms_item_create(oms_object_id_tp note_id, oms_app_type_tp type, const char *value)
	oms_item_find(oms_object_id_tp note_id, oms_app_type_tp type, const char *value)
	oms_item_find_by_id(oms_object_id_tp id)
	oms_item_find_by_query(const char *query)

Table 43: oms_item_tp Utility Functions

Type	Function
oms_item_tp *	oms_item_copy(oms_item_tp *item)
oms_boolean_tp	oms_item_equal(oms_item_tp *item1, oms_item_tp*item2)
void	oms_item_free(oms_item_tp **item)
	oms_item_print(oms_item_tp *item, FILE *fp)
	oms_item_print_image_asgn(const char *image)

Table 44: oms_item_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_item_id (oms_item_tp *item)

Table 44: oms_item_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_item_note_id (oms_item_tp *item)
oms_app_type_tp	oms_item_type (oms_item_tp *item)
oms_object_id_tp	oms_item_obj_id(oms_item_tp *item)
char *	oms_item_value (oms_item_tp *item)
char *	oms_item_value_copy (oms_item_tp *item)
void	oms_item_obj_id_asgn (oms_item_tp *item, oms_object_id_tp *node_id)
void	oms_item_note_id_asgn (oms_item_tp *item, oms_object_id_tp *note_id)
void	oms_item_type_asgn (oms_item_tp *item, oms_app_type_tp type)
void	oms_item_obj_id_asgn(oms_item_tp *item)
void	oms_item_value_asgn (oms_item_tp *item, const char *value)

Table 45: oms_item_tp Repository Management Functions

Type	Function
oms_status_tp	oms_item_update(oms_item_tp *item)
	oms_item_delete(oms_item_tp *item)
	oms_item_check(oms_item_tp *obj)

Table 46: oms_item_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_item_coll_create(const char *query)
oms_item_tp *	oms_item_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_item_coll_add(oms_coll_tp *coll, oms_item_tp *item)
	oms_item_coll_add_unique(oms_coll_tp *coll, oms_item_tp *item)
	oms_item_coll_delete(oms_coll_tp *coll, oms_item_tp *item)
	oms_item_coll_delete_equivalent(oms_coll_tp *coll, oms_item_tp *item)

Table 47: oms_item_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_create *	oms_item_bigcoll_create(const char *query)
oms_item_tp *	oms_item_bigcoll_next(oms_coll_tp *bigcoll)

Table 48: oms_item_tp Navigation Functions

Type	Function
oms_note_tp *	oms_item_note(oms_item_tp *item)
oms_file_tp *	oms_item_file(oms_item_tp *item)
oms_node_tp *	oms_item_node(oms_item_tp *item)
oms_link_tp *	oms_item_link(oms_item_tp *item)
oms_cntx_tp *	oms_item_cntx(oms_item_tp *item)

oms_link_ref_tp

The functions in this section manipulate link ref objects.

Table 49: oms_link_ref_tp Creation and Retrieval Functions

Type	Function
oms_link_ref_tp *	oms_link_ref_create(oms_object_id_tp link_id, oms_object_id_tp file_id, const char *appid, oms_object_id_tp from_node_ref_id, oms_object_id_tp to_node_ref_id)
	oms_link_ref_find(oms_object_id_tp link_id, oms_object_id_tp file_id, const char *appid, oms_object_id_tp from_node_ref_id, oms_object_id_tp to_node_ref_id)
	oms_link_ref_find_by_id(oms_object_id_tp id)
	oms_link_ref_find_by_query(const char *query)

Table 50: oms_link_ref_tp Utility Functions

Type	Function
oms_link_ref_tp *	oms_link_ref_copy(oms_link_ref_tp *lr)
oms_boolean_tp	oms_link_ref_equal(oms_link_ref_tp *lr1, oms_link_ref_tp *lr2)
void	oms_link_ref_free(oms_link_ref_tp **lr)
	oms_link_ref_print(oms_link_ref_tp *lr, FILE *fp)
	oms_link_ref_print_image_asgn(const char *image)

Table 51: oms_link_ref_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_link_ref_id (oms_link_ref_tp *lr)
oms_object_id_tp	oms_link_ref_link_id (oms_link_ref_tp *lr)
oms_object_id_tp	oms_link_ref_file_id (oms_link_ref_tp *lr)
char *	oms_link_ref_appid (oms_link_ref_tp *lr)
char *	oms_link_ref_appid_copy (oms_link_ref_tp *lr)
oms_object_id_tp	oms_link_ref_from_node_ref_id (oms_link_ref_tp *lr)
oms_object_id_tp	oms_link_ref_to_node_ref_id (oms_link_ref_tp *lr)
int	oms_link_ref_xcoord (oms_link_ref_tp *lr)
int	oms_link_ref_ycoord (oms_link_ref_tp *lr)
void	oms_link_ref_id_asgn (oms_link_ref_tp *lr, oms_object_id_tp id)
void	oms_link_ref_link_id_asgn (oms_link_ref_tp *lr, oms_object_id_tp *link_id)
void	oms_link_ref_file_id_asgn (oms_link_ref_tp *lr, oms_object_id_tp *file_id)
void	oms_link_ref_appid_asgn (oms_link_ref_tp *lr, const char *appid)
void	oms_link_ref_from_node_ref_id _asgn(oms_link_ref_tp *lr, oms_object_id_tp *from_node_ref_id)
void	oms_link_ref_to_node_ref_id _asgn (oms_link_ref_tp *lr, oms_object_id_tp *to_node_ref_id)

Table 51: oms_link_ref_tp Attribute Access and Assign Functions (Continued)

Type	Function
void	oms_link_ref_xcoord_asgn (oms_link_ref_tp *lr, int xcoord)
void	oms_link_ref_ycoord_asgn (oms_link_ref_tp *lr, int ycoord)

Table 52: oms_link_ref_tp Repository Management Functions

Type	Function
oms_status_tp	oms_link_ref_update(oms_link_ref_tp *lr)
	oms_link_ref_delete(oms_link_ref_tp *lr)
	oms_link_ref_check(oms_link_ref_tp *obj)

Table 53: oms_link_ref_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_link_ref_coll_create(const char *query)
oms_link_ref_tp *	oms_link_ref_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_link_ref_coll_add(oms_coll_tp *coll, oms_link_ref_tp *lr)
	oms_link_ref_coll_add_unique(oms_coll_tp *coll, oms_link_ref_tp *lr)
	oms_link_ref_coll_delete(oms_coll_tp *coll, oms_link_ref_tp *lr)
	oms_link_ref_coll_delete_equivalent(oms_coll_tp *coll, oms_link_ref_tp *lr)

Table 54: oms_link_ref_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_link_ref_bigcoll_create(const char *query)
oms_link_ref_tp *	oms_link_ref_bigcoll_next(oms_bigcoll_tp *bigcoll)

Table 55: oms_link_ref_tp Navigation Functions

Type	Function
oms_link_tp *	oms_link_ref_link(oms_link_ref_tp *lr)
	oms_link_ref_link(oms_link_ref_tp *lr)
oms_file_tp *	oms_link_ref_file(oms_link_ref_tp *lr)
oms_node_ref_tp *	oms_link_ref_from_node_ref(oms_link_ref_tp *lr)
	oms_link_ref_to_node_ref(oms_link_ref_tp *lr)
oms_coll_tp *	oms_link_ref_cntx_refs(oms_link_ref_tp *lr)

oms_link_tp

The functions in this section manipulate link objects.

Table 56: oms_link_tp Creation and Retrieval Functions

Type	Function
oms_link_tp *	oms_link_create(const char *name, oms_app_type_tp type, oms_object_id_tp from_node_id, oms_object_id_tp to_node_id, const char *sig)
	oms_link_find(const char *name, oms_app_type_tp type, oms_object_id_tp from_node_id, oms_object_id_tp to_node_id, const char *sig)
	oms_link_find_by_id(oms_object_id_tp id)
	oms_link_find_by_query(const char *query)

Table 57: oms_link_tp Utility Functions

Type	Function
oms_link_tp *	oms_link_copy(oms_link_tp *link)
oms_boolean_tp	oms_link_equal(oms_link_tp *link1, oms_link_tp *link2)
void	oms_link_free(oms_link_tp **link)
	oms_link_print(oms_link_tp *link, FILE *fp)
	oms_link_print_image_asgn(const char *image)

Table 58: oms_link_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_link_id (oms_link_tp *link)
char *	oms_link_name (oms_link_tp *link)

Table 58: oms_link_tp Attribute Access and Assign Functions (Continued)

Type	Function
oms_app_type_tp	oms_link_type (oms_link_tp *link)
oms_object_id_tp	oms_link_from_node_id (oms_link_tp *link)
oms_object_id_tp	oms_link_to_node_id (oms_link_tp *link)
char *	oms_link_sig (oms_link_tp *link)
oms_object_id_tp	oms_link_scope_node_id (oms_link_tp *link)
oms_object_id_tp	oms_link_annot_file_id (oms_link_tp *link)
char *	oms_link_name_copy (oms_link_tp *link)
char *	oms_link_sig_copy (oms_link_tp *link)
void	oms_link_id_asgn (oms_link_tp *link, oms_object_id_tp id)
void	oms_link_name_asgn (oms_link_tp *link, constr char *name)
void	oms_link_type_asgn (oms_link_tp *link, oms_app_type_tp *type)
void	oms_link_from_node_id_asgn (oms_link_tp *link, oms_object_id_tp from_node_id)
void	oms_link_to_node_id_asgn (oms_link_tp *link, oms_object_id_tp to_node_id)
void	oms_link_sig _asgn (oms_link_tp *link, constr char *sig)
void	oms_link_scope_node_id _asgn (oms_link_tp *link, oms_object_id_tp *scope_node_id)
void	oms_link_annot_file_id _asgn (oms_link_tp *link, oms_object_id_tp *annot_file_id)

Table 59: oms_link_tp Repository Management Functions

Type	Function
oms_status_tp	oms_link_update(oms_link_tp *link)
	oms_link_delete(oms_link_tp *link)
	oms_link_check(oms_link_tp *obj)

Table 60: oms_link_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_link_coll_create(const char *query)
oms_link_tp *	oms_link_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_link_coll_add(oms_coll_tp *coll, oms_link_tp *link)
	oms_link_coll_add_unique(oms_coll_tp *coll, oms_link_tp *link)
	oms_link_coll_delete(oms_coll_tp *coll, oms_link_tp *link)
	oms_link_coll_delete_equivalent(oms_coll_tp *coll, oms_link_tp *link)

Table 61: oms_link_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_link_bigcoll_create(const char *query)
oms_link_tp *	oms_link_bigcoll_next(oms_bigcoll_tp *bigcoll)

Table 62: oms_link_tp Navigation Functions

Type	Function
oms_node_tp *	oms_link_from_node(oms_link_tp *link)
	oms_link_to_node(oms_link_tp *link)
	oms_link_scope_node(oms_link_tp *link)
oms_file_tp *	oms_link_annot_file(oms_link_tp *link)
oms_coll_tp *	oms_link_notes(oms_link_tp *link)
	oms_link_link_refs(oms_link_tp *link)
	oms_link_cntxs(oms_link_tp *link)

oms_method_tp

The oms_method_tp auxiliary data type returns information to the user about the method they referred to. The information can be in the form of a command string, with value OMS_CMD_STRING, or in the form of a built-in function, with value OMS_BUILTIN. There are no functions for the oms_method_tp auxiliary data type.

oms_node_ref_tp

The functions in this section manipulate node ref objects.

Table 63: oms_node_ref_tp Creation and Retrieval Functions

Type	Function
oms_node_ref_tp *	oms_node_ref_create(oms_object_id_tp node_id, oms_object_id_tp file_id, const char *appid)
	oms_node_ref_find(oms_object_id_tp node_id, oms_object_id_tp file_id, const char *appid)
	oms_node_ref_find_by_id(oms_object_id_tp id)
	oms_node_ref_find_by_query(const char *query)

Table 64: oms_node_ref_tp Utility Functions

Type	Function
oms_node_ref_tp *	oms_node_ref_copy(oms_node_ref_tp *nr)
oms_boolean_tp	oms_node_ref_equal(oms_node_ref_tp *nr1, oms_node_ref_tp *nr2)
void	oms_node_ref_free(oms_node_ref_tp **nr)
	oms_node_ref_print(oms_node_ref_tp *nr, FILE *fp)
	oms_node_ref_print_image_asgn(const char *image)

Table 65: oms_node_ref_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_node_ref_id (oms_node_ref_tp *nr)
oms_object_id_tp	oms_node_ref_node_id (oms_node_ref_tp *nr)
oms_object_id_tp	oms_node_ref_file_id (oms_node_ref_tp *nr)

Table 65: oms_node_ref_tp Attribute Access and Assign Functions (Continued)

Type	Function
char *	oms_node_ref_appid (oms_node_ref_tp *nr)
char *	oms_node_ref_appid_copy (oms_node_ref_tp *nr)
oms_object_id_tp	oms_node_ref_scope_node_ref_id (oms_node_ref_tp *nr)
int	oms_node_ref_xcoord(oms_node_ref_tp *nr)
int	oms_node_ref_ycoord(oms_node_ref_tp *nr)
void	oms_node_ref_id_asgn (oms_node_ref_tp *nr, oms_object_id_tp id)
void	oms_node_ref_node_id_asgn (oms_node_ref_tp oms_object_id_tp *nr, *node_id)
void	oms_node_ref_file_id_asgn (oms_node_ref_tp *nr, oms_app_type_tp *file_id)
void	oms_node_ref_appid_asgn (oms_node_ref_tp *nr, const char *appid)
void	oms_node_ref_scope_node_ref_id_asgn (oms_node_ref_tp *nr, oms_object_id_tp scope_node_ref_id)
void	oms_node_ref_xcoord_asgn(oms_node_ref_tp *nr, int xcoord)
void	oms_node_ref_ycoord_asgn(oms_node_ref_tp *nr, int xcoord)

Table 66: oms_oms_node_ref_tp Repository Management Functions

Type	Function
oms_status_tp	oms_node_ref_update(oms_node_ref_tp *nr)
	oms_node_ref_delete(oms_node_ref_tp *nr)
	oms_node_ref_check(oms_node_ref_tp *obj)

Table 67: oms_node_ref_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_node_ref_coll_create(const char *query)
oms_node_ref_tp *	oms_node_ref_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_node_ref_coll_add(oms_coll_tp *coll, oms_node_ref_tp *nr)
	oms_node_ref_coll_add_unique(oms_coll_tp *coll, oms_node_ref_tp *nr)
	oms_node_ref_coll_delete(oms_coll_tp *coll, oms_node_ref_tp *nr)
	oms_node_ref_coll_delete_equivalent(oms_coll_tp *coll, oms_node_ref_tp *nr)

Table 68: oms_node_ref_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_node_ref_bigcoll_create(const char *query)
oms_node_ref_tp *	oms_node_ref_bigcoll_next(oms_coll_tp *bigcoll)

Table 69: oms_node_ref_tp Navigation Functions

Type	Function
oms_node_tp *	oms_node_ref_node(oms_node_ref_tp *nr)
oms_file_tp *	oms_node_ref_file(oms_node_ref_tp *nr)
oms_node_ref_tp *	oms_node_ref_scope_node_ref(oms_node_ref_tp *nr)
oms_coll_tp *	oms_node_ref_viewpoints(oms_node_ref_tp *nr)
	oms_node_ref_scoped_node_refs(oms_node_ref_tp *nr)
	oms_node_ref_out_link_refs(oms_node_ref_tp *nr)
	oms_node_ref_in_link_refs(oms_node_ref_tp *nr)

oms_node_tp

The functions in this section manipulate node objects. A node object can be identified by its name, type, scope, and signature. It can also be the scope parent of other objects and the endpoint of a link.

Table 70: oms_node_tp Creation and Retrieval Functions

Type	Function
oms_node_tp *	oms_node_create(const char *name, oms_app_type_tp type, oms_object_id_tp scope_node_id, const char *sig)
	oms_node_find(const char *name, oms_app_type_tp type, oms_object_id_tp scope_node_id, const char *sig)
	oms_node_find_by_id(oms_object_id_tp id)
	oms_node_find_by_query(const char *query)

Table 71: oms_node_tp Utility Functions

Type	Function
oms_node_tp *	oms_node_copy(oms_node_tp *node)
oms_boolean_tp	oms_node_equal(oms_node_tp *node1, oms_node_tp *node2)
void	oms_node_free(oms_node_tp **node)
	oms_node_print(oms_node_tp *node, FILE *fp)
	oms_node_print_image_asgn(const char *image)

Table 72: oms_node_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_node_id (oms_node_tp *node)
char *	oms_node_name (oms_node_tp *node)
oms_app_type_tp	oms_node_type (oms_node_tp *node)

Table 72: oms_node_tp Attribute Access and Assign Functions (Continued)

Type	Function
oms_object_id_tp	oms_node_scope_node_id (oms_node_tp *node)
char *	oms_node_sig (oms_node_tp *node)
oms_object_id_tp	oms_node_annot_file_id (oms_node_tp *node)
char *	oms_node_name_copy (oms_node_tp *node)
char *	oms_node_sig_copy (oms_node_tp *node)
void	oms_node_name_asgn (oms_node_tp *node, const char *name)
void	oms_node_type_asgn (oms_node_tp *node, oms_app_type_tp *type)
void	oms_node_scope_node_id_asgn (oms_node_tp *node, oms_object_id_tp scope_node_id)
void	oms_node_sig_asgn (oms_node_tp *node, const char *sig)
void	oms_node_annot_file_id_asgn (oms_node_tp *node, oms_object_id_tp annot_file_id)

Table 73: oms_node_tp Repository Management Functions

Type	Function
oms_status_tp	oms_node_update(oms_node_tp *node)
	oms_node_delete(oms_node_tp *node)
	oms_node_check(oms_node_tp *obj)

Table 74: oms_node_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_node_coll_create(const char *query)
oms_node_tp *	oms_node_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_node_coll_add(oms_coll_tp *coll, oms_node_tp *node)
	oms_node_coll_add_unique(oms_coll_tp *coll, oms_node_tp *node)
	oms_node_coll_delete(oms_coll_tp *coll, oms_node_tp *node)
	oms_node_coll_delete_equivalent(oms_coll_tp *coll, oms_node_tp *node)

Table 75: oms_node_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_node_bigcoll_create(const char *query)
oms_node_tp *	oms_node_bigcoll_next(oms_coll_tp *bigcoll)

Table 76: oms_node_tp Navigation Functions

Type	Function
oms_node_tp *	oms_node_scope_node(oms_node_tp *node)
oms_file_tp *	oms_node_annot_file(oms_node_tp *node)

Table 76: oms_node_tp Navigation Functions (Continued)

Type	Function
oms_coll_tp *	oms_node_notes(oms_node_tp *node)
	oms_node_viewpoints(oms_node_tp *node)
	oms_node_scoped_nodes(oms_node_tp *node)
	oms_node_scoped_links(oms_node_tp *node)
	oms_node_scoped_cntxs(oms_node_tp *node)
	oms_node_out_links(oms_node_tp *node)
	oms_node_in_links(oms_node_tp *node)
	oms_node_node_refs(oms_node_tp *node)

oms_note_tp

The functions in this section manipulate note objects.

Table 77: oms_note_tp Creation and Retrieval Functions

Type	Function
oms_note_tp *	oms_note_create(oms_object_id_tp obj_id, oms_app_type_tp type, const char *name)
	oms_node_find(oms_object_id_tp obj_id, oms_app_type_tp type, const char *name)
	oms_note_find_by_id(oms_object_id_tp id)
	oms_node_find_by_query(const char *query)

Table 78: oms_note_tp Utility Functions

Type	Function
oms_note_tp *	oms_note_copy(oms_note_tp *note)
oms_boolean_tp	oms_note_equal(oms_note_tp *note1, oms_note_tp *note2)
void	oms_note_free(oms_note_tp **note)
	oms_note_print(oms_note_tp *note, FILE *fp)
	oms_note_print_image_asgn(const char *image)

Table 79: oms_note_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_note_id (oms_note_tp *note)
oms_object_id_tp	oms_note_obj_id (oms_note_tp *note)
oms_app_type_tp	oms_note_type (oms_note_tp *note)
char *	oms_note_name (oms_note_tp *note)
char *	oms_note_name_copy (oms_note_tp *note)
oms_object_id_tp	oms_note_file_id (oms_note_tp *note)
char *	oms_note_desc (oms_note_tp *note)
char *	oms_note_desc_copy (oms_note_tp *note)
void	oms_note_id_asgn (oms_note_tp *note, oms_object_id_tp id)
void	oms_note_obj_id_asgn (oms_note_tp *note, oms_object_id_tp obj_id)
void	oms_note_type_asgn (oms_note_tp *note, oms_app_type_tp *type)

Table 79: oms_note_tp Attribute Access and Assign Functions (Continued)

Type	Function
void	oms_note_name_asgn (oms_note_tp *note, const char *name)
void	oms_note_file_id_asgn (oms_note_tp *note, oms_object_id_tp file_id)
void	oms_note_desc_asgn (oms_note_tp *note, const char *desc)

Table 80: oms_note_tp Repository Management Functions

Type	Function
oms_status_tp	oms_note_update(oms_note_tp *note)
	oms_note_delete(oms_note_tp *note)

Table 81: oms_note_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_note_coll_create(const char *query)
oms_note_tp *	oms_note_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_note_coll_add(oms_coll_tp *coll, oms_note_tp *note)
	oms_note_coll_add_unique(oms_coll_tp *coll, oms_note_tp *note)
	oms_note_coll_delete(oms_coll_tp *coll, oms_note_tp *note)
	oms_note_coll_delete_equivalent(oms_coll_tp *coll, oms_note_tp *note)

Table 82: oms_note_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_note_bigcoll_create(const char *query)
oms_node_tp *	oms_note_bigcoll_next(oms_coll_tp *bigcoll)

Table 83: oms_note_tp Navigation Functions

Type	Function
oms_file_tp *	oms_note_annot_file(oms_note_tp *note)
oms_coll_tp *	oms_note_items(oms_note_tp *note)
oms_file_tp *	oms_note_file(oms_note_tp *note)
oms_node_tp *	oms_note_node(oms_note_tp *note)
oms_link_tp *	oms_note_link(oms_note_tp *note)
oms_cntx_tp *	oms_note_cntx(oms_note_tp *note)

oms_object_id_tp

The oms_object_id_tp function returns a new object id whose value is unique in the current repository. If an error occurs or no value can be allocated, the value 0L is returned.

Table 84: oms_object_id_tp Function

Type	Function
oms_object_id_tp	oms_object_id_new()

oms_pdm_type_tp

The functions in this section return information about which PDM type is being used.

Table 85: oms_pdm_type_tp Function

Type	Function
char *	oms_pdm_type_enum_to_text(oms_pdm_type_tp pdm_type)
oms_pdm_type_tp	oms_pdm_type_text_to_enum (const char *type)

oms_repos_tp

The oms_repos_open function opens a connection with the StP repository. The repository_type parameter specifies whether the repository is Sybase or Microsoft Jet; if this parameter is not specified, the function uses Sybase as the default. If hostname is NULL, then a repository on the current or default host is opened. The opened repository becomes the current repository. If there is already a current repository or the indicated repository is found and opened successfully, OMS_SUCCEED is returned. If the StP repository could not be opened, OMS_FAIL is returned.

The oms_repos_close function closes the current repository. If there is no current repository or if the StP repository could not be closed, OMS_FAIL is returned.

The oms_repos_current_server function returns the name of the current server. If no current repository exists, NULL is returned.

The oms_repos_current_repos_name function returns the name of the current repository. If no current repository exists, NULL is returned.

Table 86: oms_repos_tp Functions

Type	Function
oms_status_tp	oms_repos_open(const char *repository_type, const char *server_name, const char *database_name)
oms_status_tp	oms_repos_close()
char *	oms_repos_current_server_name()
char *	oms_repos_current_rep_name()
oms_status_tp	oms_sys_repos_open(const char *projdir, const char *system)

oms_status_tp

The oms_status_tp data type returns status information about whether a function has succeeded or failed. Possible values are OMS_SUCCEED and OMS_FAIL.

oms_time_tp

The functions in this section manipulate time strings.

Table 87: oms_time_tp Functions

Type	Function
char *	oms_time_input_format_asgn(const char *in_format)
char *	oms_time_output_format_asgn(const char *out_format)
char *	oms_time_lctime_env_asgn(const char *lang)
char *	oms_time_to_text(oms_time_tp time)
oms_time_tp	oms_time_text_to_time(const char *time)

oms_txn_tp

The functions in this section manage StP repository transactions.

Table 88: oms_txn_tp Functions

Type	Function
int	oms_txn_begin();
int	oms_txn_commit();
int	oms_txn_abort();

oms_viewpoint_tp

The functions in this section manipulate viewpoint objects.

Table 89: oms_viewpoint_tp Creation and Retrieval Functions

Type	Function
oms_viewpoint_tp *	oms_viewpoint_create(oms_object_id_tp node_id, oms_object_id_tp node_ref_id, oms_app_type_tp type)
	oms_viewpoint_find(oms_object_id_tp node_id, oms_object_id_tp node_ref_id, oms_app_type_tp type)
	oms_viewpoint_find_by_id(oms_object_id_tp id)
	oms_viewpoint_find_by_query(const char *query)

Table 90: oms_viewpoint_tp Utility Functions

Type	Function
oms_viewpoint_tp *	oms_viewpoint_copy(oms_viewpoint_tp *vp)
oms_boolean_tp	oms_viewpoint_equal(oms_viewpoint_tp *vp1, oms_viewpoint_tp *vp2)
void	oms_viewpoint_free(oms_viewpoint_tp **vp)
	oms_viewpoint_print(oms_viewpoint_tp *vp, FILE *fp)
	oms_viewpoint_print_image_asgn(const char *image)

Table 91: oms_viewpoint_tp Attribute Access and Assign Functions

Type	Function
oms_object_id_tp	oms_viewpoint_id(oms_viewpoint_tp *viewpoint)
oms_object_id_tp	oms_viewpoint_node_id (oms_viewpoint_tp *viewpoint)
oms_app_type_tp	oms_viewpoint_type (oms_viewpoint_tp *viewpoint)
oms_object_id_tp	oms_viewpoint_node_ref_id (oms_viewpoint_tp *viewpoint)
void	oms_viewpoint_id_asgn (oms_viewpoint_tp *viewpoint, oms_object_id_tp id)
void	oms_viewpoint_node_id_asgn (oms_viewpoint_tp *viewpoint, oms_object_id_tp node_id)
void	oms_viewpoint_type_asgn (oms_viewpoint_tp *viewpoint, oms_app_type_tp *type)
void	oms_viewpoint_node_ref_id_asgn(oms_viewpoint_tp *viewpoint, oms_object_id_tp node_ref_id)

Table 92: oms_viewpoint_tp Repository Management Functions

Type	Function
oms_status_tp	oms_viewpoint_update(oms_viewpoint_tp *vp)
	oms_viewpoint_delete(oms_viewpoint_tp *vp)
	oms_viewpoint_check(oms_viewpoint_tp *obj)

Table 93: oms_viewpoint_tp Typed Collection Functions

Type	Function
oms_coll_tp *	oms_viewpoint_coll_create(const char *query)
oms_viewpoint_tp *	oms_viewpoint_coll_get(oms_coll_tp *coll, int i)
oms_status_tp	oms_viewpoint_coll_add(oms_coll_tp *coll, oms_viewpoint_tp *vp)
	oms_viewpoint_coll_add_unique(oms_coll_tp *coll, oms_viewpoint_tp *vp)
	oms_viewpoint_coll_delete(oms_coll_tp *coll, oms_viewpoint_tp *vp)
	oms_viewpoint_coll_delete_equivalent(oms_coll_tp *coll, oms_viewpoint_tp *vp)

Table 94: oms_viewpoint_tp Typed Big Collection Functions

Type	Function
oms_bigcoll_tp *	oms_viewpoint_bigcoll_create(const char *query)
oms_viewpoint_tp *	oms_viewpoint_bigcoll_next(oms_coll_tp *bigcoll)

Table 95: oms_viewpoint_tp Navigation Functions

Type	Function
oms_node_tp *	oms_viewpoint_node(oms_viewpoint_tp *vp)
oms_node_ref_tp *	oms_viewpoint_node_ref(oms_viewpoint_tp *vp)

Index

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

V

X

Y