Tatanka™
DSM-CC Server:
Programmer's Reference Guide

5 January 1998

Bionic Buffalo Corporation

2533 North Carson Street, Suite 1884
Carson City, Nevada 89706-0147 USA

http://www.tatanka.com

Document Revision History

Original release, 7 August 1997, as server implementation guide.

Revised, 2 September 1997.

Revised, 14 October 1997.

Revised, 11 November 1997. Name service modifications.

Separated from server implementation guide as a separate document, 17 December 1997.

Revised, 5 January 1998. Minor changes, cleanup.

Printed 15:57:28, Sunday, 11 January, 1998.

Additional Document Information

Project name: *portugal*.

Document file name: *ptref04.doc*.

Written with Microsoft Word 97. Illustrations created with Visio Technical 4.5. Set in Monotype Bulmer, Adobe Ocean Sans, and Adobe Tekton. Published using Adobe Distiller 3.0.

Contents

Introduction

Server API Extensions

Server applications have access to the same API available to clients, as well as to certain additional functions. The client API is described in the *DSM-CC Client Programmer's Reference Guide*. The additional functions are described in detail in the chapter, Application Programming Interface (API), which is part of this document.

(Although the DSM-CC API is common to clients and to servers, the latter typically have more permissions.)

Both server and client applications also have access to the standard CORBA API defined in the CORBA 2 specification, and described in the document, *CORBA Application Programming Interface (API)*.

The DSM-CC specification (ISO/IEC 13818-6) does not describe any mechanisms for loading content onto servers. To remedy this, DAVIC defined additional functions which are included in the API available on servers. In addition, there are certain functions which allow implementor-provided methods (primarily the video pump and database engine interfaces) to interoperate with the objects and procedures which are provided as part of the server DSM-CC software.

Implementor Procedures

Two programs are to be provided by the implementor: ***ptpump*** and ***ptsqle***. The skeleton framework for the programs is provided with DSM-CC, and the implementor is responsible for providing the subroutines to complete them.

The provided framework includes the essential process (either ***pt_pump_process()*** or ***pt_sqle_process()***), which is responsible for accepting incoming messages and converting their contents into native format. The essential process calls the implementor-provided routines, passing native-format arguments.

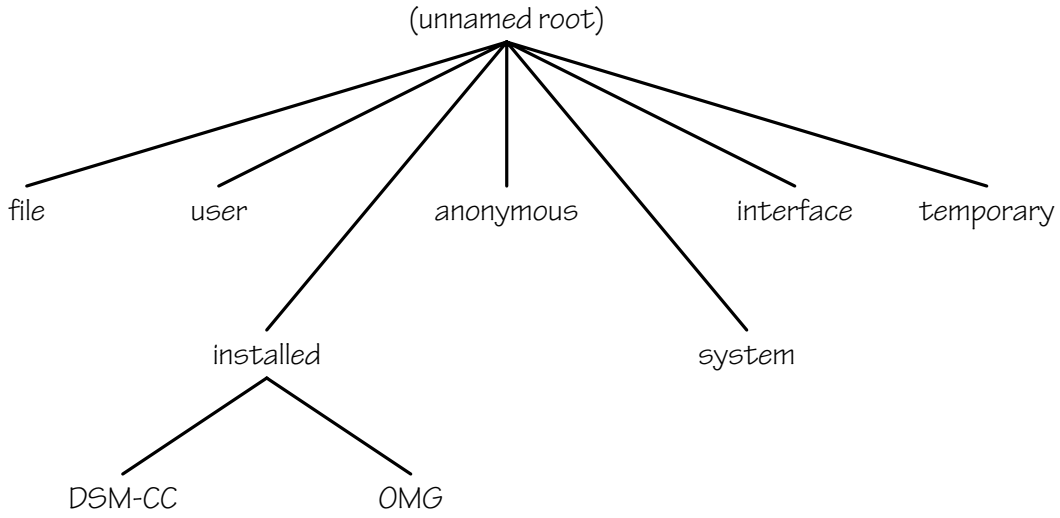
Directories and Object Names

The Object Namespaces

The name service allows for arbitrary population of namespace trees with objects. An object may be given multiple locations in the namespace. The directory, service gateway, and domain interfaces all inherit the operations of the name service, along with their topologies.

Locally, the server sees a root name space with the following partial structure:

There is no "master" directory of objects. Instead, each CORBA implementation may be given zero or more namespaces, which may or may not be related to one another. The Bionic Buffalo implementation of the Naming Service defines an initial naming context tree which includes the file system directory tree as a subtree. However, the implementor may create additional trees independent of this initial tree.



In DSM-CC, a client uses the local ***DSM::Session*** object to establish a session with a ***DSM::ServiceGateway***. The User-Network session establishment protocol returns a series of resolved object references which are returned to the application as an output parameter of the ***attach*** operation. One of these references includes a ***DSM::First*** object, which has a ***root*** operation to return the root namespace for the service gateway.

Each service gateway can configure the client's root namespace in a proprietary fashion. The directory operations are used to traverse the namespace and find the objects to be used by the client. The service gateway constructs directories as needed, then passes the desired root directory to the client.

Locally, the server begins with certain objects, which include the server's own root namespace. In Bionic Buffalo's implementation, the server's root object directory (namespace) contains at least the following items, each of which itself is a directory:

- ***anonymous*** - primarily for CORBA system use, this is a holding area for many unnamed objects of certain kinds
- ***file*** - contains the native file system viewed as objects; directories appear as naming contexts
- ***installed*** - built-in objects used by various applications
- ***interface*** - the interface repository viewed as a directory
- ***system*** - used by the CORBA implementation.
- ***temporary*** - scratch area
- ***user*** - a place for user name spaces

The objects in ***installed*** cannot be modified: it is a write-only directory. The contents of ***installed*** are defined when the system is configured.

The ***file*** objects are organized into a directory tree which is based on the server's file system. Thus, the ***file*** directory will contain the same subdirectories as the root file system of the server. Note that a persistent name does not imply that the named object itself is persistent.

The ***anonoymous*** directory contains objects which have no names. It is the name service's analog of the Unix *lost+found* directory. Not all unnamed objects will appear in the ***anonymous*** directory, however.

The implementor can modify the session attach method to assign arbitrary root name spaces to each client. This may or may not be a subtree of the local root name space tree, although in most cases the use of local subtrees is is the easiest way to manage content.

The *file* Directory and the Server File System

The server's file system is used to store persistent object information, including persistent name spaces.

File systems may contain a variety of entities, including disk files, device files, directories, and others. The naming service of the server maps all such files to objects. However, entities such as device files have opaque *kind* attributes so they are not directly accessible to clients.

In addition to host files, a name service directory also may contain the names of entities which are not files. The complete name space of a directory is a combination of file names and non-file names.

A file must have a name based on the host directory name, but may have additional names assigned by application software. Thus a file may have aliases (equivalent to Unix links or Windows shortcuts) which do not appear in the host file system directory structure.

The name service enforces the uniqueness of name components (*name* with *kind*), but cannot enforce the uniqueness of simple names only. This is because the host system might create a file with the same name as a that of a non-file object in a given directory. Occurrences of this situation can be detected because the same-named objects will have different *kind* attributes. It is up to the implementor to deal with the potential overloading of names by some combination of policy, application software, configuration choices, or other mechanisms.

The name service directories themselves, as well as information about name-object associations, are stored separately from the associated host file system directories.

When the Naming Service encounters a directory entry, it must assign a ***kind*** to the binding which associates the name with an object. The routine which determines the ***kind*** of a file can be modified easily by the implementor. An object identifier is also assigned to the directory entry. If the host system has the appropriate API, the Naming Service will determine if multiple directory entries are links to the same file, and will create only a single object identifier for the file.

The object identifier assigned to a file will persist until the Naming Service "notices" that the directory entry or entries no longer exist. Normally, notice will be taken only if a client attempts a ***list***, ***resolve***, or other Naming Service operation on the directory or name. It is possible to remove a file and create a new one of the same name without performing such a Naming Service operation, in which case the Naming Service will be fooled into thinking the object identifier assigned to the old file is valid for the new file. (Notice that this limitation is common to other software as well: most programs cannot tell if a pathname belongs to the same file to which it previously belonged.) If this behaviour in unsatisfactory, the implementor can create a notification mechanism or perhaps run a garbage collector for obsolete object identifiers.

The routine ***pt_file_remap()*** can be used to change the object identifier associated with a file name, whether or not the file itself is the same entity. This procedure can also be used to distinguish among versions of data which might be associated physically with the same file, but which are logically different.

DSM-CC Interface Objects

Objects created using DSM-CC primitives are implemented as persistent objects. When an application creates a ***DSM::Stream***, ***DSM::File***, or other such object, that object will not disappear when the system is rebooted or shut down. New objects should be named when created, or they might become orphaned if all their object references are inadvertently destroyed.

The accessibility of ***Stream*** and other objects to a client depends on their presence in a suitable object directory.

If a system implements ***Stream*** objects as files, then they will appear as ***File*** objects in the "default" name service directory associated with the server's file system. A client cannot use the ***Stream*** operations (***play***, ***pause***, etc) to "play" a ***File***. Therefore, such a file must also be placed into a directory as a ***Stream*** object. In such cases, the client will be allowed to see the directory containing the ***Stream*** object but not the directory containing the underlying ***File*** object.

Assignment of the ***Stream*** name will generally take place when the content is installed. (For example, this might occur during a ***DAV::Content::load*** operation.) However, the implementor may modify the routine which assigns ***kind***s to files. The routine can use any convenient mechanism to examine such information as the form of the path name or the first few bytes of the file's contents.

Associating Information with Objects

The methods are passed objects as parameters. For example, the ***DSM::Stream::play*** method is passed the object reference for the stream object to be played. In order to perform their operations, the methods need a mechanism to associate an object with a specific file or other entity.

If the CORBA Properties service is installed in the server, the applications which implement the DSM-CC service object operations can associate properties with the objects to record which file, directory, address, or other entity is associated with each object.

When the Properties service is not available, then an application can use two routines provided with this implementation of DSM-CC: ***pt_object_info_get()*** and ***pt_object_info_set()***. These routines allow a text string to be associated with each object. The text string can contain any information required by the implementation, such as a file name, identifier, or database key.

Service Gateways

Object Access

All initial access of a client to objects is through a Service Gateway. A Service Gateway inherits the Directory operations., and can be viewed as a kind of publicly-accessible directory.

Each Service Gateway is associated with a ***CosNaming::NamingContext*** object, which may be one of the file system directories automatically mapped by the naming service. A given directory object may be associated with more than one Service Gateway at a time.

When a client performs an ***attach*** operation, and during subsequent other interactions with a Service Gateway, the client receives various object references. The object references include profiles (as defined by the specifications) which tell the client how to communicate with the object. A client may access an object using IIOP, Download, or Carousel (BIOP) protocols. The available protocols for the objects within a Service Gateway are defined for all of the objects within the Service Gateway.

(Note, however, that one Service Gateway may be contained within other Service Gateways. In such cases, each Service Gateway may use different protocols and profiles.)

As an example, a given file system directory may be associated with two different Service Gateways. A client may use one of the Service Gateways to retrieve the files using an object carousel, or it may use the other Service Gateway to retrieve the files using IIOP.

Defining Service Gateways

The procedure ***pt_gateway_create()*** is used to create a Service Gateway object. Such an object is destroyed by ***pt_gateway_destroy()***.

The protocols used for a given Service Gateway object, and for the objects contained therein, are specified using ***pt_gateway_protocol()***. That procedure states whether objects are available for download, on carousels, or by using IIOP.

Specific users and groups are allowed or denied access to a Service Gateway by invoking the procedure ***pt_gateway_access()***. (Security and authentication may also be implemented on an object-by-object basis for other kinds of objects such as Streams and Files.)

User and Session Identification

User Management

Users are objects created by ***DAV::Domain::newMember***. The system administrator may assign users to groups as convenient.

An implementor may choose to use the object information string (accessed by ***pt_object_info_get()*** and ***pt_object_info_set()***) to contain such information as account number and network address, or the information string may be used indirectly to contain a database key or the path to a file.

Method subroutines may use the procedure ***pt_principal_get()*** to determine the user making the current request.

Session Management

When a client connects to a service gateway, a session is created. The client invokes the ***attach*** operation on a local ***DSM::Session*** object when establishing a connection, and invokes the ***detach*** operation when the connection is to be terminated. Subject to implementation limitations, a client may simultaneously have more than one connection established.

Each session is associated with a session identifier, an opaque sequence of 10 bytes. The session identifier is constant throughout the duration of the session. The session identifier known to the client is not necessarily identical to the session identifier known to the server for the same session.

When there are multiple sessions employing a single service object, then the session identifier is the only way to distinguish among them. For example, several clients playing a single stream object are differentiated at the server by different session identifiers. The user or principal identifier is not sufficient for this purpose, since a single user may establish several different, simultaneous sessions with a single object.

Within a service object method, the routine ***pt_session_id_get()*** may be used to retrieve the session identifier associated with the current request.

The Video Pump Program

Stream Play

The interaction of a client with a stream object is based on the following steps:

1. The client, having selected the desired stream from a directory, resolves the stream into an object reference.

2. Methods (***play***, ***pause***, etc) are invoked by sending messages to the object.

3. When the object is no longer needed, the ***close*** operation is performed.

All stream operations are implemented by the program ***ptpump***.

Stream Objects

In some implementations, each stream object is associated with a file, whereas in other implementations, the stream objects may be associated with other entities (such as directories, network addresses, or database keys). In order for an implementation (such as a video pump) to find the resources associated with an object, the routines ***pt_object_info_get()*** and ***pt_object_info_set()*** can be used.

In the porting kit, an implementor is given a subroutine which can be used to select the specific copy of ***ptpump*** which will receive video pump requests. In the binary versions, no such subroutine is provided, but the ptpump program itself can be a switch which acts as a front-end to redirect requests to the actual servers.

It may be appropriate to replicate heavily-used titles on multiple stream pumps or physical servers. In this case, there are two potential strategies:

- the multiple copies can be represented by multiple objects, with the selection made at the time of object resolution

- a single object can represent multiple copies of the content, with selection done at play time at the network level

Usually, the second strategy is more flexible and easier to implement, since the assignment of client to server can be done more dynamically without intervention of the object broker.

The *ptpump* State Machine

The DSM-CC specification defines each stream object as a state machine. The state transitions are computed outside of ***ptpump***, which receives simple commands. A simpler state machine models ***ptpump*** operation.

The ***ptpump*** software has three states:

- *paused* - no activity
- *playing* - transporting content
- *searching* - repositioning stream to a specified location

In the *paused* state, the pump is inactive. However, it has a current position within the stream.

In the *playing* state, the pump is transporting content to the client. The *playing* state has three parameters: rate of play, current position, and stop point.

The *searching* state is characterized by a stop point, which is the goal of the search.

The DSM-CC stream state machine passes commands to ***ptpump***, which puts them into a FIFO queue for execution. Commands are numbered sequentially, and there may be a total of up to 16 in the queue. When ***ptpump*** receives a command numbered *n*, it inserts it into the queue at the position behind command *(n - 1)*, replacing any existing command *n* and deleting any commands with numbers greater than *n*. If the current command is also numbered *n*, then the current command is terminated immediately and its replacement (the new command) is executed.

When ***ptpump*** receives a command, it returns a response (message) immediately. Depending on the command, it may also send an additional message when the command is completed.

In case of an error which occurs while executing a command, the command is terminated and an error message is sent. All subsequent commands are flushed from the queue and ***ptpump*** enters the paused state. No more commands are accepted until the error is acknowledged, preventing commands in transit from leaving the pump in an unexpected state.

Integrating the Video Pump

The implementor must provide the video pump program ***ptpump***. However, a skeleton program is included with the porting kit or with the binary forms of the video server, so an implementor need only provide certain subroutines to be called by the skeleton program.

With the porting kit, multiple copies of ***ptpump*** may be used to increase throughput. With the binary forms of the server, there is only a single copy of ***ptpump***. However, a single copy of ***ptpump*** can be used as a "switch" to route requests to multiple copies of other programs which implement the actual operations.

The subroutines to be implemented in ***ptpump*** are those named with the prefix ***pt_pump_***, and are described in more detail in the API reference portion of this document. This list summarizes their use:

- ***pt_pump_command()*** accepts a command to be enqueued by the pump
- ***pt_pump_status()*** requests an immediate status response from the pump
- ***pt_pump_reset()*** returns the pump to a known state
- ***pt_pump_info_get()*** returns information about a stream
- ***pt_pump_info_set()*** sets information for a stream
- ***pt_pump_load()*** implements the ***DAV::Content::load*** operation
- ***pt_pump_open()*** prepares a stream object for use in a specific session
- ***pt_pump_close()*** ends the use of a stream object in a specific session

These routines are provided by the implementor and are called by the ***ptpump*** skeleton code.

Object Carousels

Overview

Object carousels are used repeatedly to broadcast objects to clients. The carousel rotates through the objects, hence the name "carousel". To the client, a carousel is viewed as a Service Gateway, which inherits the directory operations.

Each carousel corresponds to a directory object on the server. The same directory object may be associated with more than one carousel.

When an object can be accessed several different ways (for example, by IIOP or by carousel), then multiple profiles appear to the client in the object's reference. The client must choose the appropriate profile to access the object.

Upon creation, each carousel is assigned an identifier which must be unique across the network. This identifier may be used by clients to select an initial carousel. The management of carousel identifiers is beyond the scope of this document.

Creation and Destruction

An object carousel is created using the procedure ***pt_carousel_create()***. A newly-created carousel is inactive until ***pt_carousel_activate()*** is invoked, and operates until ***pt_carousel_deactivate()*** is called. The procedure ***pt_carousel_destroy()*** is used to destroy a carousel which is no longer needed.

Temporary cessation and resumption of activity can be effected using ***pt_carousel_pause()*** and ***pt_carousel_resume()***. These might be used during updates of carousel parameters or contents, to prevent inconsistent carousel contents.

Carousels are persistent. They continue to exist after the server is restarted. When a server restarts, the status and parameters that were in effect when the server last ended operation are remembered by the carousel.

Operating Parameters

The operating rate of each carousel is defined by one of two strategies. An *interval-governed* carousel rotates through all objects over a predefined interval. A *bandwidth-governed* carousel transmits objects at a predefined data rate.

When objects are added to an interval-governed carousel, the bandwidth must increase to accomodate the new objects. When objects are added to a bandwidth-governed carousel, the cycle interval increases as a result.

The routine ***pt_carousel_parm_set()*** can be used to define the strategy and operating parameters for a carousel.

Frequency

By default, each object in a carousel is transmitted once in a cycle. However, the routine ***pt_carousel_frequency_set()*** can be used to increase the frequency of transmission of an object so it is sent more frequently than the normal cycle would indicate.

Coherence

In addition to ***pt_carousel_pause()*** and ***pt_carousel_resume()***, three other routines are available primarily to ensure coherence of the objects in a carousel.

When several objects are being updated simultaneously on the server, and inconsistent versions of these objects must not be sent together, ***pt_carousel_disable()*** can be used to halt temporarily the transmission of selected objects until updates are complete. Transmission can be restarted using ***pt_carousel_enable()***.

Each server file is mapped to an object, as described previously. Remapping a file to a different object may be used to distinguish one version of a file from another. Another mechanism is to use ***pt_carousel_touch()***, which merely changes the transaction identifier in the carousel.

Application Programming Interface (API)

This section describes several kinds of procedures available or used in the server implementation of DSM-CC:

1. procedures which are provided by the DSM-CC server software
2. procedures provided by the implementor (***pt_pump_****)

Refer to the preceeding part of this document for an introduction.

DAV::Content::copy - copy content

Synopsis - C

```
#include        <portugal.h>

void    DAV_Content_copy ( DAV_Content          dir1,
                           CosNaming_Name      * destination1,
                           CosNaming_Name      * source1,
                           CORBA_Environment   * env1 ) ;
```

Arguments

dir1	*(in)* content directory
destination1	*(in)* name of destination object
source1	*(in)* name of source object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

CosNaming::AlreadyBound - the name was already bound to a different object
CosNaming::CannotProceed - the resolving context did not have permission to do a resolution
CosNaming::InvalidName - the given name was not properly formed
CosNaming::NotFound - unable to resolve the given name
DSM::NO_AUTH - the end user has not provided the correct authentication in the request
(also standard exceptions)

Description

Copy a content object from one Directory to another. A copy of the source object will be created, and then bound at the destination location.

Notes

1. Requires ***WRITER*** privilege.

Reference

DAVIC 1.2, 7.3.14.

DAV::Content::load - load content

Synopsis - C

```
#include        <portugal.h>

void     DAV_Content_load ( DAV_Content                dir1,
                            CosNaming_Name            * package1,
                            CORBA_Environment         * env1 ) ;
```

Arguments

dir1	*(in)* content directory
package1	*(in)* name of content package
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

CosNaming::CannotProceed - the resolving context did not have permission to do a resolution
CosNaming::InvalidName - the given name was not properly formed
CosNaming::NotFound - unable to resolve the given name
DSM::NO_AUTH - the end user has not provided the correct authentication in the request
(also standard exceptions)

Description

Unpack a content package.

Notes

1. Requires ***WRITER*** privilege.

2. Content package encoding is as specified in DAVIC 1.2, Part 9.

Reference

DAVIC 1.2, 7.3.14.

DAV::Content::modify - modify content

Synopsis - C

```
#include        <portugal.h>

void     DAV_Content_modify ( DAV_Content                dir1,
                              CosNaming_Name            * destination1,
                              CosNaming_Name            * source1,
                              CORBA_Environment         * env1 ) ;
```

Arguments

dir1	*(in)* content directory
destination1	*(in)* name of destination object
source1	*(in)* name of source object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

CosNaming::CannotProceed - the resolving context did not have permission to do a resolution
CosNaming::InvalidName - the given name was not properly formed
CosNaming::NotFound - unable to resolve the given name
DSM::NO_AUTH - the end user has not provided the correct authentication in the request
(also standard exceptions)

Description

Replace existing content with a new source. A copy of the source object will be created, and then rebound at the destination location.

Notes

1. Requires ***WRITER*** privilege.

Reference

DAVIC 1.2, 7.3.14.

DAV::Content::move - move content

Synopsis - C

```
#include        <portugal.h>

void    DAV_Content_move ( DAV_Content                dir1,
                           CosNaming_Name            * destination1,
                           CosNaming_Name            * source1,
                           CORBA_Environment         * env1 ) ;
```

Arguments

dir1	*(in)* content directory
destination1	*(in)* name of destination object
source1	*(in)* name of source object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

CosNaming::AlreadyBound - the name was already bound to a different object
CosNaming::CannotProceed - the resolving context did not have permission to do a resolution
CosNaming::InvalidName - the given name was not properly formed
CosNaming::NotFound - unable to resolve the given name
DSM::NO_AUTH - the end user has not provided the correct authentication in the request
(also standard exceptions)

Description

Transfer content from one location to another. The source is unbound from the source parent directory, and bound to the destination location.

Notes

1. Requires ***WRITER*** privilege.

Reference

DAVIC 1.2, 7.3.14.

DAV::Domain::findGroup - return group with given name

Synopsis - C

```
#include        <portugal.h>

void    DAV_Domain_findGroup ( DAV_Domain          domain1,
                               char                * name_identifier1,
                               DAV_Group           * group1,
                               CORBA_Environment   * env1 ) ;
```

Arguments

domain1	*(in)* domain to be searched for group
name_identifer1	*(in)* name of group
group1	*(out)* group with given name
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::NAME_NO_EXIST - the name is not present in the domain
(also standard exceptions)

Description

Find a group with the given name identifier.

Notes

1. Requires ***BROKER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::findMember - return member with given name

Synopsis - C

```
#include        <portugal.h>

void    DAV_Domain_findMember ( DAV_Domain          domain1,
                                char                * name_identifier1,
                                DAV_Member          * member1,
                                CORBA_Environment   * env1 ) ;
```

Arguments

domain1	*(in)* domain to be searched for member
name_identifer1	*(in)* name of member
member1	*(out)* member with given name
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::NAME_NO_EXIST - the name is not present in the domain
(also standard exceptions)

Description

Find a member with the given name identifier.

Notes

1. Requires ***BROKER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::listGroups - list the names of groups

Synopsis - C

```
#include        <portugal.h>

void    DAV_Domain_listGroups ( DAV_Domain                domain1,
                                DSM_Domain_GroupList      * grouplist1,
                                CORBA_Environment         * env1 ) ;
```

Arguments

domain1	*(in)* domain of groups to be listed
grouplist1	*(out)* list of group names
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

List the name identifiers of groups in the given domain.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::listMembers - list the names of members

Synopsis - C

```
#include        <portugal.h>

void    DAV_Domain_listMembers ( DAV_Domain              domain1,
                                 unsigned long           count1,
                                 DAV_Roster              * roster1,
                                 DAV_RosterIterator      * roster_iterator1,
                                 CORBA_Environment       * env1 ) ;
```

Arguments

domain1	*(in)* domain of members to be listed
count1	*(in)* maximum number of member names to be returned
roster1	*(out)* initial list of member name identifiers
roster_iterator1	*(out)* iterator object to return additional member identifiers
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

List the name identifiers of members in the given domain.

If more than ***count1*** members are found, then the first ***count1*** members are returned in ***roster1*** and subsequent members can be returned using ***roster_iterator1***.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::newGroup - create new group object

Synopsis - C

```
#include        <portugal.h>

typedef DSM_Opaque              DAV_Domain_GroupToken ;

void    DAV_Domain_newGroup ( DAV_Domain              domain1,
                              char                    * name_identifier1,
                              DSM::AccessRole         access_rights1,
                              DAV_Domain_GroupToken   * token1,
                              DAV_Group               * group1,
                              CORBA_Environment       * env1 ) ;
```

Arguments

domain1	*(in)* domain to contain new group
name_identifer1	*(in)* name of new group
access_rights1	*(in)* access control rights of new group
token1	*(out)* group token
group1	*(out)* new group object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::INV_NAME - characters or length of the name is not valid
DAV::NAME_TAKEN - the proposed name already exists in this domain
(also standard exceptions)

Description

Create a new group with unique name and access role.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::newMember - create new member object

Synopsis - C

```
#include        <portugal.h>

void    DAV_Domain_newMember ( DAV_Domain          domain1,
                               char                * name_identifier1,
                               DSM_Principal       principal1,
                               DAV_Member          * member1,
                               CORBA_Environment   * env1 ) ;
```

Arguments

domain1	*(in)* domain to contain new member
name_identifer1	*(in)* name of new member
principal1	*(in)* principal of new member
member1	*(out)* new member object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::INV_NAME - characters or length of the name is not valid
DAV::NAME_TAKEN - the proposed name already exists in this domain
(also standard exceptions)

Description

Create a new member with the given name and principal.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::resolveGroup - return addressable object for group

Synopsis - C

```
#include      <portugal.h>

CORBA_Object      DAV_Domain_resolveGroup
                      ( DAV_Domain              domain1,
                      char                      * name_identifier1,
                      CORBA_Environment         * env1 ) ;
```

Arguments

domain1	*(in)* domain containing group
name_identifier1	*(in)* name of group
env1	*(in/out)* CORBA environment

Returns

an addressable multipoint object which can be used to send messages to the group

Exceptions

DAV::NAME_NO_EXIST - the specified name does not exist in the domain
(standard exceptions)

Description

Return an addressable multipoint object which can be used to send messages to a group of peers.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Domain::resolveMember - return addressable peer object

Synopsis - C

```
#include      <portugal.h>

CORBA_Object      DAV_Domain_resolveMember
                      ( DAV_Domain              domain1,
                      char                      * name_identifier1,
                      CORBA_Environment         * env1 ) ;
```

Arguments

domain1	*(in)* domain containing member
name_identifier1	*(in)* name of member
env1	*(in/out)* CORBA environment

Returns

an addressable peer object

Exceptions

DAV::NAME_NO_EXIST - the specified name does not exist in the domain
DAV::NOT_A_PEER - the member cannot receive messages (and is likely a client only)
(standard exceptions)

Description

Return an addressable peer object.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::Group::addMember - add a member to a group

Synopsis - C

```
#include    <portugal.h>

void        DAV_Group_addMember
                        ( DAV_Group             group1,
                        DAV_Member              member1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

group1	*(in)* group to which member is to be added
member1	*(in)* member to be added to group
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Add member to a group.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.2.

DAV::Group::addMemberList - add multiple members to a group

Synopsis - C

```
#include    <portugal.h>

void        DAV_Group_addMemberList
                        ( DAV_Group                 group1,
                        DAV_Group_MemberList        * member_list1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

group1	*(in)* group to which members are to be added
member_list1	*(in)* list of members to be added to group
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Add mulitple members to a group.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.2.

DAV::Group::removeMember - remove a member from a group

Synopsis - C

```
#include    <portugal.h>

void        DAV_Group_removeMember
                        ( DAV_Group             group1,
                        DAV_Member              member1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

group1	*(in)* group from which member is to be removed
member1	*(in)* member to be added to group
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Remove a member from a group.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.2.

DAV::Group::Role - access role of group

Synopsis - C

```
#include        <portugal.h>

DSM_AccessRole      DAV_Group__get_Role
                        ( DAV_Group                 group1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

group1	*(in)* group for which access role is to be determined
env1	*(in/out)* CORBA environment

Returns

access role of group

Exceptions

(standard exceptions)

Description

Returns the access role (privileges) of a group.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.2.

DAV::Member::AccessRoles - determine access roles of a member

Synopsis - C

```
#include        <portugal.h>

DAV_Member_AccessRolesList    * DAV_Member__get_AccessRoles
                                    ( DAV_Member                member1,
                                    CORBA_Environment           * env1 ) ;
```

Arguments

member1	*(in)* member whose roles are to be determined
env1	*(in/out)* CORBA environment

Returns

the access roles of the member

Exceptions

(standard exceptions)

Description

Returns the access roles of a member in a domain.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.1.

DAV::Member::grant - promote a member to a higher access role

Synopsis - C

```
#include    <portugal.h>

void        DAV_Member_grant
                        ( DAV_Member              member1,
                        DSM_AccessRole            role1,
                        CORBA_Environment         * env1 ) ;
```

Arguments

member1	*(in)* member to be promoted
role1	*(in)* new role of member
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::INV_ACCESS_ROLE - invalid access role
(standard exceptions)

Description

Promote a member to a higher access role.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.1.

DAV::Member::PrincipalId - unique identifier of principal in a domain

Synopsis - C

```
#include        <portugal.h>

DSM_Principal           * DAV_Member__get_PrinicipalId
                                ( DAV_Member                member1,
                                CORBA_Environment           * env1 ) ;
```

Arguments

member1	*(in)* member whose principal is to be returned
env1	*(in/out)* CORBA environment

Returns

the prinicipal identifier of the member

Exceptions

(standard exceptions)

Description

Returns the unique identifier of a principal in a domain.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.1.

DAV::Member::Privs - access privileges

Synopsis - C

```
#include        <portugal.h>

DSM_Access_Perms_T          * DAV_Member__get_Privs
                                    ( DAV_Member                member1,
                                    CORBA_Environment           * env1 ) ;
void                        DAV_Member__set_Privs
                                    ( DAV_Member                member1,
                                    DSM_Access_Perms_T          * perms1,
                                    CORBA_Environment           * env1 ) ;
```

Arguments

member1	*(in)* member whose privileges are set or returned
perms1	*(in)* new privileges of member
env1	*(in/out)* CORBA environment

Returns

DAV_Member__get_Privs - current member permissions
DAV_Member__set_Privs - *(void)*

Exceptions

(standard exceptions)

Description

Get or set the access permissions of a member.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.1.

DAV::Member::revoke - demote a member to a lower access role

Synopsis - C

```
#include    <portugal.h>

void        DAV_Member_revoke
                        ( DAV_Member              member1,
                        DSM_AccessRole            role1,
                        CORBA_Environment         * env1 ) ;
```

Arguments

member1	*(in)* member to be demoted
role1	*(in)* new role of member
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

DAV::INV_ACCESS_ROLE - invalid access role
(standard exceptions)

Description

Demote a member to a lower access role.

Notes

1. Requires ***MANAGER*** privilege.

Reference

DAVIC 1.2, 7.4.1.

DAV::RosterIterator::destroy - destroy roster iterator

Synopsis - C

```
#include    <portugal.h>

void        DAV_RosterIterator_destroy
                        ( DAV_RosterIterator        roster_iterator1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

roster_iterator1	*(in)* roster iterator to be destroyed
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Destroys the specified roster iterator.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

DAV::RosterIterator::next_n - return next portion of member list

Synopsis - C

```
#include    <portugal.h>

void        DAV_RosterIterator_next_n
                        ( DAV_RosterIterator      roster_iterator1,
                        unsigned long             count1,
                        DAV_Roster                * roster1,
                        CORBA_Environment         * env1 ) ;
```

Arguments

roster_iterator1	*(in)* roster iterator to be destroyed
count1	*(in)* the maximum number of members to be returned
roster1	*(out)* list of member names
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Returns the next ***count1*** members of a list.

Notes

1. Requires ***READER*** privilege.

Reference

DAVIC 1.2, 7.4.3.

pt_file_remap () - change object for automatic file mapping

Synopsis - C

```
#include    <portugal.h>

void        pt_file_remap
                        ( CORBA_Object          file_object1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

file_object1	*(in)* an object associated with a native file or directory
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Destroys the given file system object, causing a new object to be defined in its place.

Notes

1. Used to distinguish between two versions of a file which otherwise appear identical. Forces the current version of the file to have a different object identifier from the previous version.

Reference

proprietary function

pt_carousel_activate () - activate an object carousel

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_activate
                    ( pt_carousel           carousel1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* a carousel object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Places a carousel into the *ACTIVE* state.

Notes

1. A carousel which enters the *ACTIVE* state begins sending the modules it contains.

2. New carousels default to the *INACTIVE* state. This allows parameters to be set before operations begin.

Reference

proprietary function

pt_carousel_create () - create an object carousel

FIXME

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        unsigned char                   AFI ;
        unsigned char                   type ;
        unsigned long                   carousel_id ;
        struct
                {
                unsigned char           type ;
                unsigned char           organization [ 3 ] ;
                }
                specifier ;
        unsigned char                   private_data [ 10 ] ;
        }
        pt_NSAP_address_t ;

void            pt_carousel_create
                                ( pt_server                     server1,
                                pt_NSAP_address_t               * address1,
                                CosNaming_NamingContext         directory1,
                                char                            * channel1,
                                pt_carousel                     carousel1,
                                CORBA_Environment               * env1 ) ;
```

Arguments

server1	*(in)* the server on which the carousel is to be created
address1	*(in)* the carousel NSAP address
directory1	*(in)* the directory containing the carousel objects
channel1	*(in)* the output channel for the new carousel
carousel1	*(out)* the new carousel object
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Creates a new object carousel.

Notes

1. All of the objects in ***directory1*** (including contained subdirectories) are included in the carousel.

2. The carousel is created in the *INACTIVE* state.

Reference

proprietary function

pt_carousel_deactivate () - deactivate an object carousel

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_deactivate
                    ( pt_carousel           carousel1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Causes a carousel to cease operation.

Notes

(none)

Reference

proprietary function

pt_carousel_destroy () - destroy an object carousel

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_destroy
                        ( pt_carousel           carousel1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Destroys an object carousel.

Notes

(none)

Reference

proprietary function

pt_carousel_disable () - disable transmission of specified objects

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_disable
                        ( pt_carousel               carousel1,
                        CORBA_sequence_Object       * objects_to_diable1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
objects_to_disable1	*(in)* the object to disable
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Disables the transmittion of specific objects in a carousel.

Notes

1. By default, all objects in a carousel's directory are transmitted. This mechanism can prevent some of those objects from being sent.

2. The operation is atomic. All specified objects are disabled together at the next cycle of the carousel.

Reference

proprietary function

pt_carousel_enable () - enable transmission of specified objects

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_enable
                    ( pt_carousel               carousel1,
                    CORBA_sequence_Object       * objects_to_enable1,
                    CORBA_Environment           * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
objects_to_enable1	*(in)* objects for which transmission is to be enabled
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Resumes tranmission of the specified objects.

Notes

1. The objects must have been disabled previously by ***pt_carousel_disable()***. This call cannot be used to add objects to a carousel which are not in the carousel's directory or its subdirectories.

2. The operation is atomic. All specified objects are enabled on the same (next) cycle.

Reference

proprietary function

pt_carousel_frequency_set () - change frequency of object transmission

Synopsis - C

```
#include        <portugal.h>

typedef union
        {
        struct
                {
                unsigned long                   seconds ;
                unsigned long                   nanoseconds ;
                }
                interval ;
        unsigned long                           bandwidth ;
        }
        pt_carousel_frequency_parm_t ;

void            pt_carousel_frequency_set
                                ( pt_carousel                   carousel1,
                                char                            governor_type1,
                                pt_carousel_frequency_parm_t    frequency1,
                                CORBA_Environment               * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
governor_type1	*(in)* the type of governor used in transmission *'i'* - interval governor *'b'* - bandwidth governor
frequency1	*(in)* the rate of carousel, either an interval or a bandwidth
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Sets the operating frequency of the carousel.

Notes

1. If the carousel is to be interval-governed, then the frequency parameter should specify the time for one complete cycle of the carousel.

2. If the carousel is to be bandwidth governed, then the frequency parameter should specify the average bandwidth to be used. The bandwidth is specified in units of 100 bytes/second.

3. Bandwidth limits do not set the output rate of the channel. The carousel controls bandwidth by controlling the number of messages sent on the channel so that the average rate should approximate the specified rate. Each individual message, however, will be sent at the natural rate of the channel.

4. The carousel will attempt to operate at the designated frequency, but may not be able to control the rate exactly. The exact behaviour may be implementation-dependent.

Reference

proprietary function

pt_carousel_parms_set () - define operating parameters for an object carousel

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        unsigned long                   block_size ;
        CORBA_sequence_octet            * private_data ;
        }
        pt_caousel_parms_t ;

void            pt_carousel_parms_set
                                ( pt_carousel                   carousel1,
                                pt_carousel_parms_t             * parms1,
                                CORBA_Environment               * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
parms1	*(in)* operating parameters
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Set the operating parameters of a carousel.

Notes

1. The private data defines the ***privateDataByte*** field of the ***DownloadInfoIndication*** message.

Reference

proprietary function

pt_carousel_pause () - pause an object carousel

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_pause
                    ( pt_carousel           carousel1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Temporarily halts operation of a carousel.

Notes

1. Operation is halted at the end of the current transmission cycle.

Reference

proprietary function

pt_carousel_resume () - start a paused object carousel

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_resume
                    ( pt_carousel           carousel1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Resumes operation of a paused object carousel.

Notes

(none)

Reference

proprietary function

pt_carousel_touch () - update transaction identifier

Synopsis - C

```
#include    <portugal.h>

void        pt_carousel_touch
                    ( pt_carousel           carousel1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

carousel1	*(in)* an object carousel
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Increments the transaction id for carousel transmission.

Notes

1. Normally, the transaction identifier is updated automatically when the carousel determines any parameters have changed, or when the constituent file times or lengths are altered. This routine can force a transaction identifier change when the normal mechanism fails.

Reference

proprietary function

pt_gateway_access () - define access to a Service Gateway

Synopsis - C

```
#include    <portugal.h>

void        pt_gateway_access
                    ( pt_gateway            gateway1,
                    pt_access_supervisor    supervisor1,
                    CORBA_any               * parm1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

gateway1	*(in)* a service gateway object
supervisor1	*(in)* an access supervisor for the gateway
parm1	*(in)* parameters for ***supervisor1***
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Defines an access supervisor for a service gateway.

Notes

1. The access supervisor will define the security parameters of the gateway, including use of passwords, cryptographic challenges, and other mechanisms.

2. The type associated with ***parm1*** depends on the requirements of ***supervisor1***.

3. Until an access supervisor is assigned to the gateway, no access by clients is permitted.

4. Assigning the *NIL* object as access supervisor allows universal access with no restrictions.

5. Objects within the gateway may impose additional restrictions upon their own access.

Reference

proprietary function

pt_gateway_create () - create Service Gateway

Synopsis - C

```
#include    <portugal.h>

void        pt_gateway_create
                        ( pt_server                 server1,
                        CosNaming_NamingContext     directory1,

                        CORBA_Environment           * env1 ) ;
```

Arguments

server1	*(in)* the server on which the gateway is to be created
directory1	*(in)* the directory mapped to the new gateway
gateway1	*(out)* the newly-created service gateway
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Creates a new service gateway on the specified server.

Notes

1. Initially, no protocols are defined for the gateway.

Reference

proprietary function

pt_gateway_destroy () - destroy Service Gateway object

Synopsis - C

```
#include    <portugal.h>

void        pt_gateway_destroy
                        ( pt_gateway                gateway1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

gateway1	*(in)* the service gateway to be destroyed
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Destroys a service gateway.

Notes

(none)

Reference

proprietary function

pt_gateway_protocol () - specify a protocol for a Service Gateway domain

Synopsis - C

```
#include    <portugal.h>

void        pt_gateway_protocol
                    ( pt_gateway            gateway1,

                      CORBA_Environment     * env1 ) ;
```

Arguments

gateway1	*(in)* the service gateway for which protocols are to be defined
protocol1	*(in)* the protocol object
parm1	*(in)* parameter for ***protocol1***
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Define a protocol for a service gateway.

Notes

1. Using successive calls to this procedure, multiple protocols may be defined for the service gateway.

2. The protocols will define the profiles specified in the gateway objects' IORs.

Reference

proprietary function

pt_object_info_get () - return an object's information string

Synopsis - C

```
#include    <portugal.h>

void        pt_object_info_get
                    ( CORBA_Object          obj1,
                    char                    ** info1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

obj1	*(in)* any object defined on the video server
info1	*(out)* information string
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Returns the information string set by ***pt_object_info_set()***.

Notes

3. The information string may be used by an implementor to associate arbitrary information (such as a file name, network address, or comment) with an object.

    3. Requires ***READER*** privilege.

Reference

proprietary function

pt_object_info_set () - set the value of an object's information string

Synopsis - C

```
#include    <portugal.h>

void        pt_object_info_set
                    ( CORBA_Object          obj1,
                    char                    * info1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

obj1	*(in)* any object defined on the video server
info1	*(in)* information string
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Set the value of the information string to be retrieved by ***pt_object_info_get()***.

Notes

1. The information string may be used by an implementor to associate arbitrary information (such as a file name, network address, or comment) with an object.

2. Requires ***OWNER*** privilege.

Reference

proprietary function

pt_principal_get () - determine the principal making the current request

Synopsis - C

```
#include    <portugal.h>

void        pt_principal_get
                    ( CORBA_Object          * principal1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

principal1	*(out)* the principal behind the current request
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Used within method procedures. Determines the principal object (user or member) making the current request.

Notes

1. This is a simplified form of ***CORBA::BOA::get_principal*** for use in video servers.

Reference

proprietary function

pt_pump_close () - end use of stream object for a session

Synopsis - C

```
#include    <portugal.h>

void        pt_pump_close
                        ( DSM_Stream            stream1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

stream1	*(in)* the stream object used in the session
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Tells the pump that the stream object is no longer required in the session.

Notes

1. Provided by the system implementor.

Reference

proprietary function

pt_pump_command () - accept stream activity command

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        unsigned long           op_code ;
        unsigned long           command_number ;
        unsigned long           trace ;
        DSM_AppNPT              time1 ;
        DSM_AppNPT              time2 ;
        DSM_Scale               scale ;
        short                   progress_percent_interval ;
        short                   progress_maximum_interval ;
        short                   progress_minimum_interval ;
        unsigned int            start_message : 1 ;
        unsigned int            completion_message: 1 ;
        unsigned int            progress_messages: 1 ;
        }
        pt_pump_command_t ;

typedef struct
        {
        unsigned long           message_type ;
        unsigned long           trace ;
        unsigned long           status ;
        DSM_AppNPT              estimated_time_to_complete ;
        short                   estimated_percent_complete ;
        unsigned int            immediate_start : 1 ;
        unsigned int            immediate_completion : 1 ;
        }
        pt_pump_response_t ;

void            pt_pump_command
                        ( DSM_Stream                stream1,
                        pt_pump_command_t           * command1,
                        pt_pump_response_t          * response1,
                        CORBA_Environment           * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be used in the session
command1	*(in)* command sent to pump
response1	*(in)* response returned from pump
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Passes a command to the pump and retrieves a response.

Notes

1. Provided by the system implementor.

2. The command number is used to place the command into the FIFO command queue. If the command number is equal to a command number already in the queue or if it is equal to the current command number, then it replaces that command and all subsequent commands are flushed from the queue.

3. The op_code defines the action to be taken by the pump:
   PT_PUMP_CMD_RESET - reset the pump to its initial state
   PT_PUMP_CMD_SEEK - seek to the position ***time1***
   PT_PUMP_CMD_PLAY - play from ***time1*** to ***time2***
   PT_PUMP_CMD_PAUSE - pause when reaching ***time1***

4. The ***trace*** value is not interpreted by the pump, but is returned to the caller with status reports and messages. It is used by the caller to match specific messages to their source.

5. The scale determines the rate at which the stream is to be played.

6. If ***start_message*** is set, then a progress message is to be sent when the command starts. However, if the queue is empty and the command begins immediately, then ***immediate_start*** is set in the reply and no start message is sent.

7. If ***completion_message*** is set, then a progress message is to be sent when the command is completed. However, if the queue is empty and the command begins and completes immediately, then ***immediate_completion*** is set in the reply and no completion message is sent.

8. The ***pt_pump_response_t*** structure is used in the return from ***pt_pump_command()***. It is also used for start, completion, progress, and error messages. The ***message_type*** field designates its use:
   PT_PUMP_MSG_RETURN - return value from ***pt_pump_command()***
   PT_PUMP_MSG_START - notification of command start
   PT_PUMP_MSG_COMPLETION - notification of command completion
   PT_PUMP_MSG_ERROR - notification of error in processing command
   PT_PUMP_MSG_PROGRESS - command execution progress

9. If ***progress_messages*** is set, then the pump should send periodic progress messages to indicated execution progress and estimated completion time and percentage. Messages should be sent every ***progress_percent_interval*** percent, subject to the minimum and maximum allowed message intervals. The intervals ***progress_minimum_interval*** and ***progress_maximum_interval*** are specified in tenths of a second.

Reference

proprietary function

pt_pump_info_get () - return information about a stream object

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        char                            * aDescription ;
        DSM_AppNPT                      duration ;
        CORBA_boolean                   audio ;
        CORBA_boolean                   video ;
        CORBA_boolean                   data ;
        }
        DSM_Stream_Info_T ;

void            pt_pump_info_get
                                ( DSM_Stream                    stream1,
                                DSM_Stream_Info_T               ** info1,
                                CORBA_Environment               * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be used in the session
info1	*(out)* stream information
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Returns information about a stream object.

Notes

1. Used to implement the ***DSM::Stream::Info*** attribute.
2. Provided by the system implementor.

Reference

proprietary function

pt_pump_info_set () - set information values for a stream object

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        char                        * aDescription ;
        DSM_AppNPT                  duration ;
        CORBA_boolean               audio ;
        CORBA_boolean               video ;
        CORBA_boolean               data ;
        }
        DSM_Stream_Info_T ;

void            pt_pump_info_set
                                ( DSM_Stream                stream1,
                                DSM_Stream_Info_T           * info1,
                                CORBA_Environment           * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be used in the session
info1	*(in)* stream information
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Sets information for a stream object.

Notes

1. Used to implement the ***DSM::Stream::Info*** attribute.
2. Provided by the system implementor.

Reference

proprietary function

pt_pump_load () - load content package

Synopsis - C

```
#include    <portugal.h>

void        pt_pump_load
                        ( DSM_Stream            stream1,
                        char                    * package1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be loaded
package1	*(in)* pathname to content package file
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Loads a content package.

Notes

1. Used to implement the ***DAV::Content::load*** operation. Content package encoding is as specified in DAVIC 1.2, Part 9.

2. Provided by the system implementor.

Reference

proprietary function

pt_pump_open () - begin use of stream object for a session

Synopsis - C

```
#include    <portugal.h>

void        pt_pump_open
                        ( DSM_Stream            stream1,
                        void                    * recipient1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be used in the session
recipient1	*(in)* destination to be used for messages sent by ***pt_pump_send_message()***
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Tells the pump to prepare a stream object for use in a session.

Notes

1. Provided by the system implementor.

Reference

proprietary function

pt_pump_reset () - return the pump to a known, inactive state

Synopsis - C

```
#include    <portugal.h>

void        pt_pump_reset
                        ( DSM_Stream            stream1,
                        CORBA_Environment       * env1 ) ;
```

Arguments

stream1	*(in)* the stream object used in the session
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Returns the pump to a known, inactive state for the current session.

Notes

1. All commands are flushed from the queue, transport of content is stopped, and the pump enters the paused state at the beginning of the stream. The error flag is cleared.

2. The same as a ***PT_PUMP_CMD_RESET*** command as sent to ***pt_pump_command()***, except it is executed immediately and not enqueued.

3. Provided by the system implementor.

Reference

proprietary function

pt_pump_send_message () - send notification message

Synopsis - C

```
#include    <portugal.h>

int         pt_pump_send_message
                        ( void              * recipient1,
                        void                * message_body1,
                        unsigned long       message_size1 ) ;
```

Arguments

recipient1	*(in)* destination for messages (from ***pt_pump_open()***)
message_body1	*(in)* body of message
message_size1	*(in)* size of message

Returns

zero if no error, else non-zero

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Sends a notification or command completion message.

Notes

1. Messages to be sent are specified by parameters passed to ***pt_pump_command()***.
2. Usually, the message is a ***pt_pump_response_t*** structure.

Reference

proprietary function

pt_pump_status () - return status for current session

Synopsis - C

```
#include        <portugal.h>

typedef struct
        {
        unsigned long                   command_number ;
        unsigned long                   command_trace ;
        unsigned long                   command_queue_size ;
        DSM_AppNPT                      position ;
        DSM_Scale                       scale ;
        DSM_AppNPT                      estimated_time_to_complete ;
        int                             error_flag ;
        }
        pt_pump_stream_status_t ;

void            pt_pump_status
                                ( DSM_Stream                    stream1,
                                pt_pump_stream_status_t         ** status1,
                                CORBA_Environment               * env1 ) ;
```

Arguments

stream1	*(in)* the stream object to be used in the session
status1	*(out)* stream status for current session
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Called by the essential process of ***ptpump***. Returns information about the status of a stream object in the current session.

Notes

1. Used to implement the ***DSM::Stream::status*** operation.
2. Provided by the system implementor.
3. If the ***error_flag*** is set, the pump is locked due to a command error, and must be reset using ***pt_pump_reset()***.

Reference

proprietary function

pt_session_id_get () - determine the session identifier for the current request

Synopsis - C

```
#include    <portugal.h>

void        pt_session_id_get
                    ( DSM_Opaque            * session_id1,
                    CORBA_Environment       * env1 ) ;
```

Arguments

session_id1	*(out)* the session identifier associated with the current request
env1	*(in/out)* CORBA environment

Returns

(void)

Exceptions

(standard exceptions)

Description

Used within method procedures. Determines the session identifier associated with the current request.

Notes

1. The session identifier is used in the user-network protocol, and is defined as a 10-byte field.

Reference

proprietary function