ISO/IEC MPEG 13818-6 DSM-CC INTERNATIONAL STANDARD
User-to-User Interfaces

July 12, 1996 (WG11, Tampere, Finland)
Aug 27, 1996 (editorials and IDL corrections)
Donald Hooper
13818-6 Editor, User-to-User

Status: Friday plenary, reviewed, voted and approved (not the final typographical edit)
+ Editorial changes, IDL corrections to enable successful compilation

Editorial Changes

◊ 5.2.1 change "may be act as" to "may act as"

◊ 5.5.1.6.1 M1140 Service Transfer changes to match with Annex L.

replace
b) Enhanced application level service Transfer with **State suspend()** and **State resume()**

with
b) Enhanced application level service Transfer through **Session detach(), State suspend()** and **State resume()**

replace
- Release the Session with the sourceServer
- Maintain minimum resources with the sourceServer
- Maintain the Service with the sourceServer

In the method for the first two options above, the Client can resume the full context at a later time. Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

with
Release the Session with the sourceServer (using **detach()** with **aSuspend** true)
Suspend a service with forced release of its connection resources (using **suspend()** with **aRelease** true)
Suspend a service without forced release of its connection resources i.e., can be reassigned by the SessionGateway to another service within a time-out period (using **suspend()** with **aRelease** false)
Maintain the Service with the sourceServer
In this method for the first three items above, the Client can resume the full context at a later time. Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

◊ 5.5.1.6.3 change ". if" to ". If"

◊ 5.6.2 Remove "It is defined later in this section." under bullet Requesting Principal. In fact it is defined earlier.

IDL Corrections (to enable successful compilation):

◊ Section 5.4.2 Entity Identification (compile error: ServiceDomain must be a type)

replace
```
struct ServiceLocation {
        sequence<octet, 20> serviceDomain;      // ServerId NSAP
        CosNaming::Name pathName;     // path name to resolve
        UserContext  initialContext;      // starting user context
  };
```

with
```
  typedef sequence<octet, 20> ServiceDomain;     // ServerId NSAP
  struct ServiceLocation {
        ServiceDomain aServiceDomain;
        CosNaming::Name pathName;     // path name to resolve
        UserContext  initialContext;      // starting user context
  };
```

◊ Section 5.5.1.6.3 Session attach

replace
```
    void  attach (
        in sequence<octet, 20> serviceDomain,       // optional ServerId
```

with

```
        void  attach (
                in ServiceDomain aServiceDomain,            // optional ServerId
```

◊ Section 5.5.2.2.5 Event notify (compile errors: ifndef shd be ifdef, comments not allowed after #macro)

replace ifndef *with* #ifdef

replace

```
#endif  // DSM_PSEUDO
```

with

```
#endif
```

◊ Section 5.5.2.3 Composite (compile error: name collision on identifier: composite)

replace

For Composite bindings at a Directory, the BindingType shall be **DSM::composite**.

with

For Composite bindings at a Directory, the BindingType shall be **DSM::cobject**.

◊ Section 5.5.2.3.2 and 5.6.6.7 Composite (compile error: BindingType not defined, name collision)

replace

```
    const BindingType composite = 2;
```

with

```
    typedef u_long BindingType;    // extends CosNaming BindingType
    const BindingType cobject = 2;
```

◊ Section 5.5.2.6.2 (compile error: some IDL compilers do not accept hex consts)

replace

```
const TCKind tk_SERVICE_XFR = 0x49535050;
    // 0x49535051 - 0x495350FF ISO/IEC reserved
```

with

```
const TCKind tk_SERVICE_XFR = 1230196815; //  0x4953504F
    // 0x49535050 - 0x495350FF ISO/IEC reserved
```

◊ Section 5.6.3.4 Lite Protocol Profile (compile error: missing semicolon)

add semicolon at end of struct LiteComponent

◊ Section 5.6.6.5 (compile errors: extraneous semicolon, name collision)

remove extraneous semicolon after ModuleInstallList

in struct InfoResponse,
replace

```
        ModuleInfoList moduleInfoList;
```

with

```
        ModuleInfoList modulesInfo;
```

5. User-to-User Interfaces

5.1 Introduction

This Chapter of the ISO/IEC 13818-6 standard specifies a generic set of multimedia interfaces, called User-to-User interfaces. Each interface defines a set of operations that can be invoked on a service object. The operations offer function calls in a choice of programming languages, and Client/Service Remote Procedure Calls(RPC) in a choice of network protocol profiles. With a DSM-CC User-to-User Library, both application portability and network interoperability can be achieved. The User-to-User interfaces represent modular, basic building blocks that can be used to enable a range of capabilities from minimal, low-cost ‘Consumer’ Clients, that navigate and request multimedia data, to high-powered ‘Producer’ Clients, that configure and load the multimedia Service Domain.

5.1.1 What’s in this Chapter

This Chapter is organized as follows:

- The User-to-User System Environment. From a User-to-User perspective, this section explains system entities, objects, interfaces, object references and Clients, in the context of the DSM-CC system environment.
- Overview of the Interface Definition Language(IDL). Although CORBA specifies the complete IDL, this section covers the basic IDL vocabulary needed for an understanding of DSM-CC User-to-User.
- Common Definitions. This section specifies common types and constants that are used by the DSM-CC User-to-User interfaces.
- Application Portability Interfaces(API). The intended audience for this section is the application software programmer . This section is organized as two partitions. The first is the minimum compliant Core interface set. The second is the extended interface set. Each interface is further partitioned to have a consumer Client subset.
- Service Inter-operability Interfaces(SII). The intended audience for this section is the network software programmer. This section defines structures and encodings needed for interoperability between Client and Service entities over a network. It is also organized as two partitions, Core and Extended.
- Application Boot Process. This section integrates the Application Portability Interfaces, User-to-Network Session protocol, Download protocol, and Service Inter-operability Interfaces into a working system model. It presents phases of application startup, and specifies the pre-conditions for each phase.

5.1.2 Intended Usage

The DSM-CC User-to-User interfaces enable a wide-range of multimedia applications to run in heterogeneous networked environments. For example, applications that use DSM-CC as a foundation include:

- Movies On Demand
- Movie Listing
- TV Program Guide
- Tele-Shopping
- Video-Conferencing
- Near Movies On Demand
- News on Demand
- Karaoke On Demand
- Games
- Tele-Medicine
- Distance Learning

and others

DSM-CC has developed relationships with other industry standards. The standards that either use or are compatible with DSM-CC User-to-User include:

- DAVIC. Digital Audio Visual Council
- DVB. Digital Video Broadcasting
- MHEG. Multimedia Hypermedia Experts Group
- ITU-T Study Group 8, T.120 and T.130 Video Conferencing
- CORBA. Common Object Request Broker
- TINA-C. Telecommunications Information Network Architecture
- UNO. Universal Networked Objects
- ONC. Open Networked Computing
- DCE, Distributed Computing Environment
- Languages C, C++, SQL

and others

The above uses and environments have inspired the development of the DSM-CC International Standard. As a result, the protocols offer a generic, unified and extensible set of functionalities. Many innovations are included which offer performance, size, scalability and security advantages. Here are some of the key characteristics of DSM-CC User-to-User:

- DSM-CC User-to-User specifies consistent, unified interfaces for commonly-used multimedia object types, with methods for creation, navigation, access and control.
- The User-to-User APIs enable applications to be designed that can be used in both interactive and broadcast carousel environments.
- The User-to-User interfaces leverage the power of inheritance, enabling developers to build on simple, generic libraries while adding application-specific functionalities.
- User-to-User defines programming interfaces that present function calls that are common to all Clients(given a language selection), regardless of hardware and operating system. To enable interoperability over the network, User-to-User specifies protocol profiles that establish common on-the-wire encodings and protocols between Clients and Services.
- Certain User-to-User operations enable efficient operation over networks that may have a long latency between Client and Service, limited Client storage and limited network request path bandwidth.
- The User-to-User interfaces enable Client application interfaces to consist of simple stubs, that may be downloaded by the DSM-CC download mechanism. The minimal Client need not be a name Service, perform authentication or authorization, or choose which instance of Service or asset to connect to. These functions will typically be performed by service brokers and asset brokers on the Server side of the network.
- A minimal consumer Client set of interfaces supports the requirements of a TV set-top device, where there is typically limited memory and no disk.
- While the interfaces are simplified to accomplish the minimal Client, a range of privileges (Access Roles) and extended interfaces support Clients with more capabilities and resources, such as peer Clients or authoring tool Clients.
- The Stream objects offer control of MPEG-2 and other continuous media streams. The normal play time concept is introduced, to insure accurate, contiguous and predictable timestamps for synchronized audio, video and stream event data. The stream operations enable consistent behavior, accurate to a frame, regardless of the network latency between Client and Server.
- Directory objects enable producer Clients to establish and load logical path hierarchies with multimedia objects and Services, and enable consumer Clients to navigate these paths, and access these objects and Services.
- Composite objects enable the opening of several child objects as a result of resolving a single name. Composite objects also simplify the versioning relationships between objects by enabling the association of compatible versions to a Composite name.
- Security applies to the entry into a Service Domain (Directory hierarchy) and to invocation of operations. Security features include:

- ⇒ Authentication of the principal end user on a per request basis.
- ⇒ Authorization of the principal end user to perform the requested operation based on Access Role.
- ⇒ Password or encryption access challenge/response control is supported on a per request basis(including entry into Service Domains)
- ⇒ All secure flag to trigger secure transmission for all messages to/from an object.

- The Remote Procedure call configuration interface allows for either synchronous or asynchronous (synchronous deferred) operation. Synchronous deferred operation supports pipelining of requests, where multiple parallel requests can be outstanding at a given time.
- Sorts and filters can be performed with a common syntax for both Directory and Database browsing. Query results can be pre-fetched using a windowing mechanism.

5.2 The User-to-User System Environment

The DSM-CC System Environment can be visualized as a distributed network of Users of varying capabilities. The Users may be connected by a DSM-CC network as defined in the User-to-Network section of ISO/IEC 13818-6, or by private network (without the User-to-Network protocol).

The User entities are listed under physical and logical categories as follows:

5.2.1 U-U System Hardware User Entities

- **Hardware Servers**. These platforms are configured with MPEG-2 storage and delivery capability. Such Servers include a heterogeneous variety of storage and processor systems. The Server has operating system software that supports the operation of Services and applications, and isolates these from the underlying hardware devices and network protocols.
- **Hardware Clients** These devices run applications and provide the underlying capability of decoding and displaying MPEG-2 streams. Hardware Clients include a heterogeneous variety of set-top boxes, PCs, etc. The hardware Client has operating system software that supports the operation of applications that run on it and isolates these from underlying hardware devices and network protocols.
- **Other Platforms**. A heterogeneous variety of hardware/OS/Network platforms may act as Clients or Servers, or both, depending upon the application, independent of MPEG-2 streaming capability.

5.2.2 U-U System Logical Entities

- **Service Object Implementation**. A Service Object Implementation is a logical entity in the system that supports the syntax and semantics of an interface, meaning it can receive requests on that interface. The interface declares operations, and may inherit other interfaces. The Service Object Implementation is also referred to as a Service.
- **Client**. An application program or process becomes a Client when it initiates a request (invokes operations) to an Object Implementation. It is also referred to as a Client Application.
- **Application**. An application is a named entity that supports a functional theme, represented by a Client Application and one or more Service Object Implementations. One of the Service Object Implementations is designated the Primary Service for the application, and assumes the application’s logical path name.
- **Object Reference**. An object reference is a local handle through which a Client Application makes requests to a remote Object Implementation.
- **Client-side DSM-CC Library Stubs**. These functional elements mediate between the Client Application and an RPC protocol, providing a simplified function call interface to the application, while encoding data for RPC requests, and decoding data for RPC replies.

- **Server-side DSM-CC Library Stubs**. These functional elements mediate between an RPC protocol and the Service Object Implementation, providing a programming interface to the application, while decoding data from RPC requests, and encoding data for RPC replies. Server-side Library Stubs are also known as Skeletons.
- **Server-side Service Gateway**. This Service Object Implementation acts as the entry point into a domain of Services, and is the root of its name space.
- **Client-Side Local Object**. This Object Implementation offers local functionality or simplified function call interface to more complex lower-layer network protocols. Local objects are also known as Pseudo Objects.

The system logical entities are replaceable functional elements that have standard interfaces between them. This standard defines the interfaces between these functional elements, as illustrated below.

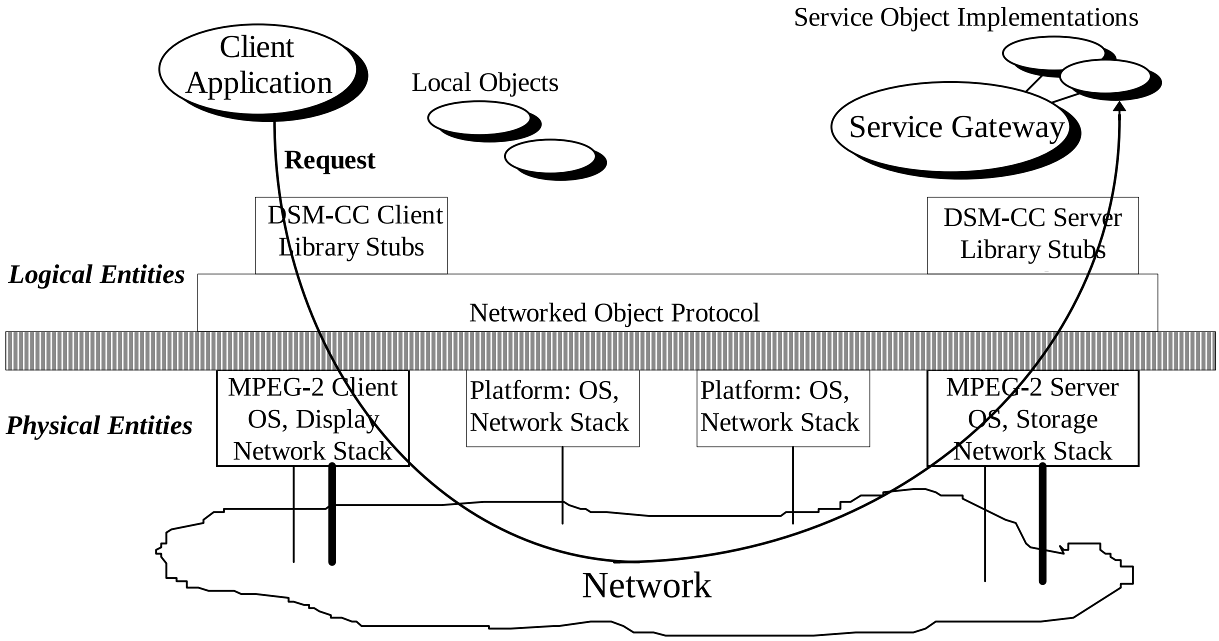


Figure 5-1 User-to-User System Environment

The hardware and software entities are logical components; they may exist on a single machine that supports all of the components, or they may be distributed across a set of special purpose machines. For example, requests for an individual media stream may be distributed across several machines in order to balance resource requirements even though a Client sees a single stream reference.

The Service Gateway presents to the Client a graph of service names and information, arranged as a hierarchy of Directories and Services. There may be one or more applications indicated by the graph. There is one top entry Service for each application, represented by the name of the application.

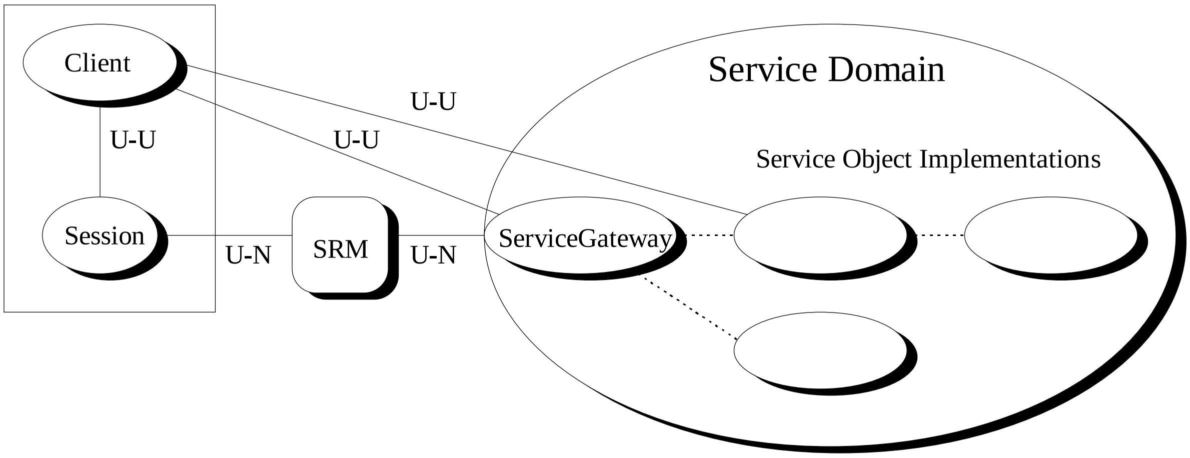


Figure 5-2 Logical Entity Relationships

The DSM-CC User-to-User Interfaces are designed as generic interface types that support access to multimedia Services and multimedia data. They are well-known basic building blocks for constructing more powerful object interfaces. DSM-CC supports single and multiple-inheritance, enabling new object interface definitions to include and build on the DSM-CC User-to-User interfaces.

In DSM-CC, the Name Service for objects is called a Directory. The Directory is a listing of names and corresponding objects. The operations of the Directory are given Access Roles, thus enabling certain Clients to create and populate a Directory hierarchy, while restricting other Clients to browsing and opening (resolving) a name to achieve the ability to communicate with an Object Implementation. A DSM-CC Directory Service can bind any kind of object, including application Services, Files, MPEG-2 Streams, Directories, and others. A Service may support (inherit) the Directory interface, thereby offering a directory hierarchy of objects. The hierarchy extends from a top node through a graph of sub-Directories, each serving to scope its own name space. All path names are therefore unique. A given object may be represented by more than one path name.

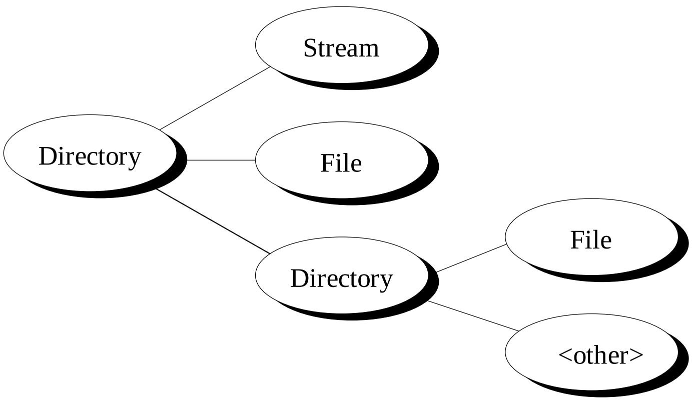


Figure 5-3 Multimedia Object Directory Hierarchy

5.2.3 Application and Service Interfaces

The Application Portability Interface is the interface between an application and the greater Client Operating System, which includes the DSM-CC Library, the processor operating system and the communications transport stack. The goal of the DSM-CC User-to-User Library is to provide Client applications a portable means of accessing Service objects (e.g. Directory, Stream and File). For most of the DSM-CC interfaces, operations translate directly 1-1 to Remote Procedure Calls, using a predictable, well-defined message assembly and data encoding rules. However, in some cases the interfaces supplied to the application by the DSM-CC Library system do not have a 1-1 mapping with the remote operations supported by Services. In other cases, the Object Implementation is completely local (i.e., there is no Remote Procedure Call).

In all cases, the DSM-CC Library must provide layered functionality, such as RPC message assembly, network lower layer protocols, or local Object Implementation. Therefore, the standard defines two interfaces for Service available to layers of the Client; the interface seen by the application and the interface supported by the Server, as shown in Figure 5-4.

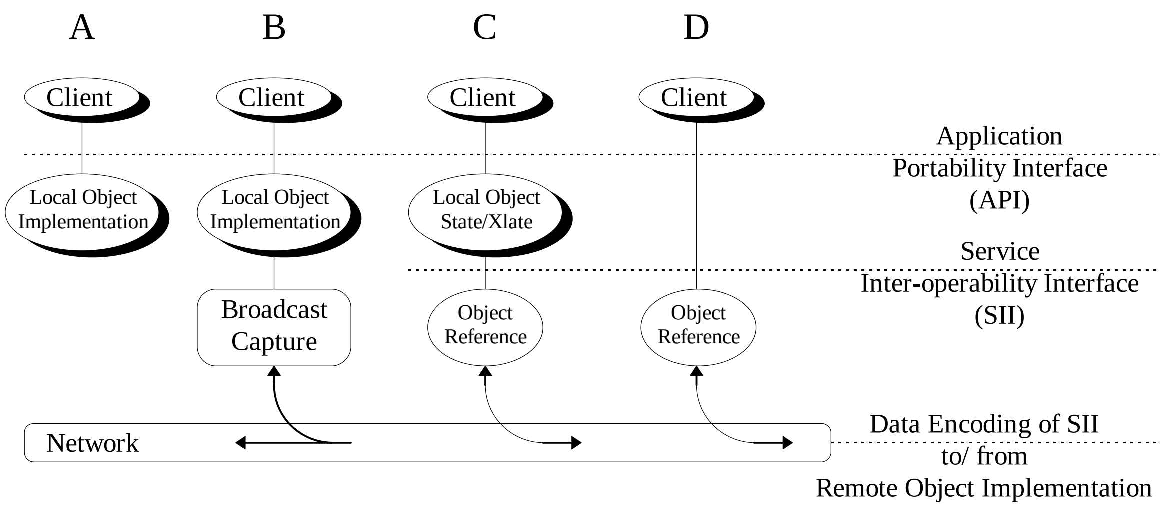


Figure 5-4 Models for Application and Service Interfaces

Figure 5-4 also illustrates four models for DSM-CC object access:

- Model A: The Client interfaces with a Local (Pseudo) Object. The Client makes requests across the Application Portability interface to invoke an operation that does not result in any network message sequence. These interfaces are specified under the heading **Application Portability (Local Library) Syntax.**
- Model B: The Client interfaces with a Local Object Implementation, which was received through a Network Broadcast. DSM-CC defines an Object Carousel for the broadcast delivery of objects.
- Model C: The Client interfaces with a Local Object implementation, which translates to a lower layer network message set, called the Service Inter-operability Interface. The operations are invoked on an object reference, which is local structure that acts as a proxy for the remote Service Object Implementation. The messages of the lower message set are 1-for-1 syntactically identical to the operations that are implemented at the remote Service. The local object interface specified under the heading **Application Portability (Local Library) Syntax**. The remote interface is specified under the heading **Service Inter-operability Syntax**.
- Model D: The Client interfaces directly through the Service Inter-operability Interface to the remote Service Object Implementation. In this case the Application Portability and Service Inter-operability Interfaces are one and the same, in addition to being 1-for-1 identical to the operations that are implemented at the remote Service. The layer between the SII and the Data Encoding is an RPC Stub that shall add a header with DSM-CC required parameters in the request and reply. The API presents the object reference as a pointer to a type with unknown structure, while the RPC Stub with SII interface defines the object reference structure for holding connection-related information. These interfaces are specified under the heading **Application Portability / Service Inter-operability Syntax.**

5.2.4 Categorization of Client Library Interface Sets

DSM-CC Library sets are defined in order to facilitate inter-operation. These are grouped as Consumer and Producer Clients of increasing capabilities. Core interfaces are defined as a foundation for minimal Clients. Extended interfaces are defined to facilitate portability and inter-operability.

5.2.4.1 Consumer Client

The Consumer Client will typically support a limited set of DSM-CC operations. Through download, it will add support for any additional operations needed to run an application. It may also store a basic set operations on a more permanent basis. The profile of a Client's DSM-CC User-to-User primitives at any given time is summarized not only by the interface name, but by the privileges (Access Role) given the Client End-User. The lowest Access Role that can be granted is that of READER. A READER cannot write to or update an object at the Server, it can only, navigate, open access to, and request delivery of objects and their data. The Consumer Client is defined as a READER with limited capabilities for saving application state, i.e., the ability to write files. Home set-top devices are usually classified as Consumer Clients.

5.2.4.2 Producer Client

A Producer Client, on the other hand, is granted OWNER privileges. In addition to the READER privileges, the OWNER can perform the create, put, write, bind, destroy functions, which enable loading of content to the Service, and ultimate removal from the Service. One kind of information provider is the author. The author, using one of many available multimedia authoring tools, will at times act in the capacity of loading content to the Service, and at other times will act as a consumer in the capacity of viewing and testing the application.

5.2.4.3 Client Library Profiles

The following list establishes an initial list of DSM-CC Library Profiles.

Core Consumer. This is the minimum-compliant Client. It supports READER operations for attaching/ detaching to/from a Service Domain, navigating Directory hierarchies, resolving names to object references, controlling MPEG-2 audio-video Streams, and reading and writing Files.

Core Producer. This Client supports all operations of the DSM-CC User-to-User Core interfaces.

Extended Consumer. This Client supports READER operations of an extended DSM-CC Library, plus the ability to write to a Database.

Extended Producer. This Client supports the complete DSM-CC Library.

5.2.5 Core Interfaces

The Core interfaces represent the most fundamental DSM-CC User-to-User Interfaces. These interfaces serve as a basis for Core Client configurations.

- **Base** Interface.

⇒ This interface provides the commonly used operations close() and destroy().

- **Access** Interface.

⇒ This interface provides commonly used attributes for size, history (version and date), lock status and permissions.

- **Stream** Interface.

⇒ This interface enables a Client to interactively control MPEG continuous media streams. It inherits **Base** and **Access** interfaces.

- **File** Interface.

⇒ This interface enables a Client to read and write opaque data (File contents) of an object. It inherits **Base** and **Access** interfaces.

- **Directory** Interface.

⇒ This interface provides a CORBA name Service interface plus operations to access objects and object data through depth and breadth-first path traversal. It additionally defines Access Roles for each of the inherited name Service operations. It inherits the **Access** interface.

- **BindingIterator** Interface.

⇒ This interface is defined by CORBA and is used for iteration through Directory lists.

- **Session** Interface.

⇒ This interface enables a Client to attach to a Service Domain.

- **ServiceGateway** Interface.

⇒ This interface inherits **Directory** and **Session** Interfaces.

- **First** Interface.

⇒ This interface provides function calls for an application to obtain root ServiceGateway and first Service. It is a Pseudo Object interface.

5.2.5.1 Core Client Application Portability Library:

level: interface:	Core Consumer API	Core Producer API
Base	close()	close() destroy()
Access	_get_Lock() _set_Lock()	_get_Size() _get_Hist() _set_Hist() _get_Lock() _set_Lock() _get_Perms() _set_Perms()
Stream	Base Consumer operations Access Consumer operations resume() pause() status() reset() play() jump() _get_Info() _set_Info()*	Base operations Access operations resume() pause() status() reset() play() jump() _get_Info() _set_Info()
File	Base Consumer operations Access Consumer operations read() _get_Content() _set_Content() write()	Base operations Access operations read() _get_Content() _set_Content() _get_ContentSize() write()
BindingIterator	next_one() next_n() destroy()	next_one() next_n() destroy()
Directory	Access Consumer operations list() resolve() open() close() get()	Access operations list() resolve() open() close() get() bind() bind_context() rebind() rebind_context() unbind() new_context() bind_new_context() destroy() put()
Session	attach() detach()	attach() detach()
ServiceGateway	Directory Consumer operations Session operations	Directory operations Session operations
First	root()	root()

	service()	service()

* Although Stream _set_Info() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.5.2 Core Client Service Inter-operability Library:

level: interface:	Core Consumer SII	Core Producer SII
Base	close()	close() destroy()
Access	_get_Lock() _set_Lock()	_get_Size() _get_Hist() _set_Hist() _get_Lock() _set_Lock() _get_Perms() _set_Perms()
Stream	Base Consumer operations Access Consumer operations resume() pause() status() reset() play() jump() _get_Info() _set_Info()*	Base operations Access operations resume() pause() status() reset() play() jump() _get_Info() _set_Info()
File	Base Consumer operations Access Consumer operations read() _get_Content() _set_Content() write()	Base operations Access operations read() _get_Content() _set_Content() _get_ContentSize() write()
BindingIterator	next_one() next_n() destroy()	next_one() next_n() destroy()
Directory	Access Consumer operations list() resolve() open() close() get()	Access operations list() resolve() open() close() get() bind() bind_context() rebind() rebind_context() unbind() new_context() bind_new_context() destroy() put()

* Although Stream _set_Info() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.6 Extended Interfaces

The Extended Interfaces establish public, standard operations for commonly used functionality. These interfaces may be used individually, i.e. only the interface Stubs required by an application need be present in the Client.

- **Event** Interface.
  - ⇒ This interface provides operations by which a Client can subscribe/unsubscribe to events , to be sent as Stream Event Descriptors in the MPEG stream.
- **Download** Interface.
  - ⇒ This interface provides a function call interface for the state machine and message encodings of the Chapter 7 Download specification. It is used to cause the transfer of Client interface stubs and Client application from a Download Service.
- **Composite** Interface.
  - ⇒ With this interface, versioned objects can be associated as a set, and opened with a single Directory open() invocation.
- **View** Interface.
  - ⇒ The interface enables a Client to sort and filter objects by their attributes using standard SQL statements.
- **State** Interface.
  - ⇒ This interface enables a Client to suspend and resume application state.
- **Interfaces** Interface.
  - ⇒ The interface provides methods for publishing and verifying new interfaces, in order to insure consistent and complete interface definitions in a DSM-CC environment.
- **Security** Interface.
  - ⇒ The interface provides a method to associate the passing of authentication parameters with **Session attach()**, **Directory open( )**, **Directory resolve()**, **Directory get()** , or other operations requiring authorization. It is a Pseudo Object interface.
- **Config** Interface.
  - ⇒ This interface provides an API by which RPC can be configured for either synchronous or asynchronous operation. It is a Pseudo Object interface.
- **LifeCycle** Interface.
  - ⇒ This interface is used by implementations for creation of objects, to insure unique object references in a DSM-CC environment. It is a Pseudo Object interface.
- **Kind** Interface.
  - ⇒ This interface provides operations to determine which interfaces are supported by an object. It is a Pseudo Object interface.

5.2.6.1 Extended Client Application Portability Library:

level: interface:	Extended Consumer API	Extended Producer API
Core interfaces (all)	Core Consumer API operations	Core Producer API operations
Download	info() alloc() start() cancel()	info() alloc() start() cancel()
Event	subscribe() unsubscribe() notify() _get_EventList() _set_EventList()*	subscribe() unsubscribe() notify() _get_EventList() _set_EventList()
Composite	list_subs()	list_subs() bind_subs() unbind_subs()
View	query() read() execute() _get_Style()	query() read() execute() _get_Style()
State	suspend() resume()	suspend() resume()
Interfaces		define() check() show() undefine()
Security	authenticate()	authenticate()
Config	wait() inquire() _get_DeferredSync() _set_DeferredSync() _get_ActiveRequests()	wait() inquire() _get_DeferredSync() _set_DeferredSync() _get_ActiveRequests()
LifeCycle		create()
Kind		is_a() has_a()

* Although Event _set_EventList() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.6.2 Extended Client Service-interoperability Library:

level: interface:	Extended Consumer SII	Extended Producer SII
Core interfaces (all)	Core Consumer SII operations	Core Producer SII operations
SessionUU	attach() detach()	attach() detach()
ServiceGatewayUU	ServiceGateway AccessRoles** Directory Consumer operations	ServiceGateway AccessRoles** Directory operations
SessionSI	attach() detach()	attach() detach()
ServiceGatewaySI	ServiceGateway AccessRoles** Directory Consumer operations SessionSI operations	ServiceGateway AccessRoles** Directory operations SessionSI operations
DownloadSI	info() proceed() cancel()	info() proceed() cancel() install() deinstall()
Event	subscribe() unsubscribe() _get_EventList() _set_EventList()*	subscribe() unsubscribe() _get_EventList() _set_EventList()
Composite	list_subs()	list_subs() bind_subs() unbind_subs()
View	query() read() execute() _get_Style()	query() read() execute() _get_Style()
State	suspend() resume()	suspend() resume()
Interfaces		define() check() show() undefine()

* Although Event _set_EventList() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

** The ServiceGateway redefines AccessRoles for Producer operations inherited from Directory.

5.3 Overview of the Interface Definition Language(IDL)

The DSM-CC User-to-User interfaces are defined in Object Management Group (OMG) Interface Definition Language (IDL). The complete IDL syntax, grammar, language mappings and related topics are contained in “The Common Object Request Broker: Architecture and Specification,” Revision 2.0, July 1995, from the Object Management Group. This specification is commonly referred to as CORBA 2.0. The OMG IDL is also standardized in ISO/IEC 14750.

IDL is a language for specifying interfaces. It is not a programming language, and therefore does not provide any procedural syntax. It is object-oriented, supporting the concepts of encapsulation and inheritance. It defines basic types such as char, short, long, and constructed types such as sequence (which is used to specify variable-length arrays). The scoping is much like ANSI C++. In IDL, a module contains interfaces, which in turn contain operations. (in ANSI C++, a namespace contains classes, which in turn contain methods).

```
module module_name {
    types
    ...
    interface interface_name {
        types
        attributes
        operations
        ...
    };
    other_interfaces
    ...
};
```

Figure 5-5 IDL File Format

Interfaces can be inherited with the syntax:

```
module module_name {
    interface interface_name :
        inherited_interface_name,
        other_inherited interface_name{
    ...
    };
};
```

Figure 5-6 Inheriting Interfaces with IDL

The inheritance feature of IDL enables new interfaces to be rapidly developed from basic ‘building block’ interfaces. All operations of the inherited interface are automatically included in the new interface. For example, the DSM-CC ServiceGateway interface inherits Directory and Session interfaces with the following IDL:

```
module DSM {
        interface ServiceGateway : Directory, Session {
        };
};
```

IDL is used to define interfaces in a generic way, independent of language and network protocol. CORBA 2.0 provides rules which map the IDL to C, C++ and Smalltalk languages. Given an IDL specification, the rules of the language mapping shall yield the declarations for constants, type definitions for all data types and function call prototypes for all operations. For example, the type struct in IDL is identical in syntax to the C language `struct`. OMG may define other language mappings in the future, which shall be considered normative as well.

In addition to the language mapping, rules for data encoding must be defined for each of the common IDL types. A data encoding is a flat sequence of octets that can be transferred between Client and object implementation using an inter-process control or network transport mechanism. Where the language mapping offers structure with accessible members, the data encoding is a contiguous, unparsed buffer. CORBA 2.0 provides a data encoding called Common Data Representation (CDR). DSM-CC Informative Annex C provides a summary of IDL to CDR, and in addition provides IDL to External Data Representation (XDR), the data encoding used by Open Networked Computing (ONC).

CORBA 2.0 also specifies a structure called an Interoperable Object Reference (IOR), which at a minimum contains a an object type id, a Protocol Profile Identifier, and an encapsulation of connection information. The Protocol Profile Identifier indicates the RPC, the data encoding and the structure of the encapsulation. The information in the IOR is used for all operation invocations on the Client/Service connection. DSM-CC has registered a range of Protocol Profile Identifiers with OMG, and defines several Protocol Profiles(identifier and associated encapsulation) for use in DSM-CC environments.

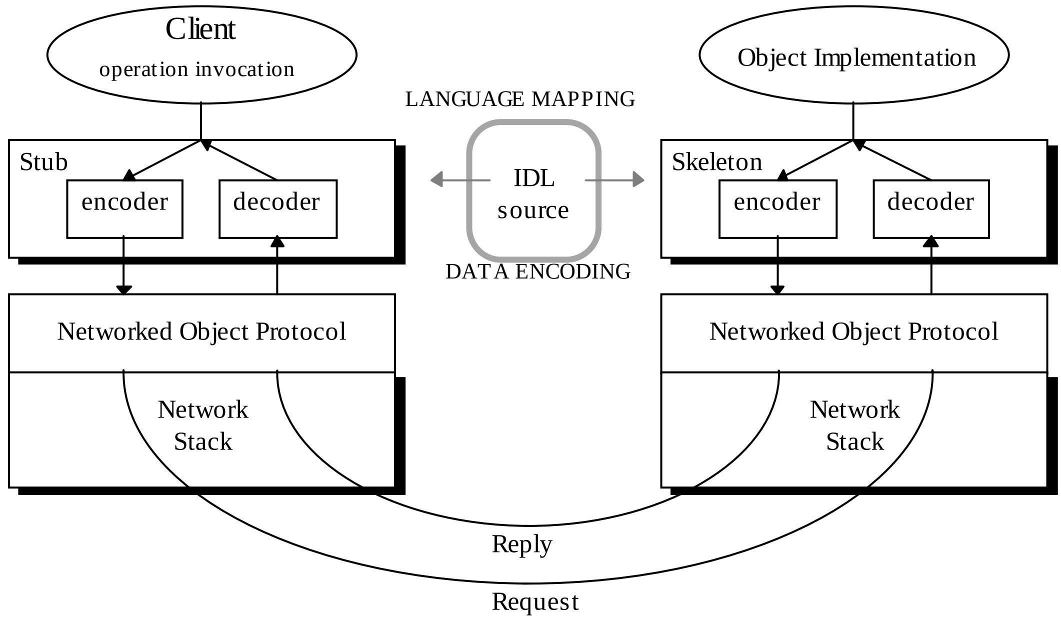


Figure 5-7 IDL to Entity Relationship

An IDL compiler is used to generate the Client Stub and a corresponding Server Stub, also known as the Skeleton. The upper interface of the Stub and Skeleton, as shown in figure 5-7, presents the language mapping interface to the programmer. The lower interface presents the data encoding to the Networked Object Protocol. In the case of a full CORBA system, this is the Object Request Broker (ORB). In other systems, it may simply be an RPC request/reply mechanism.

5.3.1 Operations

In IDL, operation parameters are specified to be either input (in), output (out), or both input and output (inout). In an underlying RPC implementation, input parameters are placed in the RPC request, and output parameters are returned in the RPC reply.

For example, the DSM-CC Event subscribe() operation is defined as follows:

```
module DSM {
        exception INV_EVENT_NAME  ExceptUser;
        interface Event {
                void subscribe (in string aEventName, out u_short eventId)
                        raises (INV_EVENT_NAME);
        }:
};
```

The exception INV_EVENT_NAME is declared using the exception declaration. The exception declaration is like a C language `struct` declaration. In addition to having an exception structure defined by the DSM Common macro ExceptUser, it will have a string id, e.g., "ex_DSM_INV_EVENT_NAME". All operations can return any of the standard CORBA System exceptions. The raises declaration identifies any additional user exceptions the operation can return.

The subscribe() operation is in the Event interface. It returns void. It has an string input argument that identifies an Event, and an unsigned short output argument that is a token associated with the Event. If the input argument is not valid, The Event subscribe() operation will return the INV_EVENT_NAME exception. Note: the Event interface is fully defined later in this Chapter.

5.3.2 Attributes

Within an interface, the attribute declaration can be used to define an identifier of a certain data type with a pair of implicit operations, one to set the value of the attribute, and the other to retrieve the value of the attribute. The readonly attribute declaration can be used to define an attribute that can only be retrieved by a Client.

For example, the Event interface declares an EventList attribute as an array of strings, as follows:

```
module DSM {
        exception INV_EVENT_NAME  ExceptUser;
        interface Event {
                typedef sequence<string> EventList_T;
                        attribute Event_List_T EventList;
        }:
};
```

5.3.3 Language Mapping

At the programing level, the C language mapping for the above Event subscribe() IDL is:

```
void DSM_Event_subscribe
      (DSM_Event object, CORBA_Environment * ev,
            CORBA_string aEventName, DSM_u_short * eventId)
```

Two arguments are inserted in the C mapping. The first is an input argument pointing to the object that will receive the request. This pointer represents a structure that normally contains addressing or local context information. The second is an output argument that will hold the exception , if there is one.

The EventList attribute produces two operations in the C mapping:

```
DSM_Event_EventList_T DSM_Event__get_EventList
            (DSM_Event object, CORBA_Environment * ev);

void DSM_Event__set_EventList
            (DSM_Event object, CORBA_Environment * ev
                  SM_Event_EventList_T * EventList);
```

A C++ language mapping produces a name space DSM and within it, an Event class. In the Event class will be the methods for the operations and attributes defined in the IDL.

note: In cases where the mapping of basic types in DSM-CC disagrees with that of OMG, the OMG mapping shall take precedent.

5.3.4 Encoding

Over the network, the above Event subscribe() becomes two messages, the request and reply of the Remote Procedure Call (RPC). A Protocol Profile (identified by a profileId in the IOR) is determined at the time the Client first obtains the IOR. This Protocol Profile establishes the data encoding to be used. For example, the UNO RPC uses Common Data Representation (CDR) encoding. For each IDL type, the encoding specifies the octet sequence to be carried over the network. DSM-CC Protocol Profiles are defined later in the Service Inter-operability section of this Chapter.

The RPC protocol defines message headers that provide a common transaction id for the request/reply pair, an operation id, etc. UNO defines the RPC and data encoding on the Inter-operability interface for a UNO RPC stack. ONC defines the RPC and data encoding on the inter-operability interface for an ONC RPC stack. Appendix G defines the UNO messages in IDL. Annex C defines UNO and ONC RPC on-the-wire message header mappings. The default for DSM-CC, unless otherwise specified by the Protocol Profile, is UNO/CDR.

The in arguments of the operation are placed in the request message body by the Client, in the order that they were specified in the IDL. In the case of Event subscribe(), this consists of one argument, a string. While at the programming interface, the string consisted of a pointer to a char(an array of char, null-terminated), a CDR encoding shall place a 4 byte length followed by the char array (null-terminated) in the byte stream.

The out arguments of the operation are placed in the reply message body by the object implementation responding to the request, again in the order that they were specified in the IDL. In this case, the reply message body consists of 2 bytes containing the eventId.

5.3.5 Typographical Conventions

In general, The DSM-CC User-to-User IDL specification is bordered, to separate it from ordinary text.

The type styles shown below are used in this Chapter to distinguish programming statements from ordinary text, and to distinguish OMG IDL specifications, DSM-CC IDL specifications and language mappings. However, these conventions are not used in tables or section headings, where no distinction is necessary, nor are the type styles used in text where their density would be distracting.

Helvetica	OMG IDL, CORBA language and syntax elements.
Times New Roman	DSM-CC User-to-User IDL specification in a border.
Times New Roman (bold)	DSM-CC User-to-User language and syntax elements in sentence text.
`Courier`	C language language and syntax elements.

5.3.6 Syntactical Conventions

The Core and Extended interfaces are scoped in module **DSM**. At the **DSM** level of scoping, Common DSM-CC data type definitions will be shared by Application and Service Inter-operability interfaces.

A Service Inter-operability Interface shall have the suffix **SI**, if it differs syntactically from the corresponding Application Portability Interface. The U-N Session Inter-operability Interface shall have the suffix **UU**.

#ifdef DSM_GENERAL shall enclose Producer Client operations. IDL Compilers can therefore generate efficient Consumer Client Stubs as the default case **(DSM_GENERAL** not defined). With **DSM_GENERAL** defined, IDL Compilers shall generate complete interface Stubs

#ifdef DSM_PSEUDO shall enclose local operations where they need to be separated from client/service operations.

#ifdef DSM_CONSUMER enables compilation of producer-only operations (without consumer operations).

The terms **TRUE** and **FALSE** shall indicate boolean 1 and 0, respectively.

5.4 Common Definitions

5.4.1 Basic Types

The following IDL specifies basic types. The basic type definitions of the language mapping may require modification to satisfy the requirements of the host processor and Operating System. For example, one system may define a long as a 32 bit quantity, whereas another may define it as a 64 bit quantity. OMG IDL defines the following ranges for integer data types:

short	$-2^{15} .. 2^{15}$
unsigned short	$0 .. 2^{16}$
long	$-2^{31} .. 2^{31}$
unsigned long	$0 .. 2^{32}$
longlong	$-2^{63} .. 2^{63}$
unsigned longlong	$0 .. 2^{64}$

```
module DSM {
    // machine-independent basic types
    //
    typedef short  s_short;                    // 16 bit signed integer
    typedef unsigned short u_short;            // 16 bit unsigned integer
    typedef long s_long;                       // 32 bit signed integer
    typedef unsigned long u_long;              // 32 bit unsigned integer
    //
    // note: this longlong is present to satisfy present OMG compilers
    // that do not support CORBA 2.0 extensions
    // If the IDL compiler cannot handle unsigned long long, treat it as an array
    // of 2 unsigned long where big-endian has MSB in octet 0 and LSB in octet 7
    // and little-endian has MSB in octet 7 and LSB in octet 0
    //
    // typedef s_long  s_longlong[2];                  // 64 bit signed integer
    // typedef u_long  u_longlong[2];                  // 64 bit unsigned integer
    //
    // else, if the IDL compiler supports CORBA 2.0 extensions
    // use the following two statements
    //
    typedef long long s_longlong;                     // 64 bit signed integer
    typedef unsigned long long u_longlong;            // 64 bit unsigned integer
};
```

5.4.2 Entity Identification

These types are used to hold information describing logical entities of the system.

The object reference(ObjRef) is defined as type CORBA Object. The Client Application has an opaque notion of object references, and is insulated from the representation of them (i.e., cannot determine any information in their structure). The Client merely receives the object reference in response to a resolve or open operation, and then invokes other operations using the object reference. It has no need to interpret or parse the object reference. The Client Library Stub shall define the Object structure and can use it to hold network addressing information or local state.

An opaque type is defined for holding a sequence of octets. Opaque values are used read and write File contents, to hold suspended application state information (UserContext), to hold the identification of the End-User (Principal), and for other values where the structure of the data does not need to be statically specified.

Version is used for identification of Object Implementation variations, and for compatibility verification.

ServiceLocation is used to contain parameters needed to attach to a Service Domain. **ServiceDomain** is in NSAP address format. For an interactive Session, it is the globally unique Server network address. For a Broadcast Carousel, it is the unique identifier of the Carousel. **pathName** provides the logical path in the network system namespace. **initialContext** provides application state enabling the resumption of an application at a given point.

```
module DSM {
    // entity identification
    //
    //  ObjRef is received by the Client in the reply to a resolve
    //  Following that, operations can be invoked against it
    //
    typedef Object ObjRef;                  // implemented as void * for C mapping
    typedef sequence<ObjRef> ObjRefs;
    typedef sequence<octet> opaque;
    typedef opaque UserContext;             // context for application state
    typedef opaque Principal;               // system-wide identification of end user
    struct Version {char aMajor; char aMinor;};
    typedef sequence<octet, 20> ServiceDomain;      // ServerId NSAP
    struct ServiceLocation {
                ServiceDomain aServiceDomain;
                CosNaming::Name pathName;       // path name to resolve
                UserContext  initialContext;    // starting user context
    };
};
```

5.4.3 Interface Identification

DSM-CC identifies interfaces by two means, enumeration and string values. The IFKind is an unsigned long used to identify an interface. For the situation where the interface inherits other interfaces, the IntfCode is defined. The IntfCode identifies the numeric and string value of the most derived interface, and lists the IFKinds for the inherited interfaces. The IntfCode provides a repository identifier to uniquely scope the source of the interface definition. For object identification in CORBA mechanisms, string ids are used in the form “<Module>::<Interface>“. For instantiable objects, 3-character string alias values are reserved, in order to reduce the size of Interoperable Object References.

```
module DSM {
    // interface identification
    //
    typedef u_long IFKind;
    typedef sequence<IFKind>  IFKindList;
    struct IntfCode {
        u_long anIFKind;            // standard DSM-CC IFkind enumeration
        string kind;                // DSM-CC format, "<Module>::<Interface>" or alias
        string repositoryId;        // id of repository where defined
        IFKindList includes;        // IFKinds of  inherited  interfaces
    };
    // The following are Service Tags registered with OMG for identifying interfaces
    // The reserved range for ISO/IEC WG11 in OMG is 0x49534F00 - 0x49534F7F
    // DSM interface IFKinds
    //
    const u_long  ik_null = 1230196480; // = 0x49534F00
    const u_long  ik_Base = 1230196481; // = 0x49534F01
    const u_long  ik_Access = 1230196482; // = 0x49534F02
    const u_long  ik_Stream = 1230196483; // = 0x49534F03
    const u_long  ik_File = 1230196484; // = 0x49534F04
    const u_long  ik_BindingIterator = 1230196485; // = 0x49534F05
    const u_long  ik_NamingContext = 1230196486; // = 0x49534F06
    const u_long  ik_Directory = 1230196487; // = 0x49534F07
    const u_long  ik_Session = 1230196488; // = 0x49534F08
    const u_long  ik_ServiceGateway = 1230196489; // = 0x49534F09
    const u_long  ik_First = 1230196490; // = 0x49534F0A
    const u_long  ik_Download = 1230196491; // = 0x49534F0B
    const u_long  ik_Event = 1230196492; // = 0x49534F0C
    const u_long  ik_Composite = 1230196493; // = 0x49534F0D
    const u_long  ik_View = 1230196494; // = 0x49534F0E
    const u_long  ik_State = 1230196495; // = 0x49534F0F
    const u_long  ik_Interfaces = 1230196496; // = 0x49534F10
    const u_long  ik_Security = 1230196497; // = 0x49534F11
    const u_long  ik_Config = 1230196498; // = 0x49534F12
    const u_long  ik_Lifecycle = 1230196499; // = 0x49534F13
    const u_long  ik_Kind = 1230196500; // = 0x49534F14
    const u_long  ik_SessionUU = 1230196501; // = 0x49534F15
    const u_long  ik_SessionSI = 1230196502; // = 0x49534F16
    const u_long  ik_DownloadSI = 1230196503; // = 0x49534F17
    //
    // aliases for string names
    //
    const string alias_Directory = “dir”;
    const string alias_BindingIterator = “bit”;
    const string alias_Stream = “str”;
    const string alias_File = “fil”;
    const string alias_Session = “ses”;
    const string alias_ServiceGateway = “srg”;
    const string alias_Interfaces = “inf”;
    const string alias_View = “viw”;
    const string alias_Download = “dnl”;
    const string alias_Event = “evt”;
};
```

5.4.4 Access Roles for Operations

An Access Role shall be associated with every DSM-CC operation, and with get and set operations individually on each attribute. An End-User Client (e.g., human) of a Service Domain shall also be given one or more Access Roles. In order for an operation request to be accepted, the Access Role of the End-User Client must satisfy the Access Role requirements of the operation. DSM-CC defines the following Access Roles:

- READER. A READER has read-only access to an object.
- WRITER. A WRITER can write to or update the state of an object. A WRITER also has READER privileges.
- BROKER. A BROKER can authorize access by a Client to an object. A BROKER also has WRITER privileges.
- OWNER. An OWNER can create and destroy objects. An OWNER also has BROKER privileges.
- MANAGER. A MANAGER has full system administration authority. A MANAGER also has OWNER privileges.

If not specified, the Access Role (ACR) for interface operations and attribute get operations shall default to READER. If not specified, the Access Role (ACR) for attribute put (set) operations shall default to WRITER.

A Service Domain may define additional Access Roles for further identification of an End-User Client's capabilities.

```
Module DSM {
    // Access Roles for operations
    //
    typedef char AccessRole;
    const AccessRole READER = 'R';
    const AccessRole WRITER = 'W';
    const AccessRole BROKER = 'B';
    const AccessRole OWNER = 'O';
    const AccessRole MANAGER = 'M';
};
```

5.4.4.1 Syntax for Access Control

The following rules enable the implementation of access control for DSM-CC:

Each operation must have an associated invocation privilege OWNER, MANAGER, BROKER, WRITER or READER. This is defined by means of the Access Control Role (ACR) definition form:

const char <operation name>_ACR = <access role>;

For example, **Stream resume** has the following Access Control Role definition:

```
module DSM {
    interface Stream {
        const char resume_ACR = READER;
    };
};
```

Each attribute shall have separate Access Control Role for both get and put operations against it. If the CORBA _set operation is used, the DSM put invocation privilege applies. If the CORBA _get operation is used, the DSM get invocation privilege applies. This is defined by means of the Access Control Role (ACR) definition form:

const char <attribute name>_put_ACR = <access role>;
const char <attribute name>_get_ACR = <access role>;

For example, the Stream Info attribute has the following Access Control Role definitions:

```
module DSM {
    interface Stream {
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
    };
};
```

5.4.5 Exceptions

An exception is an indication that an operation request was not performed successfully. The DSM-CC U-U operations are considered to be atomic. If an exception is received, all output parameters of the operation are invalid. An exception may be accompanied by additional, exception-specific information. Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exception condition has occurred during the performance of a request.

The standard OMG System exceptions are used by DSM-CC. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard System exceptions are not listed in raises expressions. The OMG IDL reference standard has a complete description of how exceptions are handled, plus code examples of C mappings. For example, the CORBA COMM_FAILURE exception has the string id "CORBA::COMM_FAILURE".

Each standard exception also includes a completion_status which takes one of the values { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES	The object implementation has completed processing prior to the exception being raised.
COMPLETED_NO	The object implementation was never initiated prior to the exception being raised.
COMPLETED_MAYBE	The status of the implementation completion is indeterminate.

The following are the standard CORBA System exceptions:

```
module CORBA {
  #define ex_body {u_long minor; completion_status completed;}
  enum completion_status {COMPLETED_YES, COMPLETED_NO,
                                   COMPLETED_MAYBE};
  enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};
  exception UNKNOWN            ex_body;    //the unknown exception
  exception BAD_PARAM          ex_body;    //an invalid parameter was passed
  exception NO_MEMORY          ex_body;    //dynamic memory allocation failure
  exception IMP_LIMIT          ex_body;    //violated implementation limit
  exception COMM_FAILURE       ex_body;    //communication failure
  exception INV_OBJREF         ex_body;    //invalid object reference
  exception NO_PERMISSION      ex_body;    //no permission for attempted op
  exception INTERNAL           ex_body;    //ORB internal error
  exception MARSHALL           ex_body;    //error marshalling param/result
  exception INITIALIZE         ex_body;    //ORB initialization failure
  exception NO_IMPLEMENT       ex_body;    //operation implementation unavailable
  exception BAD_TYPECODE       ex_body;    //bad typecode
  exception BAD_OPERATION      ex_body;    //invalid operation
  exception NO_RESOURCES       ex_body;    //insufficient resources for request
  exception PERSIST_STORE      ex_body;    //persistent storage failure
  exception BAD_INV_ORDER      ex_body;    //routine invocations out of order
```

```
  exception TRANSIENT              ex_body;    //transient failure - reissue request
  exception FREE_MEM               ex_body;    //cannot free memory
  exception INV_IDENT              ex_body;    //invalid identifier syntax
  exception INV_FLAG               ex_body;    //invalid flag was specified
  exception INTF_REPOS             ex_body;    //error accessing interface repository
  exception BAD_CONTEXT            ex_body;    //error processing context object
  exception OBJ_ADAPTER            ex_body;    //failure detected by object adapter
  exception DATA_CONVERSION        ex_body;    //data conversion error
};
```

The following are the DSM-CC User-to-User exceptions:

1. The **ALREADY_BOUND** exception indicates a Name is already in use. The Client should pick another Name. It is identical in syntax to the CORBA CosNaming AlreadyBound exception.
2. The **BAD_COMPAT_INFO** exception is returned in response to a request to initiate a Download sequence. Certain operations will always carry a DownloadInfoRequest, which provides a CompatibilityDescriptor and Download negotiation parameters to the Service. The CompatibilityDescriptor describes the Client's current hardware and software configuration. **BAD_COMPAT_INFO** is an indication of an unrecognized CompatibilityDescriptor. This feature is described in Chapter 6 of this specification.
3. The **BAD_MODULE_ID** exception indicates a module identifier value does not exist in the present configuration.
4. The **BAD_MODULE_INFO** exception indicates an incorrect Download install configuration.
5. The **BAD_SCALE** exception indicates an incorrect Scale value.
6. The **BAD_START** exception indicates the Stream start time does not exist.
7. The **BAD_STOP** exception indicates the Stream Stop value does not exist.
8. The **BLOCK_SIZE** exception indicates the requested block size is unacceptable to the Server.
9. The **BUF_SIZE** exception indicates the requested buffer size is unacceptable to the Server.
10. The **CANNOT_PROCEED** exception indicates a Directory did not have permission to resolve a node in a logical path. It is identical in syntax to the CORBA CosNaming CannotProceed exception.
11. The **ILLEGAL_SYNTAX** exception indicates an input string parameter contained incorrect syntax for IDL or SQL, or other expected language.
12. The **INV_CURSOR** exception indicates a cursor is out of range of the query results.
13. The **INV_EVENT_ID** exception indicates an Event token is not recognized.
14. The **INV_EVENT_NAME** exception indicates the Event name does not exist for this Service.
15. The **INV_KIND** exception indicates an interface kind is not recognized.
16. The **INV_NAME** exception indicates a path name is incorrectly formatted. It is identical in syntax to the CORBA CosNaming InvalidName exception.
17. The **INV_OFFSET** exception indicates the offset is outside the range of File contents.
18. The **INV_SIZE** exception indicates the size exceeds server or network limits.
19. The **MPEG_DELIVERY** exception indicates the Server was unable to deliver a multimedia object over an MPEG stream.
20. The **NO_AUTH** exception indicates the End-User has not provided the correct authentication in the request.
21. The **NO_QUERY** exception indicates a query has not yet been performed.
22. The **NO_REF_TYPE** exception indicates the referenced object or data type does not exist in this context.

23. The **NO_RESUME** exception indicates that previous application saved state cannot be recovered.
24. The **NO_SUSPEND** exception indicates the object cannot save application state.
25. The **NOT_DEFINED** exception indicates a prerequisite data type or interface has not been defined.
26. The **NOT_FOUND** exception indicates a logical path name does not exist, that an intermediate node is not a Directory, or that it could not resolve the object. It is identical in syntax to the CORBA CosNaming NotFound exception.
27. The **OPEN_LIMIT** exception indicates the number of active object references is the maximum allowed.
28. The **PREV_DEFINED** exception indicates a data type or interface has already been defined.
29. The **READ_LOCKED** exception indicates reads are temporarily prevented for an object.
30. The **TIMEOUT** exception indicates the maximum time for a transaction has been exceeded.
31. The **UNK_USER** exception indicates the Principal End-User is unknown to the Service Domain.
32. The **WRITE_LOCKED** exception indicates writes are temporarily prevented for an object.
33. The **SERVICE_XFR** exception indicates a resolve operation was unsuccessful, and provides an alternate Service Domain location where the requested Service can be resolved.

```
module DSM {
    // User-to-User common exceptions
    //
    #define ExceptUser {u_long minor; u_long completed;}
    //
    exception ALREADY_BOUND {};        // new Name conflicts with existing Name
    exception BAD_COMPAT_INFO ExceptUser;    // incorrect CompatibilityDescriptor
    exception BAD_MODULE_ID ExceptUser;        // Module Id out of range
    exception BAD_MODULE_INFO ExceptUser;    // Incorrect CompatibilityDescriptor
    exception BAD_SCALE ExceptUser;     //invalid scale
    exception BAD_START ExceptUser;     //stream does not contain this NPT
    exception BAD_STOP ExceptUser;      //invalid StopNPT, can never be reached
    exception BLOCK_SIZE ExceptUser;    // block size out of range
    exception BUF_SIZE ExceptUser;      // buffer size out of range
    exception CANNOT_PROCEED {          // unable to resolve node in path
        CosNaming::NamingContext cxt;
        CosNaming::Name rest_of_name;
    };
    exception ILLEGAL_SYNTAX {string aMessage;};  // unrecognized syntax
    exception INV_CURSOR ExceptUser;   //cursor out of bounds
    exception INV_EVENT_ID ExceptUser;            // invalid event id
    exception INV_EVENT_NAME ExceptUser;      // invalid event name
    exception INV_KIND {string aMessage;};  //type previously defined
    exception INV_NAME { };                 // incorrect name format
    exception INV_OFFSET ExceptUser;    // offset exceeds file size -1 for read
    exception INV_SIZE ExceptUser;      //size exceeds network limits
    exception MPEG_DELIVERY ExceptUser;        //error delivering MPEG stream
    exception NO_AUTH{            //not allowed to open without authentication
        u_long minor;
        u_long completed;
        opaque authData;
    };
    exception NO_QUERY ExceptUser;      // a query has not been performed yet
    exception NO_REF_TYPE  {string aMessage;};  //missing type definition
    exception NO_SUSPEND  ExceptUser;  // unable to suspend state
    exception NO_RESUME ExceptUser;    // unable to resume a previous session
    exception NOT_DEFINED {string aMessage;};  //type previously defined
    enum NotFoundReason { missing_node, not_context, not_object };
    exception NOT_FOUND {                // node in path not found
        NotFoundReason why;
        CosNaming::Name rest_of_name;
    };
    exception OPEN_LIMIT ExceptUser;    // too many resources are open
    exception PREV_DEFINED {string aMessage;};  // type previously defined
    exception READ_LOCKED ExceptUser;            // reads prevented
    exception TIMEOUT ExceptUser;        // transaction timed out
    exception UNK_USER ExceptUser;       // Principal unrecognized
    exception WRITE_LOCKED ExceptUser;           // writes prevented
    exception SERVICE_XFR {ServiceLocation transfer;};    // resolve failed, alternate location given
};
```

5.4.6 Stream and Event Synchronization

Application time and scale values are defined to enable monitoring of continuous media streams. **AppNPT** indicates absolute time value, and **Scale** indicates rate and direction(forward or reverse). These are used in the DSM-CC **Stream** and **Event** interfaces. For a further description, see the Stream interface section in this Chapter.

```
module DSM {
    // stream and event synchronization
    //
    struct AppNPT {s_long aSeconds; u_long aMicroSeconds;};      // Normal Play Time
    struct Scale {s_short aNumerator; u_short aDenominator;};    //+FF,-Rewind, Rate
};
```

5.5 Application Portability Interfaces(API)

DSM-CC specifies interfaces as either abstract or instantiable. An abstract interface is never used to define a complete object. Instead it provides operations that are designed be inherited. An instantiable interface defines the interface of the complete, realizable Object Implementation. The instantiable interface can inherit one or more abstract or instantiable interfaces.

The Service Domain must implement all Core interfaces completely in order to be DSM-CC compliant. Extended interfaces are optional. If the Server implements an interface, it must implement all of the declarations, type definitions, attributes, access types, operations and exceptions of the interface. The minimal Client (Core Consumer) need only implement the specified Core READER plus File write() operations. A Client can choose to support selected operation groups by Access Role. For example, a Client could elect to support only the READER group of an interface, meaning only those operations in the interface with READER Access Role.

5.5.1 Core Interfaces

The Core DSM-CC User interfaces are the minimum set that must be supported by a DSM-CC compliant ServiceGateway Domain. They include the abstract interfaces Base, Access, NamingContext, and the instantiable interfaces Stream, File, Directory, Session and ServiceGateway.

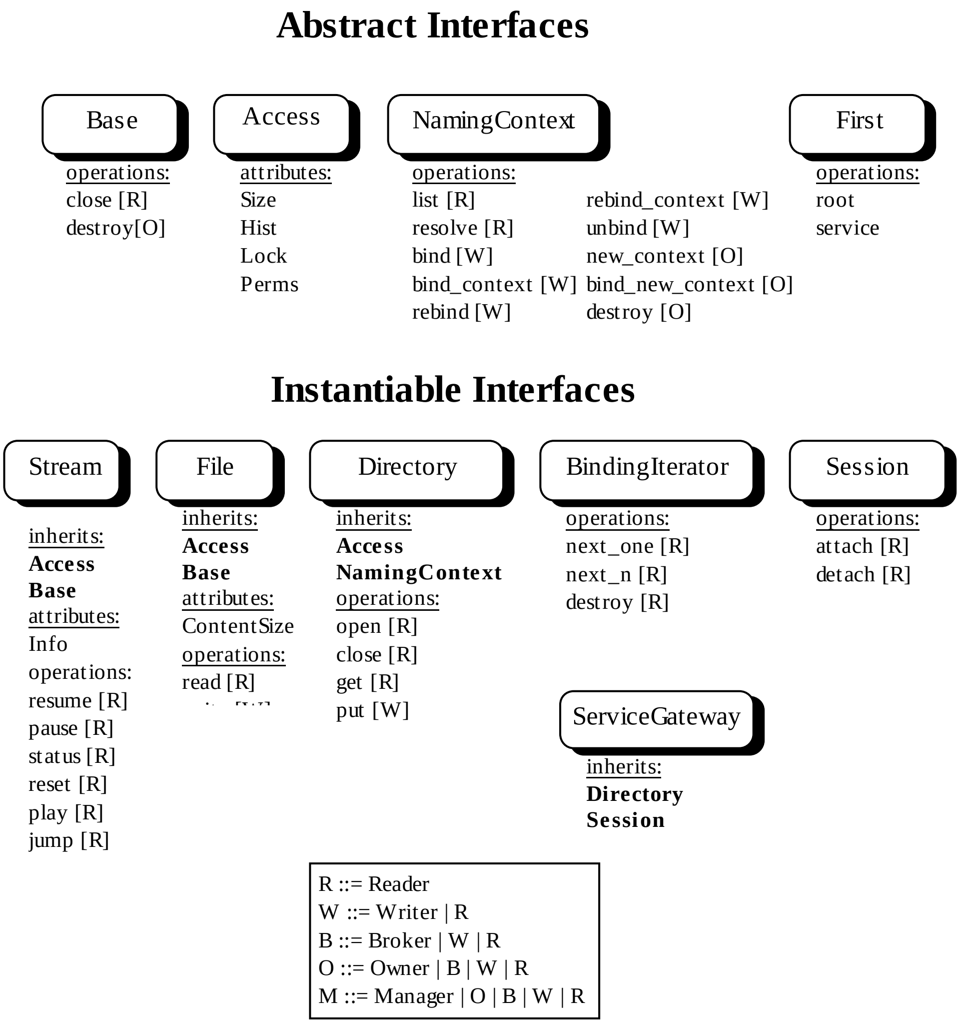


Figure 5-8 DSM Core Abstract and Instantiable Interfaces

5.5.1.1 Base

The Base interface provides common operations for deletion of DSM-CC object references and objects.

A Client (READER) may have obtained an object reference for an object, invoked some operations against it, and now has no more use for that object. The **Base close()** operation indicates the requesting Client no longer needs transient Client/Service connection-related resources associated with the object and shall not make any further requests.

When a Client (OWNER) wishes to delete the persistent data associated with an object, it may invoke the **Base destroy()** to destroy it, thus enabling the Object Implementation and related Services to free all state and storage resources associated with it. Following **destroy()**, the object shall cease to exist for all Clients.

5.5.1.1.1 Summary of Base Primitives

The following primitives are used by the interfaces of all objects.

close	Close a reference to an object. (READER)
destroy	Destroy an object instance. (OWNER)

5.5.1.1.2 DSM Base close

DSM Base close	Close a reference to an object. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
   interface Base {
            const AccessRole close_ACR = READER;
            void  close ();
    };
};
```

Semantics

Base close() is used by the Client to indicate that access to the object is no longer required. This is primarily a resource issue and is not specifically required; however, the total number of references allowed is limited, and well behaved Clients shall close whenever reasonable. If **OPEN_LIMIT** has been received when attempting to open an object reference, the Client shall need to close one or more other active references in order to free resources, before retrying the open.

Close means delete the object reference and therefore the Client's ability to communicate with the object.

If a parent Composite object is closed, its Child objects are closed as a result of the operation (See Composite interface description in the Extended Interfaces section).

Privileges Required:
READER

5.5.1.1.3 DSM Base destroy

DSM Base destroy	Destroy an object instance.

Application Portability / Service Inter-operability Syntax

```
module DSM {
   interface Base  {
#ifdef DSM_GENERAL
            const AccessRole destroy_ACR = OWNER;
            void destroy  ();
#endif
    };
};
```

Semantics

Base destroy() is used by the Client to delete a persistent Object Implementation. After this occurs, its reference will no longer be valid, and storage resources used for it will be freed.

Privileges Required:
OWNER

5.5.1.2 Access

The Access interface provides common description and access control attributes. These include size, version, date, lock and permissions attributes.

Principal End-User Clients are authorized to invoke operations through the following mechanisms:

- Capability Verification: A verification that the End-User has an Access Role (Capability) that matches or exceeds the Access Control Role of the operation.
    1. An Access Control Role is defined for each operation. e.g., OWNER, MANAGER, BROKER, WRITER or READER.
    2. The Principal End-User is the object OWNER, or a member of an authorized READER, WRITER, BROKER or MANAGER group in the Object Implementation's Service Domain.
- Authentication: Resolving an object or invoking selected operations may require a password/ PIN or encrypted key exchange. Access for all operations on an object can be setup to require secure transmissions of all messages to/from that object.

Authorization for access to Primary Services is performed by the Service Gateway or Directory of Services.

Authorization for access to an application's objects is performed by the Service or Directory in which the objects are bound..

An allSecure parameter can be set on an object which requires secure transmission for all messages to/from the object. It is expected that the lower layers of network protocol will perform this function via encryption or scrambling.

5.5.1.2.1 Setting Permissions

When an object is created the OWNER is associated with it. The OWNER may alter privileges by setting the permissions (Perms) attribute on an object to allow access by designated MANAGER, BROKER, WRITER or READER groups. Permissions are set by using **Directory put()** (where path specification is <object-name>, **Perms**).

A Client may be in more than one group. An object can be accessed by the OWNER and by more than one group in each of MANAGER, BROKER, WRITER or READER AccessRoles. Groups identifiers are scoped within arbitrary Domain boundaries, e.g., within the ServiceGateway, within a Directory, etc. The Client PrincipalId shall map to a set of groups in which that Client is a member. Corresponding authentication databases will vary with the implementation, and are not specified in this standard.

The **Perms** attribute shall identify groups that can access an object or invoke its operations. An object shall limit access to the Client who has the specified group or individual identification. Each method of that object shall further restrict invocation by allowing access by role and group. For example, if a method requires WRITER privileges, the Client must be in one of the WRITER groups specified in that object's **Perms** attribute. To open or resolve a target object, the Client must be in one of the READER groups specified in that object's **Perms** attribute.

The OWNER may set permissions to associate a password with an object, or to associate encrypt key data with an object. When a request to open such an object is given, an exception is returned. If encryption is required, the exception contains an encrypt key challenge. In order to continue, the End-User must send an **authenticate** containing the correct response to the encrypt key. If a password is required, the End-User must send an **authenticate** containing the correct password. The **authenticate** must be carried in the ServiceContextList of the repeated open, in order to be granted access.

The OWNER may set an **allSecure** flag in an object's **Perms** attribute, indicating that all messages to/from the object must be secure, e.g., encrypted or scrambled. From resolve time through close, the Service shall effect secure transmission through appropriate lower network layer messaging.

5.5.1.2.2 Access Definitions

Application Portability / Service Inter-operability Syntax

```
module DSM {
    struct DateTime {               // tm from ANSI C std.
        s_long tm_sec;              //  seconds, 0-59
        s_long tm_min;              // minutes, 0-59
        s_long tm_hour;             // hours, 0-23
        s_long tm_mday;             // day of the month, 1-31
        s_long tm_mon;              // months since Jan, 0-31
        s_long tm_year;             //years from 1900
        s_long tm_wday;             // days since Sunday, 0-6
        s_long tm_yday;             // days since Jan 31, 0-365
        s_long tm_isdst;};          // Daylight Savings Time indicator

    interface Access {
#ifdef DSM_GENERAL
        // size
        const AccessRole Size_get_ACR = READER;
        readonly attribute u_longlong Size;         // size of all attributes in octets;
#endif
        // history
        struct Hist_T {
                Version  aVersion;          // object version
                DateTime  aDateTime;};   // time created or last updated, GMT
#ifdef DSM_GENERAL
        const AccessRole Hist_get_ACR = READER;
        const AccessRole Hist_put_ACR = BROKER;
        attribute Hist_T Hist;                  // version and time of persistent object
#endif
        // lock status
        struct Lock_T {boolean readLock; boolean writeLock;};
        const AccessRole Lock_get_ACR = READER;
        const AccessRole Lock_put_ACR = WRITER;
        attribute Lock_T Lock;

        // permissions
        struct Perms_T {
                // the next 4 are binary masks of binary flags signifying
                // groups that can access the object
                u_short managerPerm;
                u_short brokerPerm;
                u_longlong writerPerm;
                u_longlong readerPerm;
                opaque owner;                       //owner identifier = Principal
                string aPassword;                   //PIN
                opaque authData;            //system-specific
                // instruct lower layers to implement a secure connection for this object
                boolean allSecure;};                // all methods parameters encrypted
#ifdef DSM_GENERAL
        const AccessRole Perms_get_ACR = OWNER;
        const AccessRole Perms_put_ACR = OWNER;
        attribute Perms_T Perms;
#endif
    };
};
```

5.5.1.3 Stream

Stream primitives are used to emulate VCR-like controls for manipulating MPEG continuous media streams. Streams differ from other datatypes in that, while in play mode, the rate of the stream delivery will be governed by an MPEG network flow control mechanism. Streams include datatypes such as video and audio, as defined by ISO/IEC 13818.

Stream pause() and **Stream resume()** behave much like their VCR counterparts. However, each primitive that initiates play mode includes a scale parameter which controls forward or reverse operation. Position is indicated in Normal Play Time (NPT), which indicates the stream absolute position relative to the beginning of the stream. Application NPT (AppNPT) is used for the application request interface and is specified in seconds and microseconds. MPEG Transport NPT is used to carry Normal Play Time in DSM-CC MPEG Stream descriptors. Transport NPT is specified in MPEG PTS 33 bit format. **Stream play()** enables play from a start NPT position until a stop NPT position is reached. **Stream jump()** provides capability to jump when a stop NPT position is reached to any start NPT position in the stream.

A stream is first requested using one of the resolve operations. It returns an object reference for the stream, to be used with subsequent stream commands. Streams open in **O** mode, representative of Pause with AppNPT value 0,0. In case an internal error occurs (e.g.: a disk failure) the stream goes back to **O** mode. More than one stream can be opened at a time.

Successful execution of Stream commands require that the Service execute them in the exact sequence that the Client has requested them. For example, **Directory open()** and **Stream resume()** can be sent in quick succession from the Client if play mode is desired immediately after the completion of the open. In this case, the operations could not be executed out of order, since at the Service, the results of one will feed as an input to the second.

At any given time, the Server will be in one of the following play modes for a given video delivery:

Open	Application NPT is 0,0. The Server is not transporting the stream.
Pause	The Server is not transporting the media stream
Search Transport	The Server is searching for start NPT. When at start NPT, it will transport the media stream.
Transport	The Server is transporting the media stream and will pause at end of stream.
Transport Pause	The Server is transporting the media stream and will pause at stop NPT.
Search Transport Pause	The Server is searching for start NPT. When at start NPT, it will transport the media stream until stop NPT.
Pause Search Transport	The Server is transporting the media stream. It will transport the media stream until stop NPT, then will search to start NPT and transport the media stream.
End of Stream	Server NPT is at the maximum NPT of the stream. The Server is not transporting the stream.
Pre Search Transport	This is an exceptional status. The Server is stopping the sending of the media stream. Once it has stopped, it transitions to Search Transport
Pre Search Transport Pause	This is an exceptional status. The Server is stopping the sending of the media stream. Once it has stopped, it transitions to Search Transport Pause

Stream status() is used to inquire as to the current **AppNPT**, Scale and Mode of an open stream.

5.5.1.3.1 Stream Definitions, Exceptions

```
module DSM {
    interface Stream : Base, Access {
        // stream modes
        typedef u_long Mode;
        const Mode OPEN_M = 0;
        const Mode PAUSE_M = 1;
        const Mode TRANSPORT_M = 2;
        const Mode TRANSPORT_PAUSE_M = 3;
        const Mode SEARCH_TRANSPORT_M = 4;
        const Mode SEARCH_TRANSPORT_PAUSE_M = 5;
        const Mode PAUSE_SEARCH_TRANSPORT_M = 6;
        const Mode END_OF_STREAM_M = 7;
        const Mode PRE_SEARCH_TRANSPORT_M = 8;
        const Mode PRE_SEARCH_TRANSPORT_PAUSE_M = 9;
        struct Stat {
                AppNPT rPosition;
                Scale rScale;
                Mode aMode;};
        struct Info_T {
                string<255> aDescription;
                AppNPT duration;
                boolean audio;
                boolean video;
                boolean data;};
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
        attribute Info_T  Info;
    };
};
```

Note: **Info** is not intended to be an attribute database, but rather to be a minimum set of stream identification and characteristics. Pertinent title information includes title and runtime length.

5.5.1.3.2 Normal Play Time Temporal Positioning

In order to support random positioning and a variety of play rates the Media stream primitives make use of a temporal addressing scheme called Normal Play Time (NPT). Intuitively NPT is the clock the viewer associates with a program. It is often digitally displayed on a VCR. NPT advances normally when in normal play mode (scale = 1/1), advances at a faster rate when in fast scan forward (high positive scale ratio), decrements when in scan reverse (high negative scale ratio) and is fixed in pause mode. NPT is roughly equivalent to SMPTE time codes. Application NPT is defined as two values representing seconds and microseconds.

To understand DSM-CC's NPT model one must separate the application's perspective of NPT from the underlying mechanism used to coordinate NPT between the Client and Server.

From the application's perspective NPT is a clock that is maintained in the Client operating system. It is used to request position relative to a specific program (i.e. "where are we?") or to control the position of the stream (i.e. "jump to this position"). Consider the following example: as a reference time assume that real time starts at 0 and progresses in seconds. Note that RPC-latency is assumed to be 0. Suppose that the application makes the following calls:

1. At 0 seconds Stream open is called. It is followed by Stream resume requesting that the stream begin playing at normal play rate with **AppNPT** start time 30.
2. At 10 seconds Stream pause is called. At this point **AppNPT** will be 40.

3. At 16 seconds Stream resume is called requesting that the stream continue playing at **AppNPT** = 80 at ten times normal speed.
4. At 26 seconds Base close is called. At this point **AppNPT** will be 180.

The coordination of NPT between the Server and the Client is independent of the API usage of **AppNPT**. There are two possible methods for maintaining NPT in the Client. The first method is to use NPT descriptors as described in the section entitled “NORMAL PLAY TIME.” The other method is to explicitly query the Server for **AppNPT** information. Due to latency considerations the second method may be less accurate.

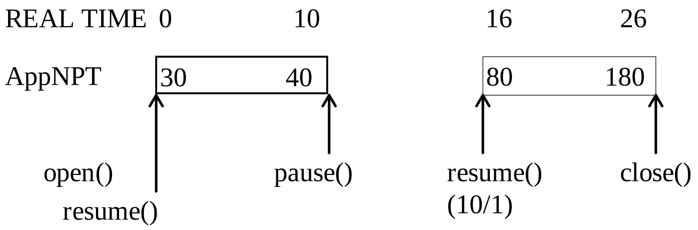
5.5.1.3.2.1 Application NPT Values

The following restrictions apply to **AppNPT** values:

- The beginning of the Stream corresponds to 0 **aSeconds**, 0 **aMicroSeconds**.
- **AppNPT** values ascend from the beginning of stream to end of stream.
- **AppNPT** pause now and resume now value is indicated by negative infinity (0x80000000) **aSeconds**. These represent the Server view of what “now” means, i.e. when the Server receives a pause or resume **AppNPT** specified as 0x80000000 aSeconds, it will pause or resume at the earliest possible time. This is the only negative value allowed.
- **AppNPT** is the addition of **aSeconds** and **aMicroSeconds**.
- **aMicroSeconds** shall be less than 1000000.

5.5.1.3.3 Summary of Stream Primitives

<u>Inherited from **Base**:</u>	
close, destroy	
<u>Inherited from **Access**:</u> attributes: **Size, Hist, Lock, Perms**	
<u>Defined in **Stream**:</u>	
attributes: **Info**	
DSM Stream pause	Stop sending stream when NPT position is reached. (READER)
DSM Stream resume	Start sending stream at NPT position within stream. (READER)
DSM Stream status	Obtain status of a stream. (READER)
DSM Stream reset	Reset the stream state machine. (READER)
DSM Stream jump	When stream reaches stop NPT, resume at start NPT. (READER)
DSM Stream play	Play stream from start NPT until stop NPT. (READER)



The above drawing illustrates an example of Stream primitives usage. In this example, time is shown in seconds for ease of explanation. The latency between Client and Server is not shown (it could be considerable). In response to a viewer input, **Directory open()** followed by **Stream resume()** commands are sent to start playing the video at normal play rate (scale 1/1), with **AppNPT** start time 30 seconds into the stream. 10 seconds later the viewer presses the VCR pause button, causing a **Stream pause()** command to be given. At this time and during the pause the video **AppNPT** remains at a point 40 seconds from the start of the stream. 6 seconds later, the viewer initiates fast forward with **AppNPT** start time 80 seconds into the stream, causing a **Stream resume()** command to be sent with scale = 10/1. Finally, the viewer quits, causing **Base close()** to be sent.

5.5.1.3.4 Stream State Machine

The stream interface provides operations to control the delivery of a media stream, through pause, resume, etc. commands. The Server is represented as a stream object that provides this interface and can transport MPEG over the network. Upon receipt of a command, it will modify its state machine, and perform the indicated function. Since each command is atomic, if any of the supplied parameters is incorrect the whole command is ignored.

The stream temporal status is comprised of current NPT, start NPT, stop NPT, mode and viewing rate (scale).

Note that the DSM-CC Library provides the Client stream status with the **status()** function. The status value which the set-top device provides allows the remote stream object to instrument the transport delay. The status value which the remote stream object returns allows the set-top device to instrument the transport delay. One application of the measurement is to configure transport buffers.

The time value which the Client provides with the **resume(rStart)** is the stream position at which to begin transport. The scale value describes both the direction (reverse is just a negative value) and the rate (normal Play is just a positive value of 1.0). The time value which the Client provides with the **pause(rStop)** is the stream position at which to suspend the transport. (There are transport mechanisms which require some message traffic to sustain the connection. It is implementation dependent, and not Client visible, how the source of the media stream stimulates the connection for such a transport solution. The obvious technique is to transmit the data stream, with its status fields.)

Since the Client can cascade methods, the sequence **resume(rStart)** plus **pause(rStop)** essentially equates to **play(rStart, rStop)**. Also the sequence **pause(rStop)** plus **resume(rStart)** essentially equates to **jump(rStop, rStart)**. However an exception to this general rule exist, in the case a **play(rStart, rStop)** is received from the Server while the state machine is in either **T, TP** or **PST** mode: this exception aims to accomodate the requirement of having **play(rStart, rStop)** actually meaning play from **rStart** to **rStop**. There is no corresponding exception for the **jump()** command. Moreover it is worth noting that as all methods are atomic, the sequence of a correct **resume()** plus an incorrect **pause()** would affect the state machine according to the **resume()**, while the single **play()** would not. Similarily, the sequence of a correct **pause()** plus an incorrect **resume()** would affect the state machine according to the **pause()**, while the single **jump()** would not.

5.5.1.3.4.1 State Machine

The interface controls a state machine. The state machine, shown below with the main transitions, comprises the ten states of a) Open b) Pause c) Transport d) Transport Pause e) Search Transport f) Search Transport Pause g) Pause Search Transport h) End of Stream i) Pre Search Transport and l) Pre Search Transport Pause.

The open causes the state to transition to the Open state. Open state is equivalent to Pause state at **AppNPT** 0.0. The default values are a) rStart=0x00000000.0x00000000, b) rStop=0xEFFFFFFF.0xFFFFFFFF, and c) Scale=1,1.

If the state machine receives a **pause(rStop)** while a stop NPT is still set (that is: has not been reached yet), the **pause(rStop)** replaces the previous stop NPT. If the state machine receives **resume(rStart)** while a start NPT is still set (that is: has not been reached yet), the **resume(rStart)** replaces the previous start NPT. If the state machine receives a **resume(rStart)** while in **T** mode, it stops sending the media stream and sets the start NPT. If the state machine receives a **play(rStart, rStop)** while in **T**, **TP** or **PST** mode, it stops sending the media stream and sets the start and stop NPTs.

The figure below shows the stream profile which corresponds to each state.

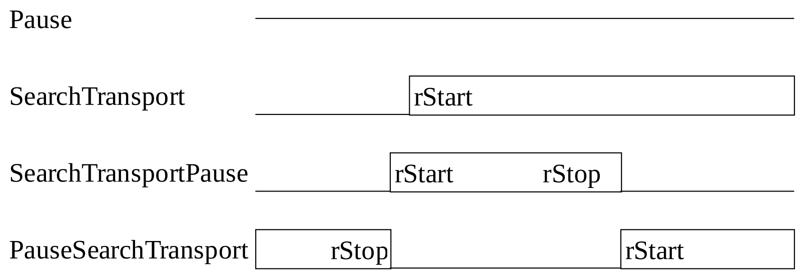


Figure 5-9 rStart and rStop State

O	Open	There is one phase . Application NPT is 0,0. The Server is not transporting the stream.
P	Pause	There is one phase .The Server is not transporting the media stream
ST	Search Transport	There are two phases with one time value to describe the transition. The Server is searching for start NPT. When at start NPT, it will transport the media stream. Since there is no rStop, the Server will continue to advance the stream and pause at End of Stream.
T	Transport	The Server is transporting the media stream. and will pause at End of Stream.
TP	Transport Pause	The Server is transporting the media stream and will pause at stop NPT.
STP	Search Transport Pause	There are three phases with two time values to describe the transitions. The Server is searching for start NPT. When at start NPT, it will transport the media stream until stop NPT. The common sequence which causes the state transition is either

		play(rStart, rStop) or **resume(rStart)** plus **pause(rStop)**.
PST	Pause Search Transport	There are three phases with two time values to describe the transitions. The Server is transporting the media stream. It will transport the media stream until stop NPT, then will search to start NPT and transport the media stream. The common sequence which causes the state transition is either **jump(rStop, rStart)** or **pause(rStop)** plus **resume(rStart)**.
EOS	End of Stream	Server NPT is at the maximum NPT of the stream. The Server is not transporting the stream.
PreST	Pre Search Transport	This is an exceptional status. The Server is stopping sending the media stream. Once it has stopped, it transitions to Search Transport
PreSTP	Pre Search Transport Pause	This is an exceptional status. The Server is stopping sending the media stream. Once it has stopped, it transitions to Search Transport Pause

The semantics of **Stream reset()** are to return to the Open state, which is a variation of the Pause state. The EndofStream is also a variation of the Pause state, where Application NPT is equal to the maximum NPT of the stream.

For Open and End of Stream states, the normal state transitions for Pause apply.

End of Stream Stream Mode descriptor shall be sent to the Client in the MPEG stream.

5.5.1.3.4.2 Basic State Machine

The interface below realizes the Basic State Machine with just the **Stream resume()** and **pause()** methods. For readibility reasons the figure does not include neither the **O** and **EOS** modes, as they behave exactly as **P**, nor the **reset()** method, as it is trivial and cause the state machine to return to the **O** mode. It also does not include neither the **PreST** and **PreSTP** modes nor the **resume(rStart)** method occurring in **T** mode, as they are better explained in the subsequent paragraph. The interface does not include **play(rStart, rStop)** because the function is almost identical to the sequence **resume(rStart)** plus **pause(rStop)**: the exception to this general rule is highlighted in the subsequent paragraph. The interface does not include **jump(rStop, rStart)** because the function is identical to sequence **pause(rStop)** plus **resume(rStart)**.

The Basic State Machine is shown below.

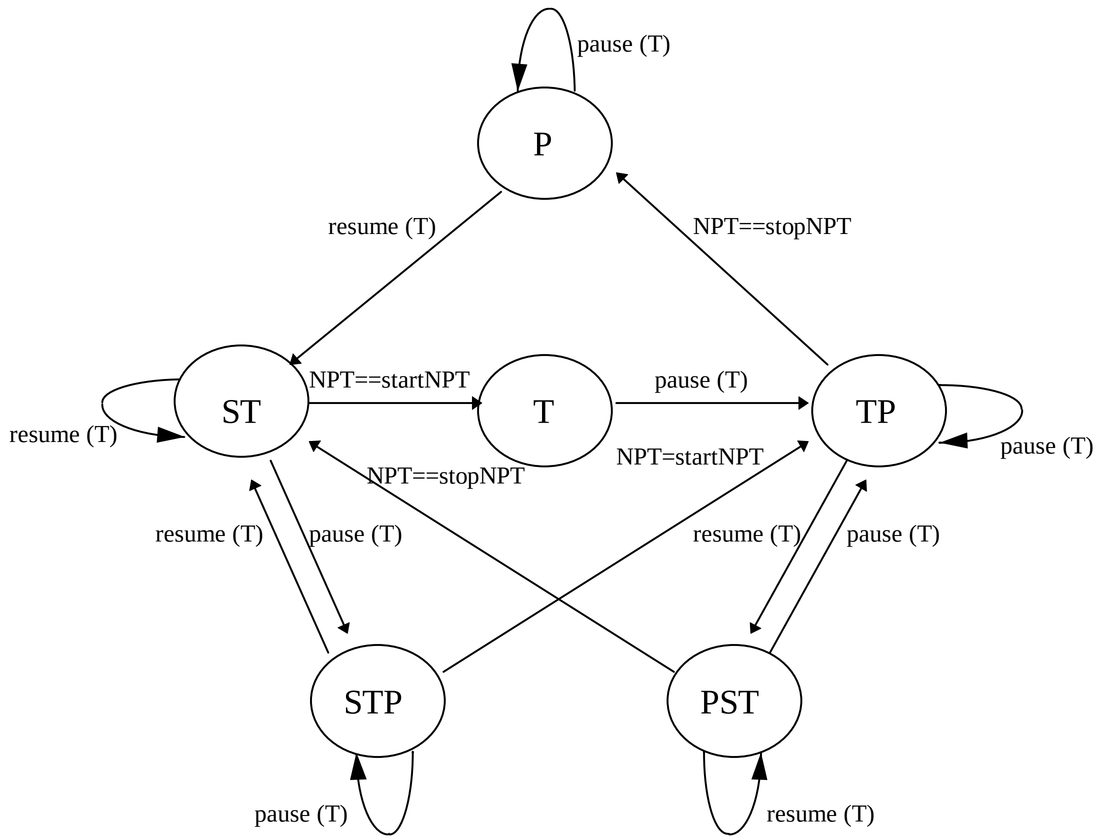


Figure 5-10 State Machine Transitions

The table below shows the transitions which the **resume()** and **pause()** method cause.

Previous State	resume()	pause()
ST	ST	STP
T	PreST	TP
TP	PST	TP
P	ST	P
STP	ST	STP
PST	PST	TP
PreST	PreST	PreSTP
PreSTP	PreST	PreSTP

The table below shows the transitions which the **startNPT** condition and the **stopNPT** condition cause.

Previous State	AppNPT = startNPT	AppNPT = stopNPT
O		
ST	T	
T		
TP		P
P		
STP	TP	
PST		ST

If the state was SearchTransport, the machine transitions to Transport when the stream position reaches the startNPT. If the state was SearchTransportPause, the machine transitions to TransportPause when the stream position reaches the startNPT. If the state was TransportPause, the machine transitions to Pause when the stream position reaches the stopNPT. If the state was PauseSearchTransport, the machine transitions to SearchTransport when the stream position reaches the stopNPT.

Transition due to startNPT or stopNPT cannot occur in PreSearchTransport or PreSearchTransportPause, as they are just defined as temporary modes entered by the State Machine while stopping sending the stream.

5.5.1.3.4.3 Complete state machine

There is an exceptional case where **play(rStart, rStop)** is not equivalent to the sequence **resume(rStart)** plus **pause(rStop)**: this happens when a **play(rStart, rStop)** is received while the Server is sending the media stream. The sequence **resume(rStart)** plus **pause(rStop)** would in essence cancel all previously set stop NPT and start NPT, and only set the stop NPT to **rStop**. This would cause the Server to continue playing up to **rStop**.

The above behavior does not correspond to the definition of the **play()** operation, thus a different transition needs to be defined. As the expected behavior is to cease sending the media stream, jump to **rStart** and send the media stream up to **rStop**, a new exceptional mode is defined to accomodate a transient condition. Upon receiving **play(rStart, rStop)** while in **T**, **TP** or **PST** mode, the state machine transitions to **PreSTP** and sets the start and stop NPTs: as soon as the Server cease sending the stream, the state machine transitions to **STP**. This transition is expected to occur almost immediately.

If the state machine receives a **pause(rStop)** before leaving the **PreSTP** mode, it updates the stop NPT and remains in the **PreSTP** mode. If the state machine receives a **resume(rStart)** before leaving the **PreSTP** mode, it updates the start NPT, cancels the stop NPT and transitions to **PreST**. Then, as soon as the Server cease sending the stream, the state machine transitions to **ST**. This transition is expected to occour almost immediately.

If the state machine receives a **resume(rStart)** before leaving the **PreST** mode, it updates the start NPT and remains in the **PreST** mode. If the state machine receives a **pause(rStop)** before leaving the **PreST** mode, it sets the stop NPT and transitions to **PreSTP** mode.

The **PreST** mode is also entered when a **resume(rStart)** is received while in **T** mode, as the expected behavior of the Server is to cease sending the stream at the earliest possible time, and then resume from **rStart**.

5.5.1.3.5 DSM Stream pause

DSM Stream pause	Stop sending stream when AppNPT position is reached.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole pause_ACR = READER;
        void pause (in AppNPT rStop)
                raises (MPEG_DELIVERY, BAD_STOP);
    };
};
```

Semantics

A Client calls **Stream pause()** to cause the video Server to stop sending the stream when it reaches **AppNPT rStop**.

If a **Stream pause()** is invoked while the state machine is in **T** mode, the state machine will immediately transition to **TP** mode.

If a **Stream pause()** is invoked while the state machine is in **TP** mode, the state machine will remain in **TP** mode, but the **rStop** value will be updated.

If a **Stream pause()** is invoked while the state machine is in **STP** or **ST** mode, the state machine will immediately transition to **STP** mode.

If a **Stream pause()** is invoked while the state machine is in **PST** mode (i.e., a resume is queued at **rStart**), the state machine will transition to **TP** mode, indicating MPEG Stream delivery will cease when **rStop** is reached, and no resume will occur.

The actual presentation of video frames (freeze frame versus blanked or alternative display) is considered implementation-specific and is therefore not specified.

If the stream is in forward transport mode, either an **rStop** of negative infinity or an **rStop** less than the current **AppNPT** will indicate pause immediately if there are no other commands in the stream state machine queue. If the stream is in reverse transport mode, either an **rStop** of negative infinity or an **rStop** greater than the current **AppNPT** will indicate pause immediately if there are no other commands in the stream state machine queue. An **rStop** which exceeds the Stream duration will result in a **BAD_STOP** exception.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStop	input	AppNPT position at which the pause will occur.

5.5.1.3.6 DSM Stream resume

DSM Stream resume	Start sending stream at AppNPT position.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole resume_ACR = READER;
        void resume (in AppNPT rStart, in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_SCALE);
    };
};
```

Semantics

A Client calls **Stream resume()** to cause the video Server to resume sending the stream at **rStart** at a rate and direction as specified by **rScale**.

If a **Stream resume()** is invoked while the state machine is in **ST**, **P**, or **STP** mode, the state machine will immediately transition to **ST** mode. The Server will then commence sending the stream from the position **rStart** (with scale **rScale**) at the earliest possible time. When the Server begins to send the MPEG stream the state machine will transition to **T** mode.

If a **Stream resume()** is invoked while the state machine is in **T** mode, the state machine shall immediately transition to **PreST** mode. At the earliest possible time the Server will stop sending the stream and transition to **ST**; then, at the earliest possible time, the Server will commence sending the stream from the position **rStart** (with scale **rScale**) and transition to **T** mode.

If a **Stream resume()** is invoked while the state machine is in **PST** mode, the state machine will update **rStart** and its associated **rScale**, and remain in **PST** mode. If a **Stream resume()** is invoked while the state machine is in **TP** mode, the state machine will transition to **PST** mode and set **rStart** and its associated **rScale**. Both of these indicate that when the state machine will reach **rStop**, the state machine will transition to **ST** and then **T** mode as described above, and MPEG stream delivery will commence from the new **rStart** (with scale **rScale**).

rScale is composed of a numerator and a denominator. An **rScale** of 1/1 indicates normal play at the normal forward viewing rate. It is recommended for efficiency that either the numerator or denominator have a value of 1. The ratio of numerator to denominator corresponds to the rate with respect to normal viewing rate. For example, a ratio of 2/1 indicates 2 times the normal viewing rate, and a ratio of 1/2 indicates one-half the normal viewing rate. The Server will respond with best effort, that is at the closest rate to the requested rate that it can deliver. The **rScale** reply will indicate the actual rate delivered. A positive numerator indicates forward direction. A negative numerator indicates reverse direction, with the exception of negative infinity which indicates resume now. An **rStart** which equals or exceeds the Stream duration will result in a **BAD_START** exception.

An **rScale** of 0/0 is indeterminate and will result in a BAD_SCALE exception. An **rScale** of 0/n (where n is not 0) indicates that the NPT is not changing.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT position at which to resume.
Scale rScale	input	The scale at which to resume. A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.1.3.7 DSM Stream status

DSM Stream status	Obtain status of a stream.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole status_ACR = READER;
        void  status (in Stat rAppStatus, out Stat rActStatus)
                raises (MPEG_DELIVERY);
    };
};
```

Semantics

Stream status() is used to request status of a stream in progress. It returns the current **AppNPT** position, scale and mode of the stream. The application's estimation of current position may be specified in the call request. The reply will contain the actual position.

Privileges Required
READER

Parameters

type/variable	direction	description
Stat rAppStatus	input	Expected stream status, AppNPT, Scale and Mode
Stat rActStatus	output	Actual stream status, AppNPT, Scale and Mode.

5.5.1.3.8 DSM Stream reset

DSM Stream reset	Reset the stream state machine. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole reset_ACR = READER;
        void reset ();
    };
};
```

Semantics

Stream reset() is used to reset the Stream state machine to the **O** state (Pause, rStart = 0 0, rstop = 0xEFFF 0xFFFF).

Privileges Required
READER

5.5.1.3.9 DSM Stream jump

DSM Stream jump	When stream reaches stop AppNPT, resume at start AppNPT.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole jump_ACR = READER;
        void   jump (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Semantics

Stream jump(rStart, rStop) behaves exactly as if **Stream pause(rStop)** then **Stream resume(rStart)** have been called in quick succession. However **Stream jump()** is a single method, thus if an incorrect **rStart** or **rScale**

parameter is given, the **rStop** is ignored too and the state machine is not affected at all. There are no other exceptions to the equivalence between **Stream jump(rStart, rStop)** and the sequence **Stream pause(rStop)** plus **Stream resume(rStart)**.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT position to resume from as a result of the jump.
AppNPT rStop	input	AppNPT position at which the jump will occur.
Scale rScale	input	A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.1.3.10 DSM Stream play

DSM Stream play	Play stream from start AppNPT until stop AppNPT.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole play_ACR = READER;
        void  play (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Semantics

Stream play(rStart, rStop) behaves almost exactly as if **Stream resume(rStart)** then **Stream pause(rStop)** have been called in quick succession. However **Stream play()** is a single method, thus if an incorrect **rStop** parameter is given, the **rStart** and the **rScale** are ignored too and the state machine is not affected at all. In addition, if **Stream play(rStart, rStop)** is invoked while the state machine is in **TP** or **PST** mode, it behaves differently from the sequence **Stream resume(rStart)** plus **Stream pause(rStop)**, as described already.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT position at which to resume play.
AppNPT rStop	input	AppNPT position at which to stop play and change mode to Pause.
Scale rScale	input	A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.1.4 File

This section describes the interface for two operations that read and write files. When combined with the other Core DSM interfaces such as Directory, Base, and Access, a minimal file system interface is realized. When combined with Extended DSM interfaces such as Lifecycle and View, a more complete file system interface can be realized. The IDL permits the Server Object Implementation to map to any of a variety of heterogeneous object and file systems.

5.5.1.4.1 File Definitions, Exceptions

```
module DSM {
    interface File : Base, Access {
        const AccessRole Content_get_ACR = READER;
        const AccessRole Content_put_ACR = WRITER;
        attribute opaque Content;                  // file content
#ifdef DSM_GENERAL
        const AccessRole ContentSize_get_ACR = READER;
        readonly attribute u_longlong ContentSize;  // file content size in octets
#endif
    };
};
```

The ContentSize is implicitly updated at any time that the Contents of the File are updated. Contents of the File can be updated with File write(), File _set_Content() to the File object, or **Directory_put** (of the Content attribute) to the parent Directory object. A **Directory_get** (of the Content attribute) at the parent Directory object is equivalent to opening and reading an entire file. **Directory_get()** and **Directory_put()** are described later in this Chapter.

5.5.1.4.2 Summary of File Primitives

Inherited from **Base**:

close, destroy

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Defined in **File**:

DSM File read	Random access read from a file. (READER)
DSM File write	Random access write to a file. (WRITER)

5.5.1.4.3 DSM File read

DSM File read	Random access read from a file. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole read_ACR = READER;
        void read (
                in u_longlong aOffset,
                in u_long aSize,
                in boolean aReliable,
                out opaque rData)
                raises (INV_OFFSET, INV_SIZE, READ_LOCKED);
    };
};
```

Semantics

File read() provides random access to opened files, using a File reference obtained from a previous resolve operation. **aOffset** is the absolute offset into the File Content attribute (from 0). Because **aOffset** and **aSize** are explicit parameters, seeks can be accomplished assuming the application maintains the current byte position in the file.

aSize specifies the amount of buffer the Client has allocated for the read. It can exceed the File (**ContentSize - aOffset**), in order to specify a read to End-of-File. **INV_OFFSET** shall be returned if **aOffset** is greater than File (**ContentSize -**1). **INV_SIZE** shall be returned for **aSize** that exceeds the capacity of Server, Network Resources or Client as derived from Compatibility Information.

In the case where the network imposes a long round-trip latency, efficient operation of multimedia object access requires that the underlying RPC and network protocol stack support overlapped, synchronous deferred transactions. The application will need to pre-fetch files in an attempt to stay ahead of the anticipated user actions. The RPC must assure that the Server executes the operations for a Client in the same order that the Client has invoked them.

The underlying RPC stack will retry to re-fetch lost or erroneous data if **aReliable** is TRUE. If **aReliable** is FALSE, the operation will not be retried in the event of time-out or error. This is useful in the case where media, e.g. short audio or image, is presented in fast-paced normal play application time, in which case it is more important for the presentation to move forward on schedule than to stall while an object is being re-fetched.

Privileges Required
READER

Parameters

type/variable	direction	description
u_longlong aOffset	input	64 bit value indicating starting byte position within the file.
u_long aSize	input	Number of bytes to read.
boolean aReliable	input	If aReliable = FALSE, Client indicates that the RPC reply need not be reliable, e.g., for use with multimedia data for transient presentations.
opaque rData	output	Pointer to data returned by the File read.

5.5.1.4.4 DSM File write

DSM File write	Random access write to a file. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole write_ACR = WRITER;
        void  write (
                in u_longlong aOffset,
                in u_long aSize,
                in opaque rData)
                raises (INV_OFFSET, INV_SIZE, WRITE_LOCKED);
    };
};
```

Semantics

File write() provides a mechanism to write data to a file starting at a designated offset. **aOffset** is the absolute offset into the File Content attribute (from 0). WRITER privileges are required. The **File write()** uses the File reference obtained from a previous resolve operation. Appends may be performed by using the size of the file as **aOffset**. **Size** is an exported attribute of the Access Interface and may be obtained through a **Directory get()** operation. Appends may be performed by using the **ContentSize** attribute of the File as **aOffset**. **INV_OFFSET** shall be returned if **aOffset** is greater than **ContentSize**. **INV_SIZE** shall be returned for **aSize** that exceeds the capacity of Server, Network Resources or Client as derived from Compatibility Information.

Privileges Required
WRITER

Parameters

type/variable	direction	description
u_longlong aOffset	input	64 bit value indicating starting byte position within the file.
u_long aSize	input	Number of bytes to write.
opaque rData	input	Pointer to data to be written.

5.5.1.5 Directory

The Directory interface provides a general name space for binding names to Services or data. A ServiceGateway implements the directory interface, and as such provides the primary mechanism for accessing other Services or applications.

Directory defines four kinds of operations:

- Binding a name to an object reference or data value
- Resolving a name to the bound object reference or data value
- Removing a name's binding
- Listing the bindings

The Directory interface inherits the attributes defined by the Access interfaces to allow permission definitions for directories as well as individual Services and data.

For the basic operations involving object references, Directory inherits from the NamingContext interface defined in the CORBA Object Services Naming module (CosNaming). Using the OMG NamingContext interface allows a CORBA environment to readily support DSM-CC, while not requiring a DSM-CC implementation to use a CORBA system.

Directory does not inherit from Base because both Base and NamingContext define destroy operations. The current IDL specification requires that operation names, including inherited operations, be unique and case-insensitive. Thus, Base::destroy would collide with the NamingContext::destroy. Because it cannot inherit from Base, Directory defines its own **Directory close()** operation.

For completeness, the NamingContext operations are presented below; however, the CORBA Naming specification is the reference definition of operations that a directory must support. In some cases, DSM-CC defines equivalent syntax that supports DSM-CC extensions that are compatible with CosNaming. For example, the macro DSM_GENERAL is used to enable compilation of the Core Consumer Client.

The CORBA Naming specification includes the definitions for names and bindings below. The DSM module defines equivalent types either by referencing the CosNaming definitions with a typedef or by having a complete definition of the type. A name component consists of two strings that must be unique within a specific context. A name is a sequence of components that can describe a path through a set of contexts.

When an object is bound to a Directory using **Directory bind()** or **Directory bind_context()**, an objects **id**, **kind** and **BindingType** become known to the Directory. The **BindingType** is either a object or a name context, depending upon which of the two functions was called. In DSM-CC, the **id** is expected to be the simple string name of the object, and the **kind** is the interface type of the object, in the form "<module>::<interface>" or specified interface alias. The application can see the **id**, **kind** and **BindingType** as a result of the **Directory list()** operation.

A **Directory resolve()** or **Directory open(..DEPTH..)** with multiple **NameComponents** results in a cascade of single resolves, when following the path depth-wise. Each node in the path except the last shall be a Directory. Each node in the path (except the last) will propagate the resolve to the rest of the path.

5.5.1.5.1 Directory Definitions, Exceptions

Interface names are constructed from NameComponents. The NameComponent has two parameters, an id and a kind. The id is an arbitrary string selected by the application. In the DSM-CC environment, the kind is required to be a string constructed of module name and interface, in the form “<module>::<interface>”, or a DSM-CC interface alias. For example, DSM-CC Directory uses “DSM::Directory” or the alias “dir” for the NameComponent kind value.

The CORBA Naming Service is used as a basis for directory functions. CosNaming definitions are shown below:

```
module CosNaming{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // note: BindingType equates to the CORBA enum definition
    // while allowing extension for DSM-CC implementations
    typedef unsigned long BindingType;
    const BindingType nobject = 0;
    const BindingType ncontext = 1;
    struct Binding {
        Name binding_name;
        BindingType binding_type;
    };
    typedef sequence <Binding> BindingList;
};
```

NamingContext defines operations for naming object references but not general data, so Directory adds similar operations for binding names to data values (type “any” in IDL). The structure for these values is determined by their type. In addition to naming data values, the directory interface extends the NamingContext interface with operations to bind or resolve a list of names in a single call. These operations allow a compact call to access a number of objects. The semantics of these operations are always identical to performing a sequence of the individual calls. To bind or resolve a single name, the sequence length can be set to 1.

The list of NameComponents specified in a single **Directory open()** is specified by a **aPathType** and **rPathSpec**.

The **aPathType** indicates the format of the spec, which may be a linear path of objects (DEPTH traversal) or child objects at a given level of hierarchy (BREADTH traversal).

The **rPathSpec** is a sequence of **Step** structures, each of which contains a name component and a process flag. If the **process** flag in a **Step** structure is TRUE, the **Step** name is to be resolved, otherwise the name is simply used to traverse further along the path. If the operation using **rPathSpec** returns object references, these will be contained in a separate resolved references output parameter. If the operation using **rPathSpec** returns data values, these will be contained in a separate resolved values output parameter. The definition of these types is as follows:

```
module DSM {
    // two types of path traversal, depth and breadth match traditional methods
    typedef char PathType;          // DEPTH or BREADTH
    const PathType DEPTH = 'D';
    const PathType BREADTH = 'B';
    struct Step {
        CosNaming::NameComponent name;
        boolean process;
    };
    typedef sequence<Step> PathSpec;
    typedef sequence<any> PathValues;
};
```

Directory operations return several types of exceptions, specified below. These exceptions are defined by the CosNaming module, and therefore must be either available as part of the DSM-CC environment or defined explicitly by a DSM-CC implementation.

```
module CosNaming {
    interface NamingContext {
        enum NotFoundReason { missing_node, not_context, not_object };

        exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
        };
        exception CannotProceed {
            NamingContext cxt;
            Name rest_of_name;
        };
        exception InvalidName {};
        exception AlreadyBound {};
        exception NotEmpty {};
    };
};
```

The NotFound and CannotProceed exceptions return the unresolved part of the requested name. For example, if a **bind** operation on the path (A,B,C) raises NotFound with the rest_of_name as (B,C) then the context named by A could not resolve the B name component. The meaning of the InvalidName, AlreadyBound, and NotEmpty exceptions is straightforward. The CannotProceed exception means that the resolving context did not have permission to do a resolve. In the example above, a CannotProceed exception means that the A context does not have permission to perform a resolve on the B context. In this case, the caller may wish to attempt to perform the resolve directly, as the caller might have permission even though the A context did not.

DSM::NOT_FOUND is equivalent to CosNaming::NamingContext NotFound. DSM::CANNOT_PROCEED is equivalent to CosNaming::NamingContext CannotProceed. DSM::INV_NAME is equivalent to CosNaming::NamingContext InvalidName. These exceptions are used in multiple DSM-CC interfaces, and are defined in the Common Definitions section of this Chapter.

5.5.1.5.2 Summary of Directory Primitives

Inherited from **Access**:

attributes: **Size, Hist, Lock, Perms**

Inherited from **NamingContext**:

DSM Directory list	Return a list of all the bindings to object references in the context. (READER)
DSM Directory resolve	Given a name, return an object reference for a Service Object Implementation instance. (READER)
DSM Directory bind	Bind an object reference to a name. (WRITER)
DSM Directory bind_context	Bind a naming context to a name. (WRITER)
DSM Directory rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)
DSM Directory rebind_context	Bind a context to a name, overwriting any previous binding. (WRITER)
DSM Directory unbind	Remove a binding for a name. (WRITER)
DSM Directory new_context	Create a new naming context. (OWNER)
DSM Directory bind_new_context	Create a new naming context and bind it to the given name. (OWNER)
DSM Directory destroy	Destroy the naming context. (OWNER)

Defined in **Directory**:

DSM Directory open	Resolve the objects associated with names in the given path (READER).
DSM Directory close	Close a reference to a Directory. (READER)
DSM Directory get	Return the attribute values bound to a PathSpec. (READER)
DSM Directory put	Bind attribute values to a PathSpec, overwriting any previous bindings. (WRITER)

Directory defines AccessRoles of the inherited NamingContext operations as follows:

```
module DSM
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole list_ACR = READER;
        const AccessRole resolve_ACR = READER;
        const AccessRole bind_ACR = WRITER;
        const AccessRole bind_context_ACR = WRITER;
        const AccessRole rebind_ACR = WRITER;
        const AccessRole rebind_context_ACR = WRITER;
        const AccessRole unbind_ACR = WRITER;
        const AccessRole new_context_ACR = OWNER;
        const AccessRole bind_new_context_ACR =OWNER;
        const AccessRole destroy_ACR = OWNER;
    };
};
```

5.5.1.5.3 DSM Directory list

DSM Directory list	Return a list of all the bindings to object references in the context. (READER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
        void list (in unsigned long how_many,
                out BindingList bl, out BindingIterator bi);
    };
};
```

Semantics

The list operation returns a list of bindings in the Directory. The count parameter indicates how many bindings to return immediately; the remaining bindings can be retrieved from the returned iterator. The iterator interface simply has two operations defined as follows:

```
module CosNaming {
    interface BindingIterator {
        boolean next_one (out Binding b);
        boolean next_n (in unsigned long how_many,
                    out BindingList bl);
        void destroy ();
    };
};
```

module DSM {
typedef CosNaming::BindingIterator BindingIterator;
};

The **BindingIterator next_one()** and **next_n()** operations return more bindings from the context, if there are any. Both operations return false if there were no additional bindings. The **BindingIterator destroy()** operation discards any Server-side storage associated with the iterator and makes the iterator no longer valid to access.

Privileges Required:

READER

Parameters

type/variable	direction	description
unsigned long count	input	The maximum number of bindings to return.
BindingList bindings	output	A sequence containing up to count bindings.
BindingIterator itr	output	An iterator for retrieving additional bindings.

5.5.1.5.4 DSM Directory resolve

DSM Directory resolve	Given a name, return an object reference for a Service Object Implementation instance. (READER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
        Object resolve (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **Directory resolve()** operation returns an object reference that is an Object Implementation instance of the name binding. If no name is bound, then the **NotFound** exception is raised.

Privileges Required:
READER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object	output	The object reference that is bound to the name.

5.5.1.5.5 DSM Directory bind

DSM Directory bind	Bind an object reference to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
     interface NamingContext {
#ifdef DSM_GENERAL
          void bind (in Name n, in Object obj)
                raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
#endif
     };
};
```

Semantics

The **Directory bind()** operation associates an object reference with a name. This operation raises the **AlreadyBound** exception if the name is bound to another object or data value in this context. The name specifies one or more name components that indicate intermediate contexts through which to search. If any of the components is not bound, then the **NotFound** exception is raised. If an intermediate context is found that refuses permission to the outer context then a **CannotProceed** exception is raised.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object obj	input	The object reference that is bound to the name.

5.5.1.5.6 DSM Directory bind_context

DSM Directory bind_context	Bind a naming context to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
     interface NamingContext {
#ifdef DSM_GENERAL
          void bind_context (in Name n, in NamingContext nc)
                raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
#endif
     };
};
```

Semantics

The **Directory bind_context()** operation associates a naming context with a name. This operation raises the AlreadyBound exception if the name is bound to another object or data value in this context.

This operation is distinct from the **Directory bind()** operation to allow the option of binding a context into a name space where it will not implicitly resolve components of a path. This approach also simplifies the resolution process, as a context knows exactly which contexts to search inside it rather than needing to narrow every bound object to see if it is a context.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext nc	input	The naming context that is bound to the name.

5.5.1.5.7 rebind

rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void rebind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory rebind()** operation associates an object reference with a name in a directory**.** Unlike the **Directory bind()** operation, this operation will replace the binding for a name if it was previously-bound.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object obj	input	The object reference that is bound to the name.

5.5.1.5.8 DSM Directory rebind_context

DSM Directory rebind_context	Bind a naming context to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void rebind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory rebind_context()** operation associates a naming context with a name. Unlike the **Directory bind_context()** operation, this operation will replace the binding for a name if it was previously-bound.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext nc	input	The naming context that is bound to the name.

5.5.1.5.9 DSM Directory unbind

DSM Directory unbind	Remove a binding for a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void unbind (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory unbind()** operation removes the binding associated with the given name from the directory.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.

5.5.1.5.10 DSM Directory new_context

DSM Directory new_context	Create a new naming context. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        NamingContext new_context();
#endif
    };
};
```

Semantics

The **Directory new_context()** operation returns a newly-created NamingContext.

Privileges Required:
OWNER
Parameters

type/variable	direction	description
NamingContext	output	The newly-created naming context.

5.5.1.5.11 DSM Directory bind_new_context

DSM Directory bind_new_context	Create a new naming context and bind it to the given name. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
            NamingContext bind_new_context(in Name n)
                        raises (AlreadyBound, NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory bind_new_context()** operation creates a new context and associates it with the given name.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext	output	The newly-created naming context.

5.5.1.5.12 DSM Directory destroy

DSM Directory destroy	Destroy the naming context. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
      interface NamingContext {
#ifdef DSM_GENERAL
            void destroy ()
                   raises (NotEmpty);
#endif
      };
};
```

Semantics

The **Directory destroy()** operation destroys the Directory object. It does not destroy the objects that were bound to it.

Privileges Required:
OWNER

5.5.1.5.13 DSM Directory open

DSM Directory open	Find the objects associated with the names in the given path (READER).

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole open_ACR = READER;
        void open(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out ObjRefs resolvedRefs)
                raises(OPEN_LIMIT, NO_AUTH , UNK_USER, SERVICE_XFR,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

Semantics

The **Directory open()** operation provides a path traversal with a resolve of object references from names at specified nodes in the path. The **aPathType** and **rPathSpec** parameters define the specific set of names and values that are resolved. The result is sequence of object references that corresponds to the **Steps** of the input **rPathSpec** that have process flag set. Therefore, note that the length of **rPathSpec** does not necessarily correspond to the length of the returned object references.

This operation looks up each path element sequentially, but not atomically (other directory operations may occur between the lookups of elements). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Directory open() can be used to open multiple objects at one time. **Directory open()** can also traverse the name graph in either breath-first or depth-first fashion. **Directory resolve()**, on the other hand, opens a single object at the end of a depth-first path.

Directory open() and **Directory resolve()** are similar, but not equivalent. **Directory resolve()** shall return a single object reference which is at the end of a sequence of **NameComponent**s. **Directory open()** can be used to obtain more than one object reference, either by depth-first path traversal or breadth-first path traversal. **Directory open()** can be used to resolve a single object, but there is more overhead in the message. **Directory open()** is useful where the Client needs to resolve a number of objects at one time (typically at a startup or transition points). The single message can improve response time, especially in systems where there is high latency on the request channel between the Client and the Service Domain. Regardless of whether an object is a file or a directory, a Client can use either **Directory resolve()** or **Directory open()** to obtain the object reference. **Directory resolve()** is equivalent to a **Directory Open()** in which **PathType** = DEPTH and the **PathSpec** has all process flags set to FALSE, except that last **Step** in the path, and the last **Step** represents a single object. See also Composite Application Portability and Service Inter-operability interfaces later in this Chapter, for a variation of depth-first traversal.

Directory open() in which **PathType** = BREADTH indicates a an explicit resolve of multiple objects at the target Directory (flat, not hierarchical).

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	Defines the traversal form of rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, each representing a node in a directory hierarchy.
ObjRefs rPathRefs	output	The object references resolved as a result of this operation.

5.5.1.5.14 DSM Directory close

DSM Directory close	Close a reference to a Directory. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole close_ACR = READER;
        void  close ();
    };
};
```

Semantics

Directory close() is used by the Client to indicate that access to the directory is no longer required. This operation is sent to the Directory to be closed. Closing a Directory is not specifically required unless the directory is bound as a Service to the ServiceGateway, in which case network resources may need to be freed as a result of the close.

If a Directory object is closed, object references that were resolved using that Directory remain valid.

Privileges Required:
READER

5.5.1.5.15 DSM Directory get

DSM Directory get	Return the attribute values bound to a PathSpec. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole get_ACR = READER;
        void get(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out PathValues rPathValues)
                (NO_AUTH, UNK_USER, SERVICE_XFR,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

Semantics

The **Directory get()** operation provides a path traversal that returns one or more attribute values of a Directory entry, without returning any object references. **aPathType** specifies DEPTH or BREADTH. If **aPathType** is DEPTH, the **Steps** of **rPathSpec** are <Directory>, <Directory>, ... , <Object>, <Attribute>, indicating a linear path traversal and single attribute access. If **aPathType** is BREADTH, the **Steps** of rPathSpec shall be <Object>, <Attribute>, ...,<Attribute>. indicating multiple attribute access.

This operation looks up each path node sequentially, but not atomically (other directory operations may occur between the lookups of nodes). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Directory get() and **Directory put()** operations are used to retrieve and set the values of an object's attributes, without resolving an object reference. The attribute values are stored at the Directory Service.

The data structures for the **rPathValues** of **Directory get()** and **Directory put()** are determined by their type. The TypeCode of the CORBA_any of **rPathValues** shall identify the structure of the attribute value.

The attribute NameComponent kind may be NULL, because rPathValues identifies the kind. The attribute NameComponent id shall be the identifier of the attribute, as defined in the IDL. For example, The id for the attribute DSM::File::Content shall be "Content".

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	Defines the traversal form rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, representing nodes in a directory hierarchy and attributes of a Directory entry.
PathValues rPathValues	output	The attribute values returned as a result of this operation.

5.5.1.5.16 DSM Directory put

DSM Directory put	Bind attribute values to a PathSpec, overwriting any previous bindings. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
     interface Directory : Access, CosNaming::NamingContext {
#ifdef DSM_GENERAL
            const AccessRole put_ACR = WRITER;
            void put(
                        in PathType aPathType,
                        in PathSpec rPathSpec,
                        in PathValues rPathValues)
                        raises(UNK_USER, NO_AUTH, SERVICE_XFR,
                                    NOT_FOUND, CANNOT_PROCEED, INV_NAME);
#endif
      };
};
```

Semantics

The **Directory put()** operation provides a path traversal that sets one or more attribute values of a Directory entry, without returning any object references. **aPathType** specifies DEPTH or BREADTH. If **aPathType** is DEPTH, the **Steps** of **rPathSpec** are <Directory>, <Directory>, ... , <Object>, <Attribute>, indicating a linear path traversal and single attribute access. If **aPathType** is BREADTH, the **Steps** of rPathSpec shall be <Object>, <Attribute>, ...,<Attribute>. indicating multiple attribute access.

This operation looks up each path node sequentially, but not atomically (other directory operations may occur between the lookups of nodes). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Directory get() and **Directory put()** operations are used to retrieve and set the values of an object's attributes, without resolving an object reference. The attribute values are stored at the Directory Service.

The data structures for the **rPathValues** of **Directory get()** and **Directory put()** are determined by their type. The TypeCode of the CORBA_any of**rPathValues** shall identify the structure of the attribute value.

The attribute NameComponent kind may be NULL, because rPathValues identifies the kind. The attribute NameComponent id shall be the identifier of the attribute, as defined in the IDL. For example, The id for the attribute DSM::File::Content shall be "Content".

Privileges Required:
WRITER

Parameters

type/field	direction	description
PathType aPathType	input	Defines the traversal form rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, representing nodes in a directory hierarchy and attributes of a Directory entry.
PathValues rPathValues	input	The attribute values to be bound as a result of this operation.

5.5.1.6 Session

The Session interface enables a Client to establish a Session with a ServiceGateway Domain of Services, using the operation **Session attach()**. While in the Session, the Client can navigate the Domain of Services, open new Services, and perform any operations the Service interfaces support. When finished, the Client invokes **Session detach()** to close, i.e., perform **Base close()** against, all object references of the Session.

5.5.1.6.1 Service Transfer

Session attach() and **Session detach()** can be used to perform Service Transfer. Most applications will have some nesting of navigation and will require a suspension of one level of nest when proceeding to another. For example, a top level navigator can be located in one Service Domain and a movie browser in another. When going from the top navigator to the movie browser, the Client can suspend the navigator and attach to the movie browser. Then, when done with the movie browser, the Client can pop back to the top navigator by resuming it at the previous state. This is very common in applications and natural for human behavior.

Two models are available for Service Transfer:

a) Basic application level Service Transfer

In this method, the parameters required in the next **Session attach()** (i.e., for the destinationServer) are sent to the Client from the Server at the application level. The Client uses this information to do one of the following:

- Release the Session with the sourceServer and establish a new session with the destinationServer
- Maintain the Service with the sourceServer and establish a new session with the destinationServer

In the method when the Session with the sourceServer is released the Client cannot resume the context at a later time.

b) Enhanced application level Service Transfer through Session detach(), **State suspend()** and **State resume()**.

In this method, the parameters required in the next DSM Session attach are sent to the Client from the Server at the application level. The Client uses the **State suspend()** and **State resume()** operations to perform one of the following:

- Release the Session with the sourceServer (using **detach()** with **aSuspend** true)
- Suspend a service with forced release of its connection resources (using **suspend()** with **aRelease** true)
- Suspend a service without forced release of its connection resources i.e., can be reassigned by the SessionGateway to another service within a time-out period (using **suspend()** with **aRelease** false)
- Maintain the Service with the sourceServer

In this method for the first three items above, the Client can resume the full context at a later time. Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

- Release the Session with the sourceServer
- Maintain minimum resources with the sourceServer
- Maintain the Service with the sourceServer

In the method for the first two options above, the Client can resume the full context at a later time. Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

5.5.1.6.2 Summary of Session Primitives

Defined in **Session**:

DSM Session attach	Attach to a ServiceGateway Domain of Services. (READER)
DSM Session detach	Detach from a ServiceGateway Domain of Services. (READER)

5.5.1.6.3 DSM Session attach

DSM Session attach	Attach to a ServiceGateway domain of Services. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Session {
        const AccessRole attach_ACR = READER;
        void  attach (
                in ServiceDomain aServiceDomain,        // optional ServerId
                in CosNaming::Name pathName,    // optional path name to resolve
                in UserContext  savedContext,        // previous application user context
                out ObjRefs resolvedRefs)           // objects resolved
                raises (OPEN_LIMIT, NO_AUTH, UNK_USER, SERVICE_XFR,
                        BAD_COMPAT_INFO, NO_RESUME,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

Semantics

The Client invokes **Session attach()** to establish a Session context with a ServiceGateway. This operation is invoked on the local Session object. The identification of ServiceGateway may be provided by either**aServiceDomain** or **pathName**, or both. **aServiceDomain** is in NSAP address format. For an interactive Session, it is the globally unique Server network address. For a Broadcast Carousel, it is the unique identifier of the Carousel. **pathName** provides the logical path to a ServiceGateway and optionally a first Service.

aServiceDomain may be given with a 0 length **pathName**, and **pathName** may be given with a 0 length **aServiceDomain** . In the former, the **pathName** shall be a non-conflicting path to the Service Domain in the system environment namespace. If both are given, **aServiceDomain** uniquely identifies the server associated with the

ServiceGateway, and **pathName** provides the logical name of the ServiceGateway followed by the path to the first Service.

A previous Session can be resumed by providing **savedContext** from that Session. If this is not available, **savedContext** shall be 0 length.

Depending on the **pathName** input parameter, **Session attach()** shall return object references for a ServiceGateway and optionally for a first Service. If the first Service is a Composite object, it shall return object references for the Composite parent and child objects.

Privileges Required:
READER

Parameters

type/variable	direction	description
ServiceDomain aServiceDomain	input	The ServerId entry point to a Service Domain, identifying a ServiceGateway.
CosNaming::Name pathName	input	The logical path to the first Service.
UserContext savedContext	input	Application state from a previously suspended application.
ObjRefs resolvedRefs	output	The object references resolved. ServiceGateway and optionally first Service objects.

5.5.1.6.4 DSM Session detach

DSM Session detach	Detach from a ServiceGateway domain of Services. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Session {
        const AccessRole detach_ACR = READER;
        void  detach (
                in boolean aSuspend,
                out UserContext savedContext)      // suspended user context
                raises (NO_SUSPEND);
    };
};
```

Semantics

A Client may invoke **Session detach()** to disconnect from a ServiceGateway and all objects of a Session. The operation is invoked on a remote ServiceGateway object reference. If **aSuspend** is TRUE, application state shall be returned as **savedContext**. If **UserContext** is not available, or if **aSuspend** is FALSE, **savedContext** shall be 0 length.The Client may later invoke **Session attach()** with this **savedContext** in order to resume the application from the point of suspension.

Privileges Required:
READER

Parameters

type/variable	direction	description
boolean aSuspend	input	If true, suspend application state and return savedContext.
UserContext savedContext	output	The suspended user context.

5.5.1.7 ServiceGateway

ServiceGateway inherits the Directory and Session interfaces. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGateway

5.5.1.7.1 Summary of ServiceGateway Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**

Inherited from **Session**:
operations: **attach, detach**

Application Portability Syntax

```
module DSM {
    interface ServiceGateway : Directory, Session {
            const AccessRole bind_ACR = MANAGER;
            const AccessRole bind_context_ACR = MANAGER;
            const AccessRole rebind_ACR = MANAGER;
            const AccessRole rebind_context_ACR = MANAGER;
            const AccessRole unbind_ACR = MANAGER;
    };
};
```

5.5.1.8 First

The First interface enables an application Client to obtain its first objects.

5.5.1.8.1 Summary of First Primitives

<u>Defined in **First**:</u>

DSM First root	Obtain the ServiceGateway object.
DSM First service	Obtain the Primary Service object.

5.5.1.8.2 DSM First root

DSM First root	Obtain the ServiceGateway object.

Application Portability (Local Library) Syntax

```
module DSM {
    interface First {
        Object  root ();
    };
};
```

Semantics

A Client may invoke **First root()** to obtain the object reference of the ServiceGateway for the current Session.

Privileges Required:
NONE

Parameters

Object	output	Current ServiceGateway object.

5.5.1.8.3 DSM First service

DSM First service	Obtain the Primary Service object.

Application Portability (Local Library) Syntax

```
module DSM {
    interface First {
        Object  service ();
    };
};
```

Semantics

A Client may invoke **First service()** to obtain the object reference of the current Session's Primary Service. Following **Session attach()**, this shall be the object reference of the Service specified in the **pathName** of the **Session attach()**.

Privileges Required:

NONE

Parameters

Object	output	Current Primary Service object.

5.5.2 Extended Interfaces

The DSM-CC Extended interfaces are optional. Each of these interfaces may be implemented at the discretion of the Service Provider.

The **Event** interface can be used to subscribe to receiving Stream Event Descriptors that are synchronized with audio and video in the MPEG stream.

Download provides a function call interface for the Download Protocol of Chapter 7.

The **Composite** interface is useful in high latency networks or for resolving multiple objects that have version interdependencies.

The **View** interface can be used for sorting/filtering Directory information or accessing a database.

With the **State** interface, an object declares that it supports functionality to suspend and resume application state.

In a CORBA Server environment, there is an Interface Repository where new interfaces, operations, and associated parameters may be discovered. In a DSM-CC system, the **Interfaces** operations are used to verify system integrity of all shared interfaces and types.

The **Security** interface may be used in secure system environments, where passwords or encrypt keys need to be exchanged.

The **Config** interface is useful if some applications require synchronous operations, while others can make asynchronous requests.

LifeCycle create() is a convenience function which assures uniqueness of Interoperable Object References.

The **Kind** interface enables Clients to dynamically determine which interfaces an object supports.

Abstract interfaces

Event

<u>attributes</u>
EventList
<u>operations:</u>
subscribe[R]
unsubscribe[R]

Composite

<u>operations:</u>
list_subs[R]
resolve_subs[R]
bind_subs[W]

State

<u>operations:</u>
suspend [R]
resume [R]

Config

<u>attributes</u>
DeferredSync
<u>operations:</u>
inquire [R]
wait [R]

Security

<u>operations:</u>
authenticate[R]

LifeCycle

<u>operations:</u>
create [O]

Kind

<u>operations:</u>
is_a [R]
has_a [R]

Instantiable interfaces

Download

<u>operations:</u>
info[R]
alloc[R]
start[R]
cancel[R]

View

<u>attributes</u>
Style
<u>operations:</u>
query [R]
read [R]
exec [W]

Interfaces

<u>operations:</u>
define[M]
check [M]
show [R]
undefine[M]

DownloadSI

<u>operations:</u>
info[R]
proceed[R]
cancel[R]
install[O]
deinstall[O]

SessionSI

<u>operations:</u>
attach [R]
detach [R]

R ::= Reader
W ::= Writer | R
B ::= Broker | W | R
O ::= Owner | B | W | R
M ::= Manager | O | B | W | R

5.5.2.1 Download

5.5.2.1.1 Download Definitions, Exceptions

```
module DSM {
    struct ModuleInfo {
        u_short moduleId;
        octet moduleVersion;
        u_long moduleSize;
        sequence<octet, 255>  moduleInfoBytes;
    };
    typedef sequence<ModuleInfo, 65535> ModuleInfoList;
    typedef sequence<u_short> ModuleList;
};
```

5.5.2.1.2 Summary of Download Primitives

Download info	Obtain information about prerequisite download modules. (READER)
Download alloc	Allocate memory buffers for a download. (READER).
Download start	Start and transfer the modules to the Client. (READER).
Download cancel	Cancel a download in progress. (READER).

5.5.2.1.3 DSM Download info

DSM Download info	Obtain information about prerequisite download modules. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole info_ACR = READER;
        void info (out  ModuleInfoList  rModulesInfo);
    };
};
```

Semantics

Download info() enables a Client to obtain information about modules that must be downloaded in order for an application to proceed. The fields of ModuleInfo are defined in Chapter 7.

Privileges Required
READER

Parameters

ModuleInfoList rModulesInfo	output	ModuleId, version, size and other information for each module of the Download image.

5.5.2.1.4 DSM Download alloc

DSM Download alloc	Allocate memory buffers for a download. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole alloc_ACR = READER;
        void  alloc (in u_short aModuleId,
                in  Object rWriteBuffer,
                out Object rReadBuffer)
                raises (BAD_MODULE_ID);
    };
};
```

Semantics

With **Download alloc()**, the application negotiates buffer management for a module to be downloaded. It can choose to allocate memory or otherwise assign the buffer location for each module, or it can defer to the lower transport layer to choose and manage the buffer location. If the convention is for the application to assign the buffer space for the module, it will place a non-null pointer in **rWriteBuffer**. In this case the **rReadBuffer** reply will be null. The data will be placed at **rWriteBuffer**. If the application defers to the lower layer transport to assign the buffer location, it places null in **rWriteBuffer** and the lower layer transport will return a non-null reference to the **rReadBuffer** buffer location. In the second case, the data will be placed in **rReadBuffer**. The configuration step is performed individually on each module, following a **Download info()** invocation, and prior to **Download start().**

Privileges Required

READER

Parameters

u_short aModuleId	input	Identifier of a module.
Object writeBuffer	input	Whether to write data (TRUE) or address (FALSE) to rReadBuffer.
Object readBuffer	output	Buffer or location of buffer.

5.5.2.1.5 DSM Download start

DSM Download start	Start and transfer the modules to the Client. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole start_ACR = READER;
        void  start (in ModuleList aModuleList)
                raises(BAD_MODULE_ID, MPEG_DELIVERY, TIMEOUT);
    };
};
```

Semantics

With **Download start()**, the Client requests that the transfer of modules begin. When it completes, all modules will be downloaded. A module sequence can be selected. From the application interface, aModuleList indicates only those modules that are to be placed in the Client's allocated memory.

Privileges Required
READER
Parameters

ModuleList aModuleList	input	A sequence of u_short listing modules by their id.

5.5.2.1.6 DSM Download cancel

DSM Download cancel	Cancel a download in progress. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole cancel_ACR = READER;
        void  cancel (in ModuleList aModuleList )
                raises (BAD_MODULE_ID, TIMEOUT) ;
    };
};
```

Semantics

The Client calls **Download cancel()** in order to cancel a download in progress. aModuleList identifies the ids of the modules to cancel. After this operation is invoked, the transport layer will not write to Client's allocated memory any modules that are on aModuleList. The transport layer will issue cancel to the Server when the download has progressed to the point where all remaining module ids to be downloaded are on aModuleList.

Privileges Required
READER

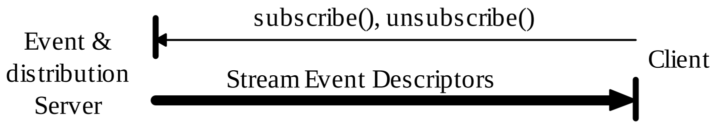
Parameters

ModuleList aModuleList	input	A sequence of u_short listing modules by their id.

5.5.2.2 Event

The event interface in the Common Object Services specification of the Object Management Group is the foundation on which the DSM-CC Event interface was built. The Event interface, however, differs in two respects. First the interface packages the functions, which scatter across multiple interfaces in the Object Management Group design, into a single interface. Second, the audience for the interface is a Client, such as the set-top device, which receives the media stream. The stream object distributes the event data through a transport mechanism that differs from that of the object invocation interface. For example, the Object Implementation of the Event interface could deliver events through the MPEG-2 stream.

The Client discovers the existence of an event by accessing the **EventList** attribute.



The Client invokes **Event subscribe()** to request that the event be sent. The Client provides an **eventName** from the EventList, which is a simple string. The stream object returns a **eventId** which uniquely identifies the event. There is only one eventId associated with a stream event name.

Note that the event includes both the **eventId** and the **rAppTime** to which the event relates. The inclusion of the time value enables the Client to schedule the reaction to the event to correlate with the presentation of a media stream.

Where the transport for event distribution is MPEG-2, the Stream Event structure shall be sent in the MPEG-2 stream encapsulated in the Stream Event descriptor, and placed in a DSM-CC_section as specified in Chapter 9, Transport.. This descriptor is defined in a later Chapter of this specification. The stream object places the event data into the media stream near the companion media data. Unsolicited Stream Event Descriptors are allowed in the MPEG-2 stream, but shall be discarded by the DSM-CC Library.

5.5.2.2.1 Event Definitions, Exceptions

```
module DSM {
    interface Event {
        // In addition to the other descriptor fields, the stream object places the
        // StreamEvent in the private data section of the media stream:
        const u_short NULL_EVENT_ID = 0;
        typedef sequence<char, 255> eventName;
        typedef  sequence<eventName, 65535> EventList_T;
        const AccessRole EventList_get_ACR = READER;
        const AccessRole EventList_put_ACR = OWNER;
        attribute EventList_T EventList;
        // the following struct is sent in the MPEG stream
        //
        struct StreamEvent {
                u_short eventId;
                AppNPT rAppTime;
                sequence<octet>  rPrivateData;};
    };
};
```

The constant declaration captures the convention that if the eventId field of the descriptor data is the value zero, it is understood that the event data which follows is invalid. If the event trigger time is maximum negative value, the semantics are to immediately respond to the event.

5.5.2.2.2 Summary of Event Primitives

Defined in **Event**:

DSM Event subscribe	Subscribe to receive an event over an MPEG stream. (READER)
DSM Event unsubscribe	Indicate desire to no longer receive an event. (READER)
DSM Event notify	Obtain Event data from a Stream Event descriptor. (READER)

5.5.2.2.3 DSM Event subscribe

DSM Event subscribe	Subscribe to receive an event over an MPEG stream. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Event {
        const AccessRole subscribe_ACR = READER;
        void subscribe(
                in string aEventName,
                out u_short eventId)
                raises(INV_EVENT_NAME);
    };
};
```

Semantics

The Client invokes **Event subscribe()** to request that the specified event be sent when it occurs. The Client provides the event name. The Service returns an eventId to associate with the event name. The scope of the eventId is at least the media stream. The Client, in other words, should not find multiple eventIds related to the same event in the stream. The exception relates to the situation where the Client provides an event name which the Service does not recognize.

Privileges Required
READER

Parameters

type/variable	direction	description
string aEventName	input	The symbolic name of the event.
u_short eventId	output	The eventId which the Service assigns, and which the Client should associate, with the event.

5.5.2.2.4 DSM Event unsubscribe

DSM Event unsubscribe	Indicate desire to no longer receive an event. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Event {
        const AccessRole unsubscribe_ACR = READER;
        void unsubscribe(
            in u_short eventId)
            raises(INV_EVENT_ID);
    };
};
```

Semantics

The Client invokes **Event unsubscribe()** to instruct the Service to not generate the event. The Client provides the eventId to describe the subscription to which the operation refers. The eventId with respect to the Client becomes stale. The Service can assign the eventId to other subscription requests. The exception relates to the situation where the Client provides a bogus eventId, for example a eventId which was valid but is now stale.

Privileges Required
READER

Parameters

type/variable	direction	description
u_short eventId	input	The eventId which identifies the previous subscription.

5.5.2.2.5 DSM Event notify

DSM Event notify	Obtain Event data from a Stream Event descriptor. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
  interface Event {
#ifdef DSM_PSEUDO
        void notify (
            out StreamEvent rStreamEvent);
#endif
    };
};
```

Semantics

Event notify() enables the Application Portability Interface applications to receive the data of the transmitted Stream Events Descriptors. This functionality is necessary for portable applications that rely on the information carried in the Stream Event Descriptors, such as the event NPT and event private data.

Note that the **Event notify()** operation is not related to a function in the Client-Server interface because it handles the reception of the Stream Event Descriptors at the Client to the Client's application.

When the Client is using the synchronous interface, it would typically set up a thread that calls **Event_notify()**. When the Event is received, **Event_notify()** would return with the Descriptor data. When the Client is using the synchronous deferred interface, it would call **Event_notify()** to obtain a DSM_RequestHandle. This RequestHandle is then used to inquire or wait for the arrival of the Stream Event Descriptor.

Privileges Required:
READER

Parameters

type/variable	direction	description
StreamEvent rStreamEvent	output	StreamEvent Structure constructed from Stream Event descriptor that was sent in an MPEG-2 stream.

5.5.2.3 Composite

The Composite interface provides the ability to associate a set of child objects related by a common parent. The child objects are marked as either required or optional. A required child object must be resolved in order for the application to start. An optional child may be resolved at any time during the running of the application. The child objects are associated as a set of compatible versions. A **Composite list_subs()** shall list all the child objects by name, version and whether they are required or optional.

Directory open() with **aPathType** = DEPTH must be used to resolve a Composite object. The Composite object shall be recognized by the BindingType **DSM::cobject** in the **Directory list()** reply. At the Application Portability Interface, **Directory open()** returns the parent Composite. A child object may be resolved with a subsequent **Directory open()** or **Directory resolve()**. If a parent Composite object is closed, its Child objects are closed as a result of the operation.

Composite bind_subs() and **Composite unbind_subs()** operations are provided for creating and destroying the Composite association.

For Composite bindings at a Directory, the BindingType shall be **DSM::cobject**.

5.5.2.3.1 Summary of Composite Primitives

Composite list_subs	For each ChildRef, list NameComponent, Version and whether required to be resolved upon opening. (READER)
Composite bind_subs	Bind sub-objects to a Composite object. (WRITER)
Composite unbind_subs	Unbind all sub-objects of a Composite object. (WRITER)

5.5.2.3.2 DSM Composite list_subs

DSM Composite list_subs	For each ChildRef, list NameComponent, Version and whether required to be resolved by resolveSubs. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    typedef u_long BindingType;    // extends CosNaming BindingType
    const BindingType cobject = 2;
    interface Composite {
        struct ChildInfo {
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
        };
        typedef sequence<ChildInfo> ChildInfos;
        //
        const AccessRole list_subs_ACR = READER;
        void list_subs (
             in CosNaming::Name name,
             out ChildInfos infos)
            raises (NOT_FOUND, INV_NAME);
    };
};
```

Semantics
A consumer Client invokes **Composite list_subs()** to list information concerning the sub-objects of a composite object. Information returned includes NameComponent, Version, and whether the sub-object is required or optional. An application developer can use Composite to combine several well-known Service types, thus making maximum use of existing U-U Library facilities.

Privileges Required
READER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.
ChildInfos rChildInfos	output	A sequence of sub-object info structures, each containing NameComponent, Version, and whether the sub-object is required or optional.

5.5.2.3.3 DSM Composite bind_subs

DSM Composite bind_subs	Bind sub-objects to a Composite object. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM
     interface Composite {
#ifdef DSM_GENERAL
          struct ChildBinding{
               CosNaming::NameComponent n;
               Version rVersion;
               boolean required;
               ObjRef obj;
          };
          typedef sequence<ChildBinding>  ChildBindings;
          const AccessRole bind_subs_ACR = WRITER;
          void bind_subs (
                in CosNaming::Name name,
                in ChildBindings rChildBindings)
               raises (NotFound, INV_NAME, AlreadyBound);
#endif
     };
};
```

Semantics

A producer Client uses **Composite bind_subs()** to bind a set of required and optional objects and their names to a composite binder. The composite set consists of compatible versions of objects.

Privileges Required
WRITER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.
ChildBindings rChildBindings	input	A sequence of sub-object references, each containing NameComponent, object reference, and whether the sub-object is required or optional.

5.5.2.3.4 DSM Composite unbind_subs

DSM Composite unbind_subs	Unbind all sub-objects of a Composite object. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM
     interface Composite {
#ifdef DSM_GENERAL
            const AccessRole unbind_subs_ACR = WRITER;
            void unbind_subs (in CosNaming::Name name)
                 raises (NotFound, INV_NAME);
#endif
     };
};
```

Semantics

A producer Client uses **Composite unbind_subs()** to unbind all sub-objects from a parent composite object.

Privileges Required
WRITER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.

5.5.2.4 View

Multimedia client-server applications using MPEG for audio, video and file access also have a need for viewing information in the perspective of the end user, as opposed to how the information is stored at the server. The View primitives provide operations for sorting and filtering data such that directories and database information can be presented to the user in a more palatable form.

Using the View interface, the relational model can be applied to objects in directories. The View Style in this case is **NON_DB**, meaning the directory is not a database. The objects' exported attributes and their associated values are used in a **View query()** query to produce a sorted and filtered result set. The result set can then be browsed using **View read()**.

Alternatively, the View interface can be applied to an actual database at the server. The View Style for this case is either **SQL89, SQL92, or SQL3**. **View query()** and **View read()** are again used by the calling application to retrieve database attributes. Note that name for SQL3 may change as that standard nears completion.

For all View Styles, the SQL language syntax is used as the basic query form.

View read() is provided which will return a number of rows. A **View read()** from the calling application can result in a **View read()** RPC call by the DSM-CC Library which will pre-fetch results in anticipation of the cursor location in the next **View read()**.

Using the application portability interface, **Directory open()** can be pipelined with **View query()** and **View read()** in deferred synchronous mode, resulting in the pipelined execution of the operations.

The overlapped execution and local results caching overcomes a potentially significant response time issue in long-latency networks.

5.5.2.4.1 Non-Database View

The View interface can be used as an extension of the Directory Interface to enable searching, sorting and filtering of Directory objects, using a minimal SQL set. A NON_DB View Style indicates that the View is not a Database, but does support limited SQL queries against a container of objects, e.g., a Directory.

The result set from the view contains temporary attribute objects which are browse-able by the client. For example, a client can sort objects by the **Access Size** attribute, using view.

The SQL set supported by a NON_DB View Style is as follows:

SELECT, as defined in SQL92, with keywords (in order normally found):

ALL	is the default and specifies that all objects that satisfy the SELECT statement should be returned
FROM	indicates which object types to perform the query against
WHERE	specifies conditions
ASC	sort in ascending order
DESC	sort in descending order
GROUP BY	return summary information about groups of objects
HAVING	return summary information about groups of objects
ORDER BY	the order in which rows are returned
UNION	combine the results of two select statements
INTERSECT	combine the results of two select statements
MINUS	combine the results of two select statements

Conditions, i.e., [attribute operator value] combinations, as defined in SQL92 Some attributes are stored in structures. The query will specify attributes within structures in ANSI-C syntax, i.e., <attribute structure name>.<attribute name>. Operators in the query must compare a value to a basic type, e.g., an integer or string.

The following are strictly NOT allowed for NON_DB View Style:

1. DISTINCT, since there are no duplicate objects within a name context
2. CONNECT BY, START WITH, since hierarchy is explicit through use of Directories
3. FOR UPDATE OF, since writes are not allowed through SQL on NON_DB View
4. NO WAIT, since locks may not be set with NON_DB View
5. plus any other non-SELECT statement

5.5.2.4.2 Database View

A View object may represent an actual database at the server. The View Styles for a database are SQL89, SQL92 and SQL3. Each of these refers to a SQL standard. Based on the type, the syntax and semantics of that standard are applicable.

5.5.2.4.3 View Procedures

The following steps outline the query sequence:

1. The client application makes the **Directory open()** of a View object, followed immediately by a**View query()** with a SQL statement.

2. The DSM-CC Library issues **Directory open()**, **View query()** and **View read()** RPCs in synchronous deferred mode, allowing them to be pipelined.

3. The RPC Server establishes the query, executes it, and creates a results area for all rows matched. The RPC Server fetches the initial set of result rows. In addition it marks which rows are to be returned.

4. The RPC reply sent to the client with a subset of the rows matched.

5. The rows returned from the **View read()** are stored in a local buffer at the client.

The following steps outline the browsing sequence:

1. The client obtains the initial set of rows or objects from **View read()** in its local buffer, as described above.

2. The client perform can obtain a row by invoking **View read()** with cursor value that points into the matching result set at the Server. The DSM-CC Library will invoke the remote interface **View read()** as needed or to pre-fetch a window of rows in anticipation of further reads.

3. Finally the client issues **Base close()** is used to close the View and the query.

5.5.2.4.4 Definition: View Style Attribute

```
module DSM
    interface View{
        // View Style identifies the Query set supported by the View
        // NON_DB indicates service is not a Database but performs minimal searches,
        // filters and sorts using SELECT as described in the DSM-CC spec
        // SQL89 indicates the View is a SQL89-compliant database
        // SQL92 indicates the View is a SQL92-compliant database
        // SQL3 indicates the View is a SQL3-compliant database
        const char NON_DB = 'N';
        const char SQL89 = '1';
        const char SQL92 = '2';
        const char SQL3 = '3';
        const AccessRole Style_get_ACR = READER;
        readonly attribute char Style;
    };
};
```

5.5.2.4.5 View Definitions: Statement, Result

```
module DSM {
    interface View {
        typedef string SQLStatement;
        //
        typedef u_short FieldCode;
        struct FieldDescribe {
                string fieldName;           // name of the field
                FieldCode aType;            // type of the field
                opaque typeParameters;      // parameters related to the given type
        };
        typedef sequence<FieldDescribe> ResultDescribe;
        //
        // FieldCodes for standard SQL types
        const FieldCode VTC_CHAR =                  1;
        const FieldCode VTC_SMALLINT =              2;
        const FieldCode VTC_INTEGER =               3;
        const FieldCode VTC_FLOAT =                 4;
        const FieldCode VTC_SMALLFLOAT =            5;
        const FieldCode VTC_DECIMAL =               6;
        const FieldCode VTC_REAL =                  7;
        const FieldCode VTC_DOUBLEPRECISION =       8;
        const FieldCode VTC_CORBA_TYPECODE =        9;
        //
        // these structs are placed into the opaque typeParameters field of
        // the FieldDescribe struct depending on the value of the FieldCode
        struct InfoChar {
                u_short length;
                boolean nullTerminated;
        };
        struct InfoFloat {
                u_short precision;
         };
        struct InfoDecimal {
                u_short precision;
                u_short scale;
```

```
            };
            //
            // type definitions for the returned data
            typedef opaque Field;
            typedef sequence<Field> Row;
            typedef sequence<Row> Result;
      };
};
```

5.5.2.4.6 Summary of View Primitives

Defined in **View**:

DSM View query	Execute a SQL select statement, fetch an initial set of result objects. (READER)
DSM View read	Read additional result rows in the context of a View query. (READER)
DSM View execute	Execute a SQL write statement. (WRITER)

5.5.2.4.7 DSM View query

DSM View query	Execute a SQL select statement. Fetch an initial set of result objects.

Application Portability / Service Inter-operability Syntax

```
module DSM {
      interface View {
            const AccessRole query_ACR = READER;
            void query(
                        in SQLStatement aSQLStatement,
                        in u_short maxRows,
                        out ResultDescribe describe,
                        out Result aResult,
                        out View iterator)
                        raises (ILLEGAL_SYNTAX);
      };
};
```

Semantics

View query() sends the SQL statement specified by **aSQLStatement** to the View object for execution. **maxRows** defines the maximum number of rows to return. **describe** contains a description of the fields in the data returned in **aResult**.

The iterator output is a View object reference. A **View query()** can be sent to this iterator to perform a further reduction of the current result set by means of a SQL statement. It also enables views to be created and manipulated without closing or destroying the initial View.

Privileges Required
READER

Parameters

type/variable	direction	description
SQLStatement aSQLStatement	input	Standard SQL statement, subject to restrictions of the View Style.
u_short maxRows	input	Maximum number of rows to return.
ResultDescribe describe	output	Description of return fields.
Result aResult	output	Results buffer of rows and fields.
View iterator	output	New View iterator for reading additional rows and further reducing the result. These are accomplished by sending a **View read** or **query** to the iterator.

5.5.2.4.8 DSM View read

DSM View read	Obtain the attributes of a single row or object.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface View {
        const AccessRole read_ACR = READER;
        void read(
                in u_short aCursor,
                in u_short maxRows,
                out Result aResult)
                raises(NO_QUERY, INV_CURSOR);
    };
};
```

Semantics

View read() is used to read additional rows after making a query. **aCursor** identifies an object in the list of objects, either in the name context, or in the select full result.

read() can be called only for the interator object returned from a query. If **read()** is called for a View object without first making a query, **NO_QUERY** exception is raised.

Privileges Required
READER

Parameters

type/field	direction	description
u_short aCursor	input	Index into the full result of a select or the list of objects in a name context.
u_short maxRows	input	Maximum number of rows to return.
Result aResult	output	The Object's attribute values or row field values.

5.5.2.4.9 DSM View execute

DSM View execute	Execute a SQL write statement.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface View {
            const AccessRole execute_ACR = WRITER;
            void execute(
                        in SQLStatement aSQLStatement)
                        raises(ILLEGAL_SYNTAX);
    };
};
```

Semantics

View execute() sends the SQL statement specified by **aSQLStatement** to the View object for execution of SQL inserts, deletes, and updates.

Use of **View execute()** is not permitted for the **NON_DB** View Style. For NON_DB Views, Directory commands such as **Directory bind()** and **unbind()** should be used instead.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
SQLStatement aSQLStatement	input	Standard SQL statement, subject to restrictions of the View Style.

5.5.2.5 State

The State interfaces enables an application to suspend state, and later resume where it left off.

5.5.2.5.1 Summary of State Primitives

DSM State suspend	Suspend application state for a Service. (READER)
DSM State resume	Resume a Service from previous application state. (READER)

5.5.2.5.2 DSM State suspend

DSM State suspend	Suspend application state for resumption at a later time. (READER)

Service Inter-operability IDL Syntax

```
module DSM {
    interface State {
        const AccessRole suspend_ACR = READER;
        void suspend(
                in boolean  aRelease,
                out UserContext  savedContext)
                raises (NO_SUSPEND);
    };
};
```

Semantics

State suspend() is used to request that a Service Object Implementation instance preserve its Client specific state or return it to the Client. If this object is a parent Composite with open child objects, **State suspend()** cascades to the child objects (how the suspend cascades to other open objects is outside the scope of DSM-CC). By setting the **aRelease** flag to 1, the Client indicates it is not planning to resume this application in the immediate future.

Privileges Required:
READER

Parameters

type/variable	direction	description
boolean aRelease	input	A hint to the underlying transport. If FALSE, an indication the Client intends to resume soon. If TRUE, an indication the Client intends to resume much later.
UserContext savedContext	output	Opaque state information.

5.5.2.5.3 DSM State resume

DSM State resume	Resume from a ServiceGateway domain of Services. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface State{
        const AccessRole resume_ACR = READER;
        void  resume (
                in UserContext  savedContext,
                out ObjRefs  restoredRefs)
                raises (NO_RESUME);
    };
};
```

Semantics

State resume() is used to request an object instance to restore a previous state using a **savedContext** sequence of octets. The structure of **restoredRefs** is the same as that for the **resolvedRefs** in the **Session attach()**.

Privileges Required:
READER

Parameters

type/variable	direction	description
UserContext savedContext	input	savedContext from a previous **State suspend()** or **Session detach()**.
ObjRefs restoredRefs	output	object references enabling the application to continue where it left off.

5.5.2.6 Interfaces

In a Service Domain where new interfaces need to be made public, an object can define its interfaces by use of the **Interfaces define()** operation. In addition to new interfaces, it can declare whether it inherits well-known interfaces such as DSM-CC Directory, Stream and File, or whether it inherits other interfaces known to the Interfaces object. More importantly, through **Interfaces define()**, an interface can be verified as being consistent with the complete and coherent system-wide interface set.

The DSM-CC application space is constructed as a name space graph starting at the ServiceGateway. The nodes represent objects of the various types specified by DSM-CC as well as additional types that may be implementation-specific.

The Directory primitives provide browsing functions to traverse the graph. Each node has a minimum of a name and type. The node may have other browse-able information such as version and date, providing the definition of the type exports these attributes. Each object type has an exported interface, which is defined through **Interfaces define()**.

An Interface Repository is recommended to insure that the object interfaces that exist in the Server are coherent and non-conflicting. DSM-CC interfaces provide a well-known base set of interfaces. However, as applications are developed, there will be new interfaces that build on the DSM-CC primitives. As these are installed in the system, it is desirable that they be verified for correctness and that they do not conflict with the existing interfaces in the system. The repository must assure all interfaces defined with it represent a valid collection of IDL definitions. For example, all inherited interfaces exist, there are no duplicate operation names or other name collisions, all parameters have known types, etc. This validation is performed by **Interfaces check()**.

5.5.2.6.1 TCKind Constants

```
module DSM {
    typedef u_long TCKind;
     //
     // TCKinds from CORBA 2.0
     const TCKind tk_null = 0;
     const TCKind tk_void = 1;
     const TCKind tk_short = 2;      // = DSM_s_short
     const TCKind tk_long = 3;       // = DSM_s_long
     const TCKind tk_ushort = 4;     //  = DSM_u_short
     const TCKind tk_ulong = 5;      //  = DSM_u_long
     const TCKind tk_float = 6;
     const TCKind tk_double = 7;
     const TCKind tk_boolean = 8;
     const TCKind tk_char = 9;
     const TCKind tk_octet = 10;
     const TCKind tk_any = 11;
     const TCKind tk_TypeCode = 12;
     const TCKind tk_Principal = 13;
     const TCKind tk_objref = 14;
     const TCKind tk_struct = 15;
     const TCKind tk_union = 16;
     const TCKind tk_enum = 17;
     const TCKind tk_string = 18;
     const TCKind tk_sequence = 19;
     const TCKind tk_array = 20;
     const TCKind tk_alias = 21;
     const TCKind tk_except = 22;
     const TCKind tk_longlong = 23;
     const TCKind tk_ulonglong = 24;
     // 25 - 0x00FFFFFF  and 0x80000000 - 0xFFFFFFFF OMG reserved
     //
     // Component Tags registered with OMG
     // The reserved range for ISO/IEC WG11 in OMG is 0x49534F00 - 0x495351FF
     //
     // TCKinds specific to DSM-CC.
     // 0x495341F00 - 0x495341F7F ISO/IEC and OMG reserved
     // DSM-CC constructed TCKinds
     const TCKind tk_IntfCode = 1230196608; // 0x49534F80 identifies supported interfaces
     const TCKind tk_Access_Hist = 1230196609; //  0x49534F81 has Version and DateTime
     const TCKind tk_Version = 1230196610; //  0x49534F82 has major and minor
     const TCKind tk_DateTime = 1230196611; //  0x49534F83 has fields
     const TCKind tk_Access_Lock = 1230196612; //  0x49534F84 has readLock and writeLock
     const TCKind tk_Access_Perms = 1230196613; //  0x49534F85 has fields
     const TCKind tk_Stream_Info = 1230196614; //  0x49534F86 has description, duration, flags
     const TCKind tk_Event_EventList = 1230196615; //  0x49534F87 is a sequence of string
     const TCKind tk_Config_ActiveRequests = 1230196616; //  0x49534F88 is a seq of RequestHandle
     // 0x49534F89 - 0x49534FFF ISO/IEC and OMG reserved
};
```

5.5.2.6.2 Exception TCKind Constants

```
module DSM {
    // CosNaming exceptions
    const TCKind tk_NotFound = 1230196736; // 0x49535000
    const TCKind tk_CannotProceed = 1230196737; //  0x49535001
    const TCKind tk_InvalidName = 1230196738; //  0x49535002
    const TCKind tk_AlreadyBound = 1230196739; //  0x49535003
    const TCKind tk_NotEmpty = 1230196740; //  0x49535004
    //0x49535005 - 0x49535030 ISO/IEC Reserved
    //
    // DSM-CC common exceptions
    const TCKind tk_ALREADY_BOUND = 1230196784; // 0x49535030
    const TCKind tk_BAD_COMPAT_INFO = 1230196785; //  0x49535031
    const TCKind tk_BAD_MODULE_ID = 1230196786; //  0x49535032
    const TCKind tk_BAD_MODULE_INFO = 1230196787; //  0x49535033
    const TCKind tk_BAD_SCALE = 1230196788; //  0x49535034
    const TCKind tk_BAD_START = 1230196789; //  0x49535035
    const TCKind tk_BAD_STOP = 1230196790; //  0x49535036
    const TCKind tk_BLOCK_SIZE = 1230196791; //  0x49535037
    const TCKind tk_CANNOT_PROCEED = 1230196792; //  0x49535038
    const TCKind tk_ILLEGAL_SYNTAX = 1230196793; //  0x49535039
    const TCKind tk_INV_CURSOR = 1230196794; //  0x4953503A
    const TCKind tk_INV_EVENT_ID = 1230196795; //  0x4953503B
    const TCKind tk_INV_EVENT_NAME = 1230196796; //  0x4953503C
    const TCKind tk_INV_KIND = 1230196797; //  0x4953503D
    const TCKind tk_INV_NAME = 1230196798; //  0x4953503E
    const TCKind tk_INV_OFFSET = 1230196799; //  0x4953503F
    const TCKind tk_INV_SIZE = 1230196800; //  0x49535040
    const TCKind tk_MPEG_DELIVERY = 1230196801; //  0x49535041
    const TCKind tk_NO_AUTH = 1230196802; //  0x49535042
    const TCKind tk_NO_QUERY = 1230196803; //  0x49535043
    const TCKind tk_NO_REF_TYPE = 1230196804; //  0x49535044
    const TCKind tk_NO_SUSPEND = 1230196805; //  0x49535045
    const TCKind tk_NO_RESUME = 1230196806; //  0x49535046
    const TCKind tk_NOT_DEFINED = 1230196807; //  0x49535047
    const TCKind tk_NOT_FOUND = 1230196808; //  0x49535048
    const TCKind tk_OPEN_LIMIT = 1230196809; //  0x49535049
    const TCKind tk_PREV_DEFINED = 1230196810; //  0x4953504A
    const TCKind tk_READ_LOCKED = 1230196811; //  0x4953504B
    const TCKind tk_TIMEOUT = 1230196812; //  0x4953504C
    const TCKind tk_UNK_USER = 1230196813; //  0x4953504D
    const TCKind tk_WRITE_LOCKED = 1230196814; //  0x4953504E
    const TCKind tk_SERVICE_XFR = 1230196815; //  0x4953504F
    // 0x49535050 - 0x495350FF ISO/IEC reserved
};
```

5.5.2.6.3 Interfaces Definitions

```
module DSM {
    // A TypeCodeBuf holds a Corba 2.0 TypeCode
    typedef opaque TypeCodeBuf;
    typedef sequence<TypeCodeBuf>  TypeCodeList;
    //
    interface Interfaces {
        typedef opaque ReferenceData;
        typedef string InterfaceDef;
    };
};
```

5.5.2.6.4 Summary of Interfaces Primitives

DSM Interfaces show	Show an interface definition, IntfCode and TypeCodes. (READER)
DSM Interfaces define	Define an object interface to the System (MANAGER)
DSM Interfaces check	Verify the coherence of an interface with the repository. (MANAGER)
DSM Interfaces undefine	Remove an object interface definition from the System. (MANAGER)

5.5.2.6.5 DSM Interfaces show

DSM Interfaces show	Show an interface definition, IntfCode and TypeCodes. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Interfaces {
#ifdef DSM_GENERAL
        const AccessRole show_ACR = READER;
        void show (
            in string aStrKind,      // DSM-CC format, "Module::Interface Type"
            in IFKind anIFKind,   // IFKind
            out string rIDL,                 // IDL used in previous define
            out IntfCode rIntf,      // DSM-CC interface code with included IFKinds
            out TypeCodeList  rTypes)     // TypeCodes required at this level of interface
        raises (NotFound, INV_NAME);
#endif
    };
};
```

Semantics

A READER Client can call **Interfaces show()** to retrieve the IDL, IntfCode, TypeCodes of an interface or type. The semantics of the output parameters **IFKind** and **TypeCodeList** are the same as those of the **Interfaces define()** operation.

If **anIFKind** is non-zero, then it will be used to lookup the interface parameters. Otherwise **aStrKind** will be used to lookup the interface information. The Interfaces show() function enables the following sequence: A Client, after invoking **Directory list()** , discovers a new interface kind the **NameComponent kind** of one of the Bindings in the **Directory list()** reply. The Client then resolves an object that supports DSM::Interfaces for this system, and invokes **Interfaces show()** with **aStrKind** from the **NameComponent kind**. The Client then iterates the **Interfaces show()** with **anIFKind** from each of the IntfCode included **IFKinds** , to obtain all type information for the interface inheritance hierarchy. If the Client supports these types, or if it can dynamically interpret them, it can then communicate with the new interface.

IDL, IntfCodes and TypeCodes must be available for inter-operability reasons. The IDL is used to generate language mappings. IntfCodes are used by **Kind_is_a()** and other operations that desire the efficiency of u_long vs. string format. TypeCodes are used by exception handlers, **Directory_get()** , **Directory_put()**, **View_Read()** , etc., to identify the values in the **any** structure. These IntfCodes and TypeCodes are necessary at application compile time, if structures are to be pre-compiled as opposed to dynamically interpreted. A compiler may invoke **Interfaces define()** or **Interfaces show()** in the process of generating the language mapping.

aStrKind can be formatted as follows:

- It can refer to an interface, in which case it will be of the form “<Module>::<Interface>”.
- It can refer to an interface by it’s DSM-CC reserved 3-character string alias.
- It can also refer to common types at the Repository level, in which case it will be of the form “<Type>”.
- It can also refer to common types at the Module level, in which case it will be of the form “<Module::Type>”.
- It can furthermore refer to types within an interface, in which case it will be in the form “<Module::Interface Type>”.

In the above, “<Module>” refers to the module symbolic name in the IDL. “<Interface>” refers to the interface symbolic name in the IDL. “<Type>” refers to the Type definition symbolic name in the IDL.

Privileges Required:
READER

Parameters

type/variable	direction	description
string aKind	input	The DSM-CC NameComponent kind in the form “Module::Interface”, or “Module::Type”, or “Module::Interface Type”.
string rIDL	output	The IDL defined for the interface or type.
IntfCode rIntf	output	The defined IntfCode for this interface.
TypeCodeList rTypes	output	DSM-CC TypeCodes used by this interface, or TypeCodes representing common type definitions.

5.5.2.6.6 DSM Interfaces define

DSM Interfaces define	Define an object interface to the System (MANAGER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
   interface Interfaces {
#ifdef DSM_GENERAL
            const AccessRole define_ACR = MANAGER;
            void define (
                in ReferenceData id,    // unique Identifier
                in InterfaceDef  rIDL,  // IDL definition
                out IntfCode rIntf,       // DSM-CC interface code with included IFKinds
                out TypeCodeList  rTypes)       // TypeCodes required at this level of interface
                raises (PREV_DEFINED, ILLEGAL_SYNTAX, NO_REF_TYPE);
#endif
     };
};
```

Semantics

Interfaces define() is used by a MANAGER to declare object interfaces and type definitions to the system environment. The interface definition specifies the exported interface of the object, i.e., exported methods and attributes, plus data type definitions. The interface definition may 'include' other interfaces to order to enable new interfaces to extend the functionality of existing interfaces. The object type is specified in OMG IDL. This interface definition will replace any previous definition.

The interface is verified for completeness and coherence with the other interfaces that have previously been defined, i.e., success from this operation means that the set of system interfaces including this one will have a definitions for all referenced types and will be without name conflicts.

A maximum of one interface can be defined per **Interfaces_define()** operation. **IntfCode** is returned which provides an **IFKind** enumeration unique for this system environment. The **IFKind** is the input to the operation **Kind has_a()**.The **IntfCode** is the output of the operation **Kind is_a()**. **IntfCode** also contains the kind string to be used in **NameComponent kind**, the id of the Interface Repository and the **IFKind** of each of the included interfaces. A sequence of **TypeCode** is returned which identifies the component types used by this interface. If no interface is defined, i.e., component types are defined outside of interfaces, an **IFKind** of ik_null and the defined **TypeCode**s are returned.

The interface definition can specify the **AccessRole** for each method, as well as the **get AccessRole** and **put AccessRole** for each exported attribute. If these are not specified, the **AccessRole** defaults to READER.

Following the **Interfaces define()**, the object type may be used in **Lifecycle create()** to produce an object reference of a known interface type.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
ReferenceData id	input	Immutable identification information, chosen by the manager of the interface.
InterfaceDef rIDL	input	The IDL for an interface or common types definition.
IntfCode rIntf	output	The defined IntfCode for this interface.
TypeCodeList rTypes	output	DSM-CC TypeCodes used by this interface, or TypeCodes representing common type definitions.

5.5.2.6.7 DSM Interfaces check

DSM Interfaces check	Verify the coherence of an interface with the repository. (MANAGER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
      interface Interfaces {
#ifdef DSM_GENERAL
              const AccessRole check_ACR = MANAGER;
              void check (in IFKind  anIFKind)
                    raises (NO_REF_TYPE);
#endif
    };
};
```

Semantics

An MANAGER may use **Interfaces check()** to verify the repository coherency. **anIFKind** represents the interface to be checked. NO_REF_TYPE indicates a missing type definition.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
IFKind anIFKind	output	Identification of the interface to be checked in the context of the system interfaces repository.

5.5.2.6.8 DSM Interfaces undefine

DSM Interfaces undefine	Remove an object interface definition from the System. (MANAGER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Interfaces {
#ifdef DSM_GENERAL
            const AccessRole undefine_ACR = MANAGER;
            void undefine (in ReferenceData id, out IFKindList usedBy);
#endif
    };
};
```

Semantics

An MANAGER may use **Interfaces undefine()** to remove the definition of an interface and its associated type definitions from the interfaces repository. If another interface includes this interface, a system incoherence will result, indicated by non-zero **includedIn** output. When this has been fixed, an **interfaces check()** should be performed on the **includedIn** interfaces to again verify system coherency.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
ReferenceData id	input	Immutable identification information, chosen by the manager of the interface.
IFKindList includedIn	output	Identification of the interfaces that have included this interface.

5.5.2.7 Security

5.5.2.7.1 DSM Security authenticate

DSM Security authenticate	Request authentication with password or decryption key. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    struct AuthRequest {
        string aPassword;
        opaque authData;
    };
    typedef AuthRequest AuthRequest_T;
    interface Security {
        const AccessRole authenticate_ACR = READER;
        void authenticate (in AuthRequest_T authInfo);
    };
};
```

Semantics

The purpose of **Security authenticate()** is to enable the Client to provide authentication information for the purpose of obtaining access to (i.e., resolving) an object. The authenticate must be given with **a** resolve operation if the object has either a non-Null **aPassword** or **authData** with length greater than 0 in its **Perms** attribute. The input argument **authInfo** is included in the **ServiceContextList** of the immediately succeeding resolve operation. If a **Directory open()**, **Directory get(),** or **Session attach()** operation is received without a corresponding authenticate, and authentication is required as described above, a **NO_AUTH** exception will be given. **NO_AUTH** can carry an encrypt key challenge. The Client is expected to know the reason for the **NO_AUTH**, and will respond accordingly. If an encrypted data response is required, the exception will carry an **authData** challenge which must be successfully processed by the Client. The Client must then send a **Security authenticate()** with proper password or processed encrypt data followed by the repeated resolve operation (a new RPC transaction).

This standard does not specify an encryption algorithm. It does enable the following sequence: a) the Service passes **authData** encrypt challenge to the Client, b) the Client processes the received **authData** via the encryption algorithm, c) the Client returns transformed **authData** response to the Service, and d) the Service verifies the challenge/response via the encryption algorithm.

The atomic operation of the authenticate with the following operation from the Client is implicit by the fact that the input structure **AuthRequest_T** is placed in the **ServiceContextList** of the associated operation.

AuthRequest_T can be sent in the **ServiceContextList** of the **ClientSessionSetupRequest**. Also, in any UNO or DSM-CC ONC RPC request, it can be in the **ServiceContextList** of the RPC message request header, as described in Annex C. Thus, authentication can occur for access to any object and with any RPC request, in addition to **Session attach()**. In DSM-CC, an owner of an application object can require authentication by setting the **Perms** attribute of the object.

Privileges Required:
READER

Parameters

type/variable	direction	description
Password aPassword	input	A character string password for authentication to open an object.
AuthRequest_T authInfo	input	A sequence of bytes used by an encryption or other authentication mechanism.

5.5.2.8 Config

The Config interface enables an application to choose synchronous or synchronous deferred invocation behavior. In synchronous mode, the request blocks until the output parameters are valid. In synchronous deferred mode, the operation returns immediately. A RequestHandle can be tested to determine when the ouput parameters are valid.

Applications can choose to initiate synchronous deferred requests to the various Services on a per process basis. DSM-CC synchronous deferred allows the application process to pipeline its function calls in a non-blocking fashion. The DSM-CC IDL compiler can be given the option to add a RequestHandle to operations as the return value in place of void (or this can be done by hand if such a compiler is not available). The resulting C compilation will have this RequestHandle, which can be used to issue synchronous deferred requests. The following Config interface is then used by the application thread to change mode from synchronous to synchronous deferred and back.

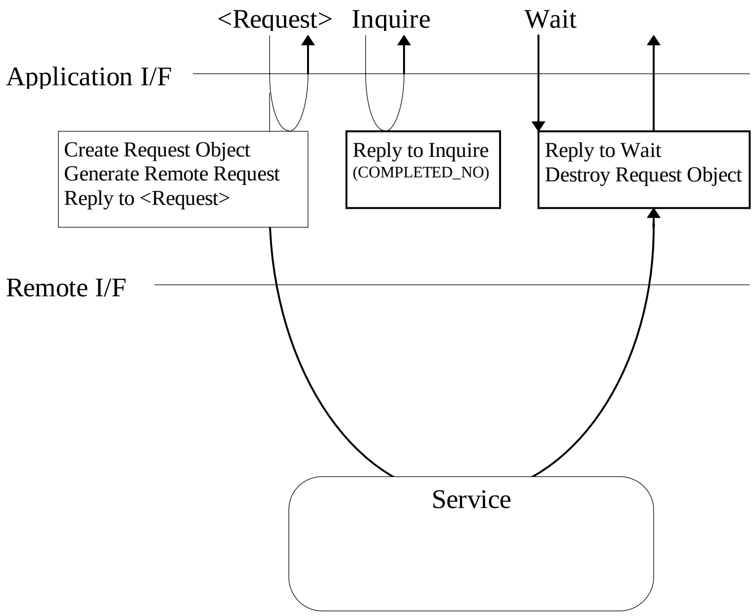


Figure 5-11 Application and Service I/O

DSM-CC Application Interface primitives can compile to be either synchronous deferred or synchronous. If the Client is multi-threaded, it can pipeline messages by sending them on separate threads. Each message does not block the next because they are called from separate threads, the calling thread will wait on the reply from the Server. Each thread can set the mode for its RPCs through the Config interface. If DeferredSync is set to FALSE, RequestHandle will always be 0 and each invocation from that thread will block until the Remote reply is received. If DeferredSync is TRUE, the invocations will not block and the RequestHandle will advance with each invocation.

The deferred synchronous mode works as follows: the Client application issues a request by calling a DSM-CC primitive, at which time the DSM-CC Library creates a request object for the transaction at hand. It then initiates the remote procedure call (RPC), and replies immediately to the calling application. The Client application may elect to continue doing something else, issue separate requests or wait on any particular outstanding request.

The Request Object is destroyed in the DSM-CC Library for any of the following reasons:

- Null exception from an Inquire invocation.
- Reply to Wait upon remote reply received.

- Destruction of higher level containment object. For example, if a remote reply is received and the DSM-CC Library determines that the parent Service is closed, the corresponding request object is destroyed.

5.5.2.8.1 Config Definitions

```
module DSM {
    // RequestHandle is 0 if Config:: DeferredSync is FALSE (synchronous RPC)
    typedef u_long RequestHandle;
    interface Config {
        // if TRUE RPC mode is deferred synchronous
        attribute boolean DeferredSync;
        typedef sequence<RequestHandle, 1024> RequestList;
        readonly attribute RequestList ActiveRequests;
    };
};
```

5.5.2.8.2 Summary of Config Primitives

DSM Config inquire	Inquire whether an operation's output parameters are valid.
DSM Config wait	Wait for an operation's output parameters to be valid.

5.5.2.8.3 DSM Config inquire

DSM Config inquire	Inquire whether an operation's output parameters are valid.

Application Portability (Local Library) Syntax

```
module DSM {
    interface Config {
        void inquire (in RequestHandle aRequest);   // inquire as to status
    };
};
```

Semantics

Config inquire() enables the Client to check the status of the transaction. If **Config inquire()** returns without an exception, the operation is completed. If it returns an exception with either COMPLETED_NO or COMPLETED_MAYBE, the operation is not complete. A successful **Config inquire()** signifies that the RPC reply data is valid. If the application has pointers to reply data as a result of the request, it may now access this data.

Privileges Required:
NONE

Parameters

type/variable	direction	description
RequestHandle aRequest	input	A handle for an outstanding request.

5.5.2.8.4 DSM Config wait

DSM Config wait	Wait for an operation's output parameters to be valid.

Application Portability (Local Library) Syntax

```
module DSM {
     interface Config {
            void wait (in RequestHandle aRequest);       // like CORBA get_response
     };
};
```

Semantics
Config wait() enables a Client to block until an operation's output parameters are valid. This is similar to the CORBA get_response operation.

Privileges Required:
NONE

Parameters

type/variable	direction	description
RequestHandle aRequest	input	A handle for an outstanding request.

5.5.2.9 LifeCycle

The **LifeCycle create()** operation is used to create a unique Interoperable Object Reference that is usable for Client-Server communications.

LifeCycle create() does not create the object itself. There are two many initialization variables for different kinds of objects to define a universal operator that actually creates the entire object for all language variations. There are many language dependencies (consider C++ constructors using inheritance vs. C). **LifeCycle Create()** does, however, generate an IOR that is unique and usable for later operations such as **Directory bind()**. The IOR remains unique and is valid across Server boots, with the exception that, if the Server address changes, the address in the IOR must be updated. It is up to the Server implementation to maintain the persistency of the associated object.

5.5.2.9.1 DSM LifeCycle create

DSM LifeCycle create	Create an object reference of a specified kind. (OWNER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface LifeCycle {
#ifdef DSM_GENERAL
            const AccessRole create_ACR = OWNER;
            void create (
                        in string aKind,
                        in Version rVersion,
                        out IOP::IOR rObjRef);
#endif
    };
};
```

Semantics

LifeCycle create() is used to create an IOR which can be associated with an instance of an object type previously defined by **Interfaces define()**. An IOR with unique object_key and valid type_id is generated. The host/port address of the caller is set in the host and port fields. The address can be modified if the object resides at a different address then the caller. The type_id is given the string format “Module::Interface” in the same format as the DSM-CC NameComponent.kind. Once created, additional tagged protocol profiles can be added at any time. The application must not alter the initial opaque data placed in the object_key, but can append to the object_key in order to extend the object identification.

LifeCycle create() is typically called as an Application Portability Interface, but is not precluded from being called as a Service Inter-operability Interface. This object reference can be used to bind the resulting IOR to a Directory, e.g. using **Directory bind()**.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
string aTypeId	input	A DSM-CC kind, as obtained by Interfaces define(), or constructed in the form “<module>::<interface>”
Version rVersion	input	Major and minor version of the new object.
ObjRef rObjRef	output	An Interoperable Object Reference with a valid type_id, version, host/port address of the caller, and unique objectkey

5.5.2.10 Kind

Kind provides local operations to determine the interfaces an object supports. If an object includes other interfaces, the **Kind has_a()** operation can be invoked on it to determine whether a specific interface is included. The **Kind is_a()** operation can be invoked to list the interfaces the object supports. An application would typically include the DSM-CC interfaces in various object types. For example, the application could define an interface type for several different formats of files, including the DSM::File interface in each one , and use **kind** to tell them apart.

5.5.2.10.1 Summary of Kind Primitives

DSM Kind has_a	Determine whether an object supports an interface. (READER)
DSM Kind is_a	Show all interfaces an object supports. (READER)

5.5.2.10.2 DSM_Kind_has_a

DSM Kind has_a	Determine whether an object supports an interface. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
     interface Kind {
#ifdef DSM_GENERAL
            const AccessRole has_a_ACR = READER;
            void has_a (
                        in IFKind anIFKind,
                        out boolean aVerdict);
#endif
     };
};
```

Semantics

Kind has_a() enables a Client to test whether an object includes(inherits) a specified interface. The interface is identified by anIFKind. If aVerdict is TRUE, the object includes that interface. If aVerdict is FALSE, it does not. has_a() is an application interface resulting in a local invocation only.

Privileges Required:
READER

Parameters

type/variable	direction	description
IFKind anIFKind	input	A DSM defined object type, e.g. Directory, File, Stream, for narrowing the type of object.
boolean aVerdict	output	TRUE or FALSE. TRUE if the object exports the interface of anIFKind.

5.5.2.10.3 DSM_Kind_is_a

DSM Kind is_a	Show all interfaces an object supports. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
     interface Kind {
#ifdef DSM_GENERAL
            const AccessRole is_a_ACR = READER;
            void is_a (
                        out IntfCode whatItIs);
#endif
     };
};
```

Semantics

Kind is_a() returns the Interface Code for an object. The Interface Code contains the IFKind and string kind of the most derived interface, the repositoryId of the defining InterfaceRepository, and the sequence of inherited IFKinds. **Kind is_a()** is an application interface resulting in a local invocation only.

Privileges Required:
READER

Parameters

type/variable	direction	description
IntfCode whatItIs	output	Identifies the TCKind of object, and the TCKinds of its inherited interfaces.

5.5.3 C Language Mappings

The application portability interfaces provide a true API with a language mapping. It can be generated using an OMG or DSM-CC IDL compiler from the Client-Service IDL. If ANSI C is the language of choice, the mapping shown in this section will be used. Applications can link with this interface as a function call library which in turn will invoke the corresponding remote procedure calls.

The application portability interfaces define a library of functions calls that can be used by Client applications to invoke the DSM-CC Client-Service interface and local DSM-CC Library functionality.

Applications that use these functions will be portable between Clients that contain the corresponding DSM-CC remote access Library.

5.5.3.1 Scoped Identifiers

A global identifier is derived from the IDL global name by converting occurrences of "::" to "_" and eliminating any leading underscores.

In some cases, very long identifiers are generated that exceed the capabilities of C compilers. To eliminate excessively long identifiers, the IDL compiler may generate a #define macro to substitute a unique, short sequence of characters in the place of a long sequence of characters. In particular, the following macro is used to shorten several CosNaming identifiers:

```
#define CosNaming COS
```

5.5.3.2 C Mapping for Operations

The IDL compiler will generate the following parameters in the C mapping equivalent of the IDL. The standard CORBA compiler generates:

- the object to which the function will be sent, and
- the environment (exception) structure.

The DSM IDL Compiler generates the above 2 parameters, and with Synchronous Deferred option on,

will also generate:

- a **RequestHandle**, which is used as an index to RPC completion status. The **RequestHandle** will be generated for those operations which have a void return value in the IDL specification. It will take the place of the void return value.

For these generated C mappings, the table describing the parameters of each primitive is augmented with these entries:

DSM_<interface name> object	input	object reference to which the call is made.
CORBA_Environment ev	output	The resulting status of the operation. If the operation succeeded the ev structure shall have the major exception type of CORBA_NO_EXCEPTION, otherwise it shall have one of the major exception types CORBA_SYSTEM_EXCEPTION or CORBA_USER_EXCEPTION, and shall contain either a CORBA system exception structure, or one of the possible DSM exception structures, as defined by the **raises** statement.
DSM_RequestHandle	output	Synchronous Deferred completion status.

Please refer to the CORBA architecture specification for additional details on C mapping rules. In addition, the CORBA architecture specification has example code on how exceptions can be handled in C.

Below is a brief overview of types frequently used by DSM-CC:

5.5.3.2.1 C Mapping for Basic Data Types

The basic IDL data types used in DSM map as follows:

IDL	DSM shorthand	C
short	**s_short**	`CORBA_short (16 bit)`
unsigned short	**u_short**	`CORBA_unsigned_short (16 bit)`
long	**s_long**	`CORBA_long (32 bit)`
unsigned long	**u_long**	`CORBA_unsigned_long (32 bit)`
long long	**s_longlong**	`CORBA_longlong (64 bit)`
unsigned long long	**u_longlong**	`CORBA_unsigned_longlong (64 bit)`

The implementation is responsible for providing the typedefs for CORBA_short, CORBA_long, etc., consistent with the IDL requirements for these types.

5.5.3.2.2 Constants

Constant identifiers are #defined in the C mapping:

5.5.3.2.3 Struct Types

A struct in IDL maps directly to the equivalent C struct.

5.5.3.2.4 Sequence Types

A sequence type is converted to a struct with a maximum length, actual length and buffer pointer.

Example:

```
typedef sequence<octet, MAX_LENGTH> rSeq;
```

is converted at application level to:

```
typedef struct {
      CORBA_unsigned_long _maximum;
      CORBA_unsigned_long _length;
      CORBA_octet * _buffer;
} rSeq;
```

It is encoded on the wire by CDR as

```
struct {
      CORBA_unsigned_long _length;
      CORBA_octet * _buffer;
};
```

The CORBA data type opaque is a sequence<octet>.

5.5.3.2.5 Strings

IDL strings are mapped to 0-byte terminated character arrays; i.e., the length of the string is encoded in the character array itself through the placement of the 0-byte.

5.5.3.2.6 Any

The CORBA typedef any maps as follows in C:

```
typedef struct any {
      CORBA_TypeCode _type;
      void * _value;}
      CORBA_any;
```

The TypeCode format is defined in the UNO CORBA 2.0 Inter-operability Specification. The first field is always an unsigned long that contains the tcKind. tcKinds are enumerated in the Interfaces interface.
Complex TypeCodes can contain a parameter list or an encapsulated CDR structure following the tcKind.

5.5.3.2.7 ev

ev is the environment structure. It is defined in the specific language mappings of the CORBA IDL. For example, it is the first parameter in the reply of the C language function mapping. In DSM-CC, **ev** is somewhat opaque in that it is specified as a structure with at a least **_ex** and **_major** members, and possibly additional implementation-specific members. **_ex** contains a string identifier plus the specific exception structure related to an invocation. **_major** identifies the exception type. _major is one of CORBA_NO_EXCEPTION, CORBA_SYSTEM_EXCEPTION or CORBA_USER_EXCEPTION.

```
enum CORBA_ExceptionType {
      CORBA_NO_EXCEPTION,
      CORBA_SYSTEM_EXCEPTION,
      CORBA_USER_EXCEPTION
    };

typedef struct CORBA_Environment {
      CORBA_ExceptionType  _major;
      CORBA_Exception *  _ex;
      ... // implementation-specific members
      } CORBA_Environment;
```

The CORBA_Exception is a structure that contains a string id in the form "<module>::<interface>::<exception>" or "<module>::<exception>". Note: the CDR Data Encoding for CORBA_exception is a string id followed by the exception body (the CORBA System exception structure or a User exception structure defined by the raises statement of the operation). CORBA provides the functions CORBA_exception_id() for accessing the string id, CORBA_exception_value() for accessing the exception structure, and CORBA_exception_free() for freeing memory associated with the exception. The member names are not defined, and it is recommended that the above pseudo-functions be used to access CORBA_Exception member values.

5.5.3.2.8 Object

Object in the C mapping is declared as type void *. This definition permits a flexibility of Client implementation.

5.5.3.3 API Definitions

Because nearly all of the Client-Service interfaces have a 1-1 mapping with the Application Portability interfaces, the semantics and parameter descriptions are maintained in the Client-Service Interfaces portion of this document. Please refer to that section for the descriptions of the interfaces or their operations.

5.5.3.3.1 C Mapping for the Synchronous Interface

The synchronous C mapping of the DSM-CC Core interfaces are shown below. These functions are generated directly from a standard CORBA IDL compiler. When called by the application, the function will always block, i.e., it will not return until all output parameters are valid.

5.5.3.3.1.1 Base

```
void DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)

void DSM_Base_destroy (
      DSM_Base object,
      CORBA_Environment * ev)
```

5.5.3.3.1.2 Access

```
DSM_u_longlong DSM_Access__get_Size (
      DSM_Access object,
      CORBA_Environment * ev)

DSM_Access_Hist_T DSM_Access__get_Hist (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Hist (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Hist_T val)

DSM_Access_Lock_T DSM_Access__get_Lock (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Lock (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Lock_T val)

DSM_Access_Perms_T DSM_Access__get_Perms (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Perms (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Perms_T val)
```

5.5.3.3.1.3 Stream

```
void DSM_Stream_pause (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStop)

void DSM_Stream_resume (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStart,
      DSM_Scale * rScale)

void DSM_Stream_status (
```

```
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_Stat * rAppStatus,
      DSM_Stream_Stat * rActStatus)

void DSM_Stream_reset (
      DSM_Stream object,
      CORBA_Environment * ev)

void DSM_Stream_jump (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStart,
      DSM_AppNPT * rStop,
      DSM_Scale * rScale)

void DSM_Stream_play (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStart,
      DSM_AppNPT * rStop,
      DSM_Scale * rScale)
```

```
DSM_Stream_Info_T DSM_Stream__get_Info (
      DSM_Stream object,
      CORBA_Environment * ev)

void DSM_Stream__set_Info (
      DSM_Stream object,
      CORBA_Environment * ev)
      DSM_Stream_Info_T val)
```

5.5.3.3.1.4 File

```
void DSM_File_read (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      CORBA_boolean * aReliable,
      DSM_opaque * rData)

void DSM_File_write (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      DSM_opaque * rData)

DSM_u_longlong DSM_File__get_ContentSize (
      DSM_File object,
      CORBA_Environment * ev)

DSM_opaque DSM_File__get_Content (
      DSM_File object,
      CORBA_Environment * ev);

void DSM_File__set_Content (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_opaque * Content);
```

5.5.3.3.1.5 Directory

```
void DSM_Directory_list (
      DSM_Directory object,
      CORBA_Environment * ev,
      CORBA_unsigned_long how_many,
      CosNaming_BindingList * bl,
      CORBA_Object * bi)

CORBA_Object DSM_Directory_resolve (
      DSM_Directory object,
      CORBA_Environment * ev,
      CosNaming_Name * n)

void DSM_Directory_open (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_ObjRefs * resolvedRefs)
```

```
void DSM_Directory_close (
      DSM_Directory object,
      CORBA_Environment * ev)

void DSM_Directory_get (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)

void DSM_Directory_put (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)

CORBA_boolean CosNaming_BindingIterator_next_one (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev,
      CosNaming_Binding * b)

CORBA_boolean CosNaming_BindingIterator_next_n (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev,
      CORBA_unsigned_long how_many,
      CosNaming_BindingList * bl)

void CosNaming_BindingIterator_destroy (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev)
```

5.5.3.3.1.6 Session

```
void DSM_Session_attach (
      DSM_Session object,
      CORBA_Environment * ev,
      CORBA_sequence_octet * aServiceDomain,
      CosNaming_Name * pathName,
      DSM_UserContext * savedContext,
      DSM_ObjRefs * resolvedRefs)

void DSM_Session_detach (
      DSM_Session object,
      CORBA_Environment * ev,
      CORBA_boolean aSuspend,
      DSM_UserContext * savedContext)
```

5.5.3.3.1.7 First

```
CORBA_Object DSM_First_root (
      DSM_First object,
      CORBA_Environment * ev)

CORBA_Object DSM_First_service (
      DSM_First object,
      CORBA_Environment * ev)
```

5.5.3.3.1.8 Event

```
void DSM_Event_subscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      CORBA_string aEventName,
      DSM_u_short * eventId)

void DSM_Event_unsubscribe (
      DSM_Event object,
      CORBA_Environment * ev,
```

```
      DSM_u_short eventId)

void DSM_Event_notify (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_Event_StreamEvent * rStreamEvent)

DSM_Event_EventList_T DSM_Event__get_EventList (
      DSM_Event object,
      CORBA_Environment * ev);

void DSM_Event__set_EventList (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_Event_EventList_T * EventList);
```

5.5.3.3.1.9 Download

```
void DSM_Download_info (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleInfoList * rModulesInfo)

void DSM_Download_alloc (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_u_short aModuleId,
      CORBA_Object rWriteBuffer,
      CORBA_Object * rReadBuffer)

void DSM_Download_start (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleList * aModuleList)

void DSM_Download_cancel (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleList * aModuleList)
```

5.5.3.3.1.10 Composite

```
void DSM_Composite_bind_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name,
      DSM_Composite_ChildRefs * rChildRefs)

void DSM_Composite_unbind_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name)
```

```
void DSM_Composite_list_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name,
      DSM_Composite_ChildInfos * infos)
```

5.5.3.3.1.11 View

```
void DSM_View_query (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement aSQLStatement,
      DSM_u_short maxRows,
      DSM_View_ViewDescribe * describe,
      DSM_View_Result * aResult,
      DSM_View * iterator)

void DSM_View_read (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_short aCursor,
      DSM_u_short maxRows,
      DSM_View_Result * aResult)

void DSM_View_execute (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement aSQLStatement)

CORBA_char DSM_View__get_Style (
      DSM_View object,
      CORBA_Environment * ev)
```

5.5.3.3.1.12 State

```
void DSM_State_suspend (
      DSM_State object,
      CORBA_Environment * ev,
      CORBA_boolean aRelease,
      DSM_UserContext * savedContext)

void DSM_State_resume (
      DSM_State object,
      CORBA_Environment * ev,
      DSM_UserContext * savedContext,
      DSM_ObjRefs * restoredRefs)
```

5.5.3.3.1.13 Interfaces

```
void DSM_Interfaces_show (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      CORBA_string aStrKind,
      DSM_IFKind anIFKind,
      CORBA_string * rIDL,
      DSM_IntfCode * rIntf,
      DSM_TypeCodeList * rTypes)

 void DSM_Interfaces_define (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      DSM_Interfaces_ReferenceData * id,
      DSM_Interfaces_InterfaceDef rIDL,
      DSM_IntfCode * rIntf,
      DSM_TypeCodeList * rTypes)

 void DSM_Interfaces_check (
      DSM_Interfaces object,
      CORBA_Environment * ev,
```

```
      DSM_IFKind anIFKind)
```

```
 void DSM_Interfaces_undefine (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      DSM_Interfaces_ReferenceData * id,
      DSM_IFKindList * usedBy)
```

5.5.3.3.1.14 Security

```
void DSM_Security_authenticate (
      DSM_Security object,
      CORBA_Environment * ev,
      DSM_AuthRequest_T * authInfo)
```

5.5.3.3.1.15 LifeCycle

```
void DSM_LifeCycle_create (
      DSM_LifeCycle object,
      CORBA_Environment * ev,
      CORBA_string aKind,
      DSM_Version * rVersion,
      IOP_IOR * rObjRef)
```

5.5.3.3.1.16 Kind

```
void DSM_Kind_has_a (
      DSM_Kind object,
      CORBA_Environment * ev,
      DSM_IFKind anIFKind,
      CORBA_boolean * aVerdict)
```

```
void DSM_Kind_is_a (
      DSM_Kind object,
      CORBA_Environment * ev,
      DSM_IntfCode * whatItIs)
```

5.5.3.3.2 C Mapping for the Synchronous Deferred Interface

When a synchronous deferred C mapping is desired, either the IDL compiler or the programmer must follow these rules:

1. The IDL operations must specify a return value of void.
2. The type DSM_RequestHandle will be substituted for the void return value.

A function with the synchronous deferred C mapping can operate either synchronously or asynchronously, using the Config interface. The following are the synchronous deferred C mappings for DSM-CC interfaces:

5.5.3.3.2.1 Config

These functions are used to configure the DSM-CC Library RPC mechanism.

```
void DSM_Config_inquire (
      DSM_Config object,
      CORBA_Environment * ev,
      RequestHandle aRequest)
```

```
void DSM_Config_wait (
      DSM_Config object,
      CORBA_Environment * ev,
      RequestHandle aRequest)
```

```
CORBA_boolean DSM_Config__get_DeferredSync (
      DSM_Config object,
      CORBA_Environment * ev);
```

```
 void DSM_Config__set_DeferredSync (
      DSM_Config object,
```

```
      CORBA_Environment * ev,
      CORBA_boolean DeferredSync)
DSM_Config_RequestList DSM_Config__get_ActiveRequests (
      DSM_Config object,
      CORBA_Environment * ev)
```

5.5.3.3.2.2 How to Convert Synchronous to Synchronous Deferred

In the following example, we show the C mapping differences between Synchronous and Synchronous Deferred:

Synchronous:

```
void DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)
```

Synchronous Deferred:

```
DSM_RequestHandle DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)
```

Any Synchronous API that returns void can be converted to Synchronous Deferred by defining it to return DSM_RequestHandle instead of void, as shown above.

5.6 Service Interoperability Interfaces(SII)

This section describes the interfaces that are use by the DSM-CC Library Stubs to invoke RPCs and U-N message sequences over the network.

5.6.1 ConnBinder and Resource to Connection Association

As a result of Broadcasts, Session Establishment and RPC resolve operations, a Client shall receive connection-related information for potentially multiple communication paths to objects. For each object, channels can be setup that have varous purposes.

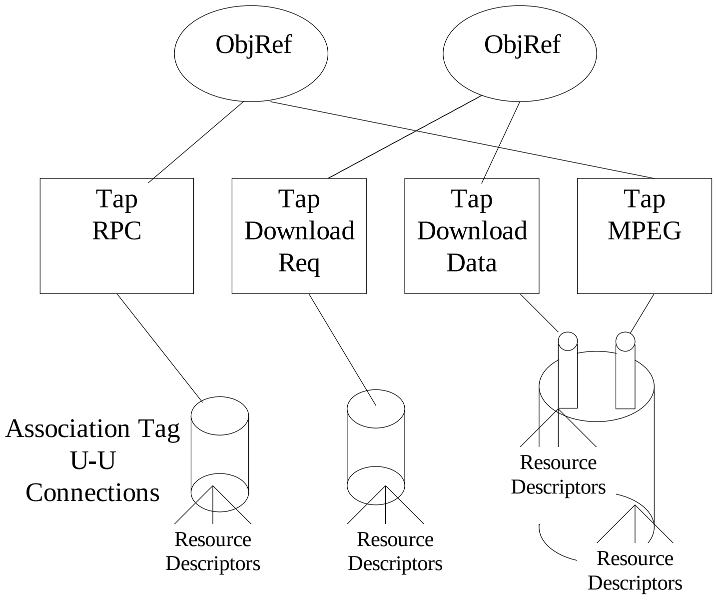
For example, communication with an Stream object (using object access Model D) may require RPC and MPEG channels. Communication with an Object Carousel Stream (using object access model B) may require elementary stream identification.

The U-U Protocol defines the **Tap**, which establishes the link from a User-to-User object reference to a lower layer communication channel. It further defines **ConnBinder** is a sequence of all **Taps** used for communication with a given object.

operations on an object reference to a channel with Network Resources.

The **Tap** has the following information:

1. An **id**. This identifies the Tap to the Client. It may be reused.
2. A **use**. This is a indication as to the type of the connection.
3. An **association tag**. This tag identifies a set of U-N Network ResourceDescriptors which have the same association tag value.
4. A **selector**. This is an opaque value which is only non-zero when the Server performs internal multiplexing of a network-level connection. If this is the case the selector is used to select a particular application-level association from those defined by the multiplexing scheme. The selector format is Tap use specific. An example of a tap selector might be a PID when the network connection is a single program transport stream.



IDL Syntax: ConnBinder and Tap

```
module DSM {
    struct Tap {
        u_short use;            // the use for the Tap
        u_short id;             // identifier
        u_short assocTag;       // the group identifier for network resource descriptors
        sequence<octet, 255> selector;  // upper protocol selection info
    };
    typedef sequence<Tap, 255> ConnBinder;   // typically have request and data channels

    typedef u_short TapUse;
    const TapUse UNKNOWN_USE = 0;
    const TapUse MPEG_TS_UP_USE = 1;                // MPEG transport upstream from Client
    const TapUse MPEG_TS_DOWN_USE = 2;                      // MPEG transport downstream to
Client
    const TapUse MPEG_ES_UP_USE = 3;                // MPEG elementary upstream from Client
    const TapUse MPEG_ES_DOWN_USE = 4;                      // MPEG elementary downstream to
Client
    const TapUse DOWNLOAD_CTRL_USE = 5;         // control request/response
    const TapUse DOWNLOAD_CTRL_UP_USE = 6;  // control request from Client
    const TapUse DOWNLOAD_CTRL_DOWN_USE = 7; // control response to Client
    const TapUse DOWNLOAD_DATA_USE = 8;         // data request/response
    const TapUse DOWNLOAD_DATA_UP_USE = 9;  // data response upstream from Client
    const TapUse DOWNLOAD_DATA_DOWN_USE = 10; // data block downstream to Client
    const TapUse STR_NPT_USE = 11;                          // NPT Descriptors
    const TapUse STR_STATUS_AND_EVENT_USE = 12; // Stream Mode and Event Descriptors
    const TapUse STR_EVENT_USE = 13;                // Stream Event Descriptor
    const TapUse STR_STATUS_USE =14;                // Stream Mode Descriptor
    const TapUse RPC_USE = 15;                      // RPC bi-directional
    const TapUse IP_USE = 16;                       // IP bi-directional
    const TapUse SDB_CTRL_USE = 17;         // control channel for Switched Digital Broadcast
    const TapUse T120_TAP1 = 18;            // reserved for use and definition by T.120
    const TapUse T120_TAP2 = 19;            // reserved for use and definition by T.120
    const TapUse T120_TAP3 = 20;            // reserved for use and definition by T.120
    const TapUse T120_TAP4 = 21;            // reserved for use and definition by T.120
    const TapUse BIOP_DELIVERY_PARA_USE = 22; // Module delivery parameters
    const TapUse BIOP_OBJECT_USE = 23;   // BIOP objects in Modules
    const TapUse BIOP_ES_USE = 24;                  // Elementary Stream
    const TapUse BIOP_PROGRAM_USE = 25;             // Program
    const TapUse BIOP_DNL_CTRL_USE = 26;            // Download control messages
};
```

The ConnBinder shall be carried in each DSM-CC IOR.

For U-N Sessions, it is not assumed that the IORs can be decoded immediatly following Session Establishment (Download might deliver the decoder). Therefore, a separate ConnBinder for Download Taps is explicitly placed in the **SessionUU attach()** output. The Server in its negotiations with the network shall determine the association tag and resource group assignments for each end-to-end connection. During the Session, the network shall inform the Client of changes to the association tags and related network resources through U-N add resource messages. Meanwhile, in the resolve reply, the Server shall return the association tag, use and selector for each Tap in the ConnBinder of the object reference.

Multiple Taps may share the same association tag, enabling one communication path to be used for more than one purpose.

5.6.1.1 Selector

A 0 length Tap selector shall indicate no selector information is present. Otherwise, the first 16 bits of the selector data shall be a type identifier to specify the remaining structure of the selector.

```
module DSM {
    typedef u_short SelectorType;
    // SelectorType 0  is ISO/IEC reserved
    const SelectorType MESSAGE = 1;
    struct MessageSelector {
        u_long transactionId;
        u_long timeout;
    };
};
```

SelectorType MESSAGE identifies a **MessageSelector.** It is used by the Object Carousel. It contains a transactionId field and a timeout field. The value of the transactionId field shall be set to the transactionId of the DownloadInfoIndication message that contains the module delivery parameters. The timeout field shall indicate the timeout period to be used to time out the acquisition of the DownloadInfoIndication message. The units of the timeout field are microseconds. Please refer to the Chapter 11, U-U Object Carousel, for further information.

5.6.2 Remote Procedure Call

The User-to-User primitives require the use of remote procedure calls to invoke operations over the network. The preferred and default RPC is Universal Networked Objects (UNO). The preferred and default data representation is Common Data Representation (CDR), also defined in the same specification. The required object reference format is the Interoperable Object Reference (IOR). These protocols and representations are defined by OMG specification “CORBA 2.0 Inter-operability: Universal Networked Objects.”. UNO does not require the use of TCP.

An alternative RPC may be chosen and negotiated by exchange of the Download InfoRequest CompatibilityDescriptor and the IOR’s ProfileId. Following this negotiation, the RPC request and reply message headers shall be those of the chosen RPC, and the data representation shall conform to the chosen encoding.

In a DSM-CC network, the RPC request header shall contain the following parameters:

- Requesting Principal (type opaque). The Requesting Principal is the identifier of the human or process that is controlling the requesting Client Application.
- Object Key(type opaque). The object key identifies a client/service connection at the server Object.
- Service Context (type IOP::ServiceContextList). The ServiceContextList carries optional client/service information.

In a DSM-CC network, the RPC reply header shall (at a minimum)contain the following parameters:

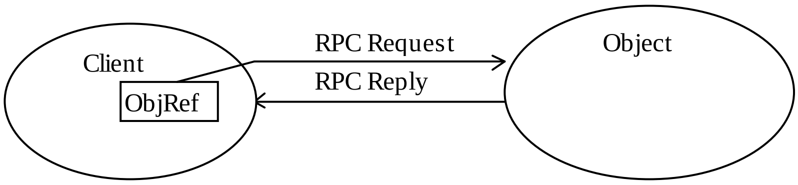
- Reply Status (type enumeration). The Reply Status, as defined by GIOP, is an enumeration:

```
module GIOP {
    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD};
};
```

- Service Context (type IOP::ServiceContextList). The ServiceContextList carries optional client/service information. It is defined later in this section.

5.6.3 The Object Reference

A Service Object Implementation instance is uniquely identified within the full DSM-CC system environment by its object reference. At the API level the object reference is a handle, with no need by the application to access its contents. However, in the RPC Stub, it is structured, and at a minimum contains the IOP Interoperable Object Reference(IOR).



Session attach(), Directory resolve(),and **Directory open()** are the resolve operations of DSM-CC User-to-User. A resolve operation takes a name input and returns a reference to an object instance. This reference is called an object reference. It contains addressing information that uniquely identifies the object, e.g., by host, port, version and object_key. The object reference is used in construction of the Network and RPC request headers that are used in routing the RPC request to the object.

DSM-CC uses the Interoperable Object Reference(IOR) format defined by OMG for object references at the Client-Service Inter-operability Interface.

```
module IOP {
    typedef unsigned long ProfileId;
    const ProfileId TAG_INTERNET_IOP = 0;
    const ProfileId TAG_MULTIPLE_COMPONENTS = 1;
    struct TaggedProfile {
        ProfileId tag;
        sequence<octet>  profile_data;
    };
    struct IOR {
        string type_id;
        sequence<TaggedProfile> taggedProfileList;
    };
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId  tag;
        sequence<octet> component_data;
    };
    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

DSM-CC has registered a range of ProfileId values with OMG. DSM-CC defines several Profile Body structures, each with an assigned ProfileId. The ProfileBody is placed in the encapsulation profile_data of the IOR. The first octet of the encapsulation is a byte_order, where a value of 0 (FALSE) indicates big-endian, and 1 (TRUE) indicates little-endian. Following that, the CORBA 2.0 encapsulation contains the encoded (according to ProfileId) connection information. The encapsulation begins at an octet according to the alignment rules of the data encoding, i.e., there may be padding for proper alignment to the first data type in the encapsulation.

The following range of ProfileIds has been registered with OMG:

Profile Tags 0x49534F00 - 0x49534F0F (the first 3 octets spell "ISO")

5.6.3.1 Min Protocol Profile

The DSM-CC Min Protocol Profile is a CORBA 2.0 single profile. It uses CDR encoding and UNO RPC. It has the minimum required ConnBinder. The ConnBinder must contain the RPC_USE Tap. The selector field of this Tap further identifies the connection, according to the network stack in use.

```
module DSM {
const IOP::ProfileId TAG_MIN = 1230196480; // 0x49534F00
    struct MinProfileBody {
        ConnBinder cbind;
    };
};
```

5.6.3.2 Child Protocol Profile

The DSM-CC Child Protocol Profile is a CORBA 2.0 single profile. It uses CDR encoding and UNO RPC. It is typically used for Composite child object references. It has the minimum required ConnBinder, plus a string for identifying the object.

```
module DSM {
const IOP::ProfileId TAG_CHILD = 1230196481; //  0x49534F01
    struct ChildProfileBody {
        ConnBinder cbind;
        string nameId;
    };
};
```

5.6.3.3 Options Protocol Profile

The DSM-CC Options Profile is a CORBA 2.0 multiple component profile. It uses CDR encoding and UNO RPC. It enables a variable list of information to be included in the IOR. The Component Tag range of 0x49534F40 - 0x49534F7F is reserved with OMG, and used by DSM-CC for IOR connection-related information. DSM-CC TaggedComponents are encoded as a ComponentId followed by length, then followed by the encoding of the type identified. DSM-CC defines the following TaggedComponents for use in this profile.

```
module DSM {
    const IOP::ProfileId TAG_OPTIONS = 1230196482; //   0x49534F02

    const IOP::ComponentId TAG_ConnBinder = 1230196544; //  0x49534F40
    //
    const IOP::ComponentId  TAG_IIOPAddr= 1230196545; //  0x49534F41
    typedef IIOP::ProfileBody IIOPAddrComponent;
    //
    const IOP::ComponentId TAG_Addr = 1230196546; //   0x49534F42
    struct AddrComponent {string host; u_short port;};
    //
    const IOP::ComponentId TAG_NameId = 1230196547; //   0x49534F43
    typedef string NameIdComponent;
    //
    const IOP::ComponentId TAG_IntfCode = 1230196548; //   0x49534F44
    //
    const IOP::ComponentId TAG_ObjectKey = 1230196549; //   0x49534F45
    typedef sequence<octet>  ObjectKeyComponent;
    //
    const IOP::ComponentId TAG_ServiceLocation = 1230196550; //   0x49534F46
};
```

TAG_ConnBinder identifies a **ConnBinder** structure. It is mandatory for all DSM-CC IORs. The ConnBinder is described in the previous section.

TAG_IIOPAddr identifies a **IIOP::ProfileBody**. It is used to hold iiop_version, host, port and object_key, as defined by the CORBA IIOP ProfileBody. iiop_version describers the version of IIOP that the agent at the specified address is prepared to communicate with. The host is the Internet host to which messages may be sent. It may be a fully quallified domain name or Internet standard “dotted decimal” form (e.g., “192.231.79.52”). The port is the TCP/IP or UDP/IP port number at the specified host where the target agent is listening for requests. The object_key is opaque value supplied by the agent producing the IOR. It uniquely identifies the object instance.

```
module IIOP {
    struct Version{
        char major; char minor;
    };
    struct ProfileBody {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;
    };
};
```

TAG_Addr identifies an **AddrComponent**. It provides IP host address and port. The semantics of host and port are the same as described above for **TAG_IIOPAddr**.

TAG_NameId identifies a **NameIdComponent**. It is used to hold the bound name identifier of the object, such as for a child of a Composite object.

TAG_IntfCode identifies an **IntfCodeComponent**. It describes inherited interfaces.

TAG_ObjectKey identifies an **ObjectKeyComponent**. It contains the unique identification of an object instance in the context of a Server Object Implementation.

TAG_ServiceLocation identifies a **ServiceLocation**. It may be used to convey **aServiceDomain**, **pathName** and **savedContext** for a subsequent **Session attach()**.

5.6.3.4 Lite Protocol Profiles

The Lite Profiles have the same IDL syntax as their counterpart DSM-CC CDR profiles. The encoding is CDR Lite. CDR Lite differs from CDR in two ways:

- It uses the maximums on sequences and strings to reduce the size of the encoded length value. An IDL-specified maximum of 255 results in an encoded octet to hold the sequence or string length. An IDL-specified maximum of 65,535 results in encoded unsigned short to hold the sequence or string length.
- It uses byte alignment in order to achieve compact packing of data.

A Multiple Component Profile called **LiteComponentProfile** is used for TAG_LITE_OPTIONS, in order to reduce the size of the profile.

```
module DSM {
    struct LiteComponent {
        IOP::ComponentId tag;
        sequence<octet, 255> component_data;
    };
    typedef sequence<LiteComponent, 255> LiteComponentProfile;

    const IOP::ProfileId TAG_LITE_MIN = 1230196483; //   0x49534F03
    const IOP::ProfileId TAG_LITE_CHILD = 1230196484; //   0x49534F04
    const IOP::ProfileId TAG_LITE_OPTIONS = 1230196485; //   0x49534F05
};
```

5.6.3.5 BIOP Protocol Profile

The BIOP profile identifies the default data encoding method used within DSM-CC U-U Object Carousels. The default data encoding is CDR Lite.

```
module DSM {
    const IOP::ProfileId TAG_BIOP = 1230196486; //   0x49534F06
};
```

```
module BIOP {
    const IOP::ComponentId  TAG_ObjectLocation =   1230196560; //   0x49534F50

    struct ObjectLocation {
        unsigned long           carouselId;
        unsigned short          moduleId;
        DSM::Version            version;
        sequence <octet, 255>   objectKey;
    };
};
```

BIOP::TAG_ObjectLocation identifies a **BIOP::ObjectLocation** component. It uniquely locates the object within the Broadcast network. Please refer to Chapter 11, U-U Object Carousel, for further information.

5.6.3.6 ONC Protocol Profile

The DSM-CC ONC Profile is a CORBA 2.0 multiple component profile. It uses XDR encoding and ONC RPC. It enables a variable list of information to be included in the IOR. The TaggedComponents of the Options Profile may be included. In addition, DSM-CC defines the following TaggedComponents for use in this profile.

```
module DSM {
    const IOP::ProfileId TAG_ONC = 1230196487; //   0x49534F07
};
```

```
module DSM_ONC {
    const IOP::ComponentId TAG_Intf  = 1230196568; //   0x49534F58

    struct IntfComponent {
        unsigned long aProgram;
        unsigned long aVersion;
    };
};
```

DSM_ONC::TAG_Intf identifies the **DSM_ONC::IntfComponent**. This structure contains ONC Program and ONC Version. Please refer to Informative Annex C, ONC RPC XDR Mappings, for further information.

5.6.4 ServiceContextList

The UNO RPC Request and Reply Message Headers carry a **ServiceContextList** parameter that is useful in carrying certain information related to the RPC but not necessarily information that is exposed to the application level. Examples of this are authentication, compatibility information, and End-User preferences. DSM-CC uses this encoding to carry similar information in the U-N Session Establishment messages **uuData** field. The ServiceContextList is defined by CORBA 2.0, shown below:

```
module IOP {
    typedef unsigned long ServiceID;
    struct ServiceContext {
        ServiceID  context_id;
        sequence<octet> context_data;
    };
    typedef sequence <ServiceContext> ServiceContextList;
    const ServiceID TransactionService =  0;
};
```

The context_data in the ServiceContext is a UNO encapsulation, meaning that it always begin with an octet that has a boolean value for byte order. FALSE (0) byte_order indicates big-endian, TRUE (1) byte_order indicates little-endian for the ensuing encapsulation.

If there is no ServiceContext, the **ServiceContextList** shall consist of a DSM::u_long with value = 0.

Note: The CORBA 2.0 ServiceId should not be confused with the DSM-CC ServerId. The ServiceId identifies some application contextual information, whereas the DSM-CC ServerId is a Server identifier as specified by the U-N Chapters of DSM-CC.

5.6.4.1.1 ServiceContext

The ServiceContext is identified by a ServiceContext identifier. Within DSM-CC, ServiceContext identifiers are expected to be allocated as ISO reserved and private, where identifiers 0-255 are ISO reserved and 256-2^{32} are for private application use.

The ServiceContext identifier shall have the following format:

MSB Octet: a disposition which indicates how the receiving object should respond to a ServiceContext indication:

0: Act on the message and the ServiceContext
1: Return an error if the context_id is unknown
2: Ignore the ServiceContext if the context_id is unknown

The 24 bits of the 3 LSB octets are the actual id with reserved and private values as described above. The following are reserved values for DSM-CC:

value	structure	description
0		ISO/IEC reserved
1	DSM::CompatibilityDescriptor	Compatibility information format (ref Chapter 6) in request or reply. Used for operation-specific compatibility exchange.
2	DSM::Download InfoRequest	For negotiation large transfer flow-control in a request. An operation may imply a possible Download. Without changing the syntax of the operation, Download request and response can be carried in the ServiceContextList.
3	DSM::Download InfoResponse	For negotiation large transfer flow-control in a reply.
4	DSM::AuthRequest_T	A **Security authenticate()** request can cause insertion of authentication data in the ServiceContextList of the following operation.
5	DSM::ConnBinder	For carrying Download Taps information in a File read() or other reply to a large transfer request.
6	DSM::Version	To identify version selection in a resolve request, for use where a version other than latest version is required.

5.6.5 Core Interfaces

5.6.5.1 Base

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Base API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Base {
        const AccessRole close_ACR = READER;
        void  close ();
#ifdef DSM_GENERAL
        const AccessRole destroy_ACR = OWNER;
        void destroy  ();
#endif
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

Base close() implies that network resources needed for the Client communication to that object can be deleted, if these resources are not used for communication between the Client and another object.

Base destroy() indicates that the object shall cease to exist for all Clients, and all resources related to it should be deleted.

5.6.5.2 Access

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Access API for data type and attribute descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    struct DateTime {                 // tm from ANSI C std.
        s_long tm_sec;                //  seconds, 0-59
        s_long tm_min;                // minutes, 0-59
        s_long tm_hour;               // hours, 0-23
        s_long tm_mday;               // day of the month, 1-31
        s_long tm_mon;                // months since Jan, 0-31
        s_long tm_year;               //years from 1900
        s_long tm_wday;               // days since Sunday, 0-6
        s_long tm_yday;               // days since Jan 31, 0-365
        s_long tm_isdst;};            // Daylight Savings Time indicator

    interface Access {
#ifdef DSM_GENERAL
        // size
        const AccessRole Size_get_ACR = READER;
        readonly attribute u_longlong Size;          // size of all attributes in octets;
#endif
        // history
        struct Hist_T {
                Version  aVersion;          // object version
                DateTime  aDateTime;};   // time created or last updated, GMT
#ifdef DSM_GENERAL
        const AccessRole Hist_get_ACR = READER;
        const AccessRole Hist_put_ACR = BROKER;
        attribute Hist_T Hist;                   // version and time of persistent object
#endif
        // lock status
        struct Lock_T {boolean readLock; boolean writeLock;};
        const AccessRole Lock_get_ACR = READER;
        const AccessRole Lock_put_ACR = WRITER;
        attribute Lock_T Lock;

        // permissions
        struct Perms_T {
                // the next 4 are binary masks of binary flags signifying
                // groups that can access the object
                u_short managerPerm;
                u_short brokerPerm;
                u_longlong writerPerm;
                u_longlong readerPerm;
                opaque owner;                       //owner identifier = Principal
                string aPassword;                   //PIN
                opaque authData;            //system-specific
                // instruct lower layers to implement a secure connection for this object
                boolean allSecure;};                // all methods parameters encrypted
#ifdef DSM_GENERAL
        const AccessRole Perms_get_ACR = OWNER;
        const AccessRole Perms_put_ACR = OWNER;
        attribute Perms_T Perms;
#endif
    };
};
```

Service Inter-operability Semantics

The End-User's Principal Id shall be sent in the RPC message header of the request. The Requesting Principal is available as an index associating the End-User's AccessRoles. The DSM-CC Library may optionally send authentication data (**AuthRequest_T** containing encryption information or password) in the ServiceContextList of the request. At the Service, authorization to invoke shall be performed on a per operation basis, according to the Access Control Role of the operation, the AccessRole of the End-User, and authentication data. It is beyond the scope of DSM-CC to describe the authentication and authorization mechanisms.

TapUse

RPC_USE

5.6.5.3 Stream

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Stream API for state machine and individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        typedef u_long Mode;
        const Mode OPEN_M = 0;
        const Mode PAUSE_M = 1;
        const Mode TRANSPORT_M = 2;
        const Mode TRANSPORT_PAUSE_M = 3;
        const Mode SEARCH_TRANSPORT_M = 4;
        const Mode SEARCH_TRANSPORT_PAUSE_M = 5;
        const Mode PAUSE_SEARCH_TRANSPORT_M = 6;
        const Mode END_OF_STREAM_M = 7;
        const Mode PRE_SEARCH_TRANSPORT_M = 8;
        const Mode PRE_SEARCH_TRANSPORT_PAUSE_M = 9;
        struct Stat {
                AppNPT rPosition;
                Scale rScale;
                Mode aMode;};
        struct Info_T {
                string<255> aDescription;
                AppNPT duration;
                boolean audio;
                boolean video;
                boolean data;};
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
        attribute Info_T  Info;
        const AccessRole pause_ACR = READER;
        void pause (in AppNPT rStop)
                raises (MPEG_DELIVERY, BAD_STOP);
        const AccessRole resume_ACR = READER;
        void resume (in AppNPT rStart, in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_SCALE);
        const AccessRole status_ACR = READER;
        void  status (in Stat rAppStatus, out Stat rActStatus)
                raises (MPEG_DELIVERY);
        const AccessRole reset_ACR = READER;
        void reset ();
        const AccessRole jump_ACR = READER;
        void   jump (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
        const AccessRole play_ACR = READER;
        void  play (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Possible TapUses

RPC_USE	operation requests and replies
MPEG_DOWN_TS_USE	MPEG-2 Transport Stream delivery
MPEG_DOWN_ES_USE	MPEG-2 Elementary Stream delivery
STR_NPT_USE	NPT Descriptors
STR_STATUS_AND_EVENT_USE	Stream Mode and Event Descriptors
STR_STATUS_USE	Stream Mode Descriptor

Service Inter-operability Semantics

The DSM-CC Library Client Stream stub may be modified to intercept MPEG-2 Stream descriptors that contain Normal Play Time and Stream state machine information. In this case, **Stream status()** may become a local library operation that returns parameters of higher accuracy.

5.6.5.3.1 Transport and Application Level NPT

There are two formats for NPT, transport level NPT and application level NPT.

The transport NPT on the MPEG transport as described in section 8 of this specification is in PTS 33 bit format. The reason is that networks may want to deal with Audio, Video and DSM-CC descriptors using the same timestamp format. The conversion from PTS format to seconds and microseconds is easy and not compute-intensive. the conversion the other way is not. The Server will format 33 bit values (in 64 bit parameters) on the MPEG downstream for all NPT descriptors.

The application NPT is raised to the human level, i.e. seconds and microseconds, to satisfy requirements from MHEG and many application-level users. This is referred to as application NPT. This can be carried on the RPC request to the Server in this form (the current IDL). The Server can convert from application NPT to transport NPT bit format if needed.

Conversion from transport NPT to application NPT is included in section 8 of this document.

5.6.5.3.2 Consistent Quantization Rules

The Stream Service shall always apply consistent quantization rules:

- Resume at current NPT (retrieved by **Stream status()**) after pause now will be no loss (will not drop nor duplicate stream data).
- Resume now after pause now will be no loss (will not drop nor duplicate stream data).
- Resume at other time than now means no guarantee of the frame relationship to a previous pause (now).

Regarding Stream Mode descriptors and NPT descriptors in the MPEG stream:

- Sending of NPT and Stream Mode descriptors is optional. However, sending of Stream Mode descriptor for End of Stream is mandatory. The Stream object must send an EOS Stream Mode Descriptor when the State Machine transitions to EOS.
- There is a general policy to include descriptors just before discontinuities.
- Use post_discontinuity indicator to define when effective (see section 8.4.1).
- Stream Mode descriptors may be sent in the MPEG transport stream upon occurrence of a stream state machine change.

5.6.5.4 File

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the File API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole Content_get_ACR = READER;
        const AccessRole Content_put_ACR = WRITER;
        attribute opaque Content;                       // file content
#ifdef DSM_GENERAL
        const AccessRole ContentSize_get_ACR = READER;
        readonly attribute u_longlong ContentSize;  // file content size in octets
#endif
        const AccessRole read_ACR = READER;
        void read (
            in u_longlong aOffset,
            in u_long aSize,
            in boolean aReliable,
            out opaque rData)
            raises (INV_OFFSET, INV_SIZE, READ_LOCKED);
        const AccessRole write_ACR = WRITER;
        void  write (
            in u_longlong aOffset,
            in u_long aSize,
            in opaque rData)
            raises (INV_OFFSET, INV_SIZE, WRITE_LOCKED);
    };
};
```

TapUse

RPC_USE

5.6.5.5 BindingIterator

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax: CosNaming Types

```
module CosNaming{
      typedef string Istring;
      struct NameComponent {
            Istring id;
            Istring kind;
      };
      typedef sequence<NameComponent> Name;
      // note: BindingType equates to the CORBA enum definition
      // while allowing extension for DSM-CC implementations
      typedef unsigned long BindingType;
      const BindingType nobject = 0;
      const BindingType ncontext = 1;
      struct Binding {
            Name binding_name;
            BindingType binding_type;
      };
      typedef sequence <Binding> BindingList;
};
```

Application Portability / Service Inter-operability Syntax: Operations

```
module CosNaming {
      interface BindingIterator {
            boolean next_one (out Binding b);
            boolean next_n (in unsigned long how_many,
                              out BindingList bl);
            void destroy ();
    };
};
```

TapUse

RPC_USE

5.6.5.6 NamingContext

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module CosNaming {
    interface NamingContext {
        enum NotFoundReason { missing_node, not_context, not_object };

        exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
        };
        exception CannotProceed {
            NamingContext cxt;
            Name rest_of_name;
        };
        exception InvalidName {};
        exception AlreadyBound {};
        exception NotEmpty {};

        void list (in unsigned long how_many,
                out BindingList bl, out BindingIterator bi);
        Object resolve (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
#ifdef DSM_GENERAL
        void bind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
        void bind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName);
        void rebind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName);
        void unbind (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context (in Name n)
            raises (NotFound, AlreadyBound, CannotProceed, InvalidName);
        void destroy ()
            raises (NotEmpty);
#endif
    };
};
```

TapUse

RPC_USE

5.6.5.7 Directory

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        // Access Control Roles for inherited NamingContext operations
        const AccessRole list_ACR = READER;
        const AccessRole resolve_ACR = READER;
        const AccessRole bind_ACR = WRITER;
        const AccessRole bind_context_ACR = WRITER;
        const AccessRole rebind_ACR = WRITER;
        const AccessRole rebind_context_ACR = WRITER;
        const AccessRole unbind_ACR = WRITER;
        const AccessRole new_context_ACR = OWNER;
        const AccessRole bind_new_context_ACR = OWNER;
        const AccessRole destroy_ACR = OWNER;
        //
        const AccessRole open_ACR = READER;
        void open(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out ObjRefs resolvedRefs)
        raises(OPEN_LIMIT, NO_AUTH , UNK_USER, SERVICE_XFR,
                NOT_FOUND, CANNOT_PROCEED, INV_NAME);
        const AccessRole close_ACR = READER;
        void  close ();
        const AccessRole get_ACR = READER;
        void get(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out PathValues rPathValues)
        raises(NO_AUTH, UNK_USER, SERVICE_XFR,
                NOT_FOUND, CANNOT_PROCEED, INV_NAME);
#ifdef DSM_GENERAL
        const AccessRole put_ACR = WRITER;
        void put(
                in PathType aPathType,
                in PathSpec rPathSpec,
                in PathValues rPathValues)
        raises(NO_AUTH, UNK_USER, SERVICE_XFR,
                NOT_FOUND, CANNOT_PROCEED, INV_NAME);
#endif
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

AccessRoles are defined for all inherited CosNaming operations. The Directory Service therefore performs authorization of Clients to perform these operations.

A **ConnBinder** must be present in the object reference (IOR) of the **Directory resolve()** and **Directory open()** RPC replies.

The API level **Directory resolve()** or **Directory open()** may result in a local operation to return an object reference that was previously obtained through **SessionUU attach()** or **Directory open()** RPC of a Composite object.

5.6.6 Extended Interfaces

5.6.6.1 SessionUU

The SessionUU interface provides definitions for uuData of the DSM-CC U-N Session establishment and teardown messages. The in parameters become the uuData in the ClientSessionSetupRequest, and the out parameters become the uuData in the ClientSessionSetupConfirm.

pseudo-IDL

```
module DSM {
    interface SessionUU {
        const AccessRole attach_ACR = READER;
        void  attach (
                in opaque downloadInfoReq,          // download info request
                in CosNaming::Name pathName,    // path name to resolve
                in UserContext savedContext,        // previous application user context
                in Principal aPrincipal,            // identification of End User
                in IOP::ServiceContextList inSC, // optional Service info
                out opaque downloadInfoResp,       // download info response
                out ConnBinder downloadTaps,      // download connection info
                out ObjRefs resolvedRefs,           // objects resolved
                out IOP::ServiceContextList  outSC) // optional Service info
                raises (NO_AUTH, BAD_COMPAT_INFO, UNK_USER,
                        SERVICE_XFR, NO_RESUME, OPEN_LIMIT,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
        const AccessRole detach_ACR = READER;
        void  detach (
            in boolean aSuspend,
            in Principal aPrincipal,                // identification of End User
            in IOP::ServiceContextList inSC,        // optional Service info
            out IOP::ServiceContextList outSC,      // optional Service info
            out UserContext savedContext)           // suspended user context
            raises (NO_SUSPEND);
    };
};
```

Service Inter-operability Semantics

Both **SessionUU attach()** and **SessionUU detach()** are used by local library Session object operations **Session attach()** and **Session detach()**, respectively.

SessionUU attach() specifies the parameters to be included in the uuData fields of the User-Network Session Establishment messages. The input parameters are placed in uuData field of the U-N ClientSessionSetupRequest, and the output parameters are retrieved from uuData field of the U-N ClientSessionSetupResponse. **downloadInfoRequest** and **DownloadInfoResponse** contain the data encodings specified in Chapter 7. An identification of a previously suspended user context **savedContext** enables the Client to indicate that an application is to resume from previously suspended state. If this is set to 0, it indicates the application is starting up for the first time. A path specification **pathName** names the path to ServiceGatewayUU and possibly a first Service to be opened. **resolvedRefs** shall contain an object reference for the ServiceGatewayUU and resolved object reference(s) for the first Service. A default Service may be specified in the second **Step** of **pathName** as a NULL string **NameComponent.id**. This shall be an indication to the ServiceGatewayUU that it can choose the first Service for the Client. Each resolved IOR of the **SessionUU attach()** output must contain a **ConnBinder**.

The ServiceGatewayUU output from SessionUU attach() is presented to the Client as a ServiceGateway object reference, i.e., the Client only sees the ServiceGateway, while the remote Server implements ServiceGatewayUU.

SessionUU detach() specifies the parameters to be included in the uuData fileds of the User-Network Session Teardown messages.

In addition to Session Teardown, all object references are closed for that Session. If **aSuspend** is true, the ServiceGatewayUU shall inform Services which are maintaining user context for this End User to return user context state for a possible resumption of those Services. It is up to the application to determine which state is to be returned and maintained between Sessions. The Client may later invoke **attach()** with this **UserContext** in order to resume from the saved state.

aPrincipal specifies a specific user (i.e., human) or requesting principal in the OMG sense. Each system environment shall establish a format for recognizing the identity of subscribers and other End User Clients. This is used on all RPC messages, as well, for identifying the requester. It is also used in DSM-CC for identifying the owner of an object, and would commonly be used for obtaining a Client's permissions to access or perform operations on an object. **ServiceContextList** carries optional application-specific information passed between Client and Service.

5.6.6.1.1 Partial Path

On a ClientSessionSetupConfirm, a valid session can exist even though the entire **pathName** has not been resolved. The partial path must include at least the object reference of the first **Step** in the path, a ServiceGatewayUU. The length of the resolved references specify the portion of the path that was successfully resolved.

5.6.6.2 ServiceGatewayUU

ServiceGatewayUU inherits the Directory interface. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGatewayUU object.

The Client only sees the local ServiceGateway object at the Application Portability interface. This object translates to the Service Inter-operability ServiceGatewayUU interface, if the U-N Session protocol is used. The Server in this case would implement the ServiceGatewayUU interface.

5.6.6.2.1 Summary of ServiceGatewayUU Primitives

Inherited from **Access**: attributes: **Size, Hist, Lock, Perms** Inherited from **Directory and NamingContext**: operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**	

Service Inter-operability Syntax

```
module DSM {
    interface ServiceGatewayUU : Directory {
        const AccessRole bind_ACR = MANAGER;
        const AccessRole bind_context_ACR = MANAGER;
        const AccessRole rebind_ACR = MANAGER;
        const AccessRole rebind_context_ACR = MANAGER;
        const AccessRole unbind_ACR = MANAGER;
    };
};
```

TapUse

RPC_USE

5.6.6.3 SessionSI

The SessionSI interface may be used in systems where there is no DSM-CC User-to-Network signaling. It provides operations to establish a Session without management of network resources. The Session in this case is a context for active object references and their state, plus it may serve to identify the requesting Principal involved.

The Client only sees the local ServiceGateway object at the Application Portability interface. This object in turn calls the Service Inter-operability ServiceGatewaySI interface, if the RPC Session protocol is used. The Server in this case would implement the ServiceGatewaySI interface.

Service Inter-operability Syntax

```
module DSM {
    interface SessionSI {
        const AccessRole attach_ACR = READER;
        void  attach (
                in InfoRequest downloadInfoReq,   // download info request
                in CosNaming::Name pathName,      // path name to resolve
                in UserContext savedContext,      // previous application user context
                out InfoResponse downloadInfoResp,      // download info response
                out ObjRefs resolvedRefs)         // objects resolved
                raises (NO_AUTH, UNK_USER, OPEN_LIMIT, SERVICE_XFR,
                        BAD_COMPAT_INFO, NO_RESUME,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
        const AccessRole detach_ACR = READER;
        void  detach (
            in boolean aSuspend,
            out UserContext savedContext)         // suspended user context
            raises (NO_SUSPEND);
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

SessionSI attach() is invoked against a local SessionSI object, which acts as a logical path root to remote Services. The first node in the pathName is that of a ServiceGateway. The SessionSI object shall resolve the ServiceGateway, and if the path extends beyond the ServiceGateway, shall propagate the resolve.

A Client invokes **SessionSI attach()** to open a new Session over the network using an RPC. **downloadInfoRequest** and **DownloadInfoResponse** contain Download negotiation parameters as specified in the DownloadSI interface. An identification of a previously suspended user context **savedContext** enables the Client to indicate that an application is to resume from previously suspended state. If this is set to 0, it indicates the application is starting up for the first time. A path specification **pathName** names the path to ServiceGateway and possibly a first Service to be opened. A default Service may be specified in the second **Step** of **pathName** as a NULL string **NameComponent.id**. A default Service **NameComponent.id** is an indication to the ServiceGateway that it can choose the first Service for the Client. **resolvedRefs** shall contain references for the resolved object references.

A Client invokes **SessionSI detach()** to close all object references for a Session using an RPC. If **aSuspend** is true, the ServiceGateway will inform Services which are maintaining user context for this End User to return user context state for a possible resumption of those Services. It is up to the application to determine which state is to be returned and maintained between Sessions. The Client may later invoke **attach()** with this **UserContext** in order to resume from the saved state.

5.6.6.4 ServiceGatewaySI

ServiceGatewaySI inherits the Directory and SessionSI interfaces. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGatewaySI object.

5.6.6.4.1 Summary of ServiceGatewaySI Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**

Inherited from **SessionSI**:
operations: **attach, detach**

Service Inter-operability Syntax

```
module DSM {
    interface ServiceGatewaySI : Directory, SessionSI {
        const AccessRole bind_ACR = MANAGER;
        const AccessRole bind_context_ACR = MANAGER;
        const AccessRole rebind_ACR = MANAGER;
        const AccessRole rebind_context_ACR = MANAGER;
        const AccessRole unbind_ACR = MANAGER;
    };
};
```

TapUse

RPC_USE

5.6.6.5 DownloadSI

When an RPC stack is present in the Client, the DownloadSI interface can be used to perform lower layer Download requests. This interface implements model 2 of the Download Specification in Chapter 7 (where ControlUp, DataUp and ControlDown are carried on an RPC bi-directional channel). All control messages including DataRequest are exchanged over this interface. The DownloadDataBlocks are sent over a high-bandwidth downstream channel, e.g., one which conveys an MPEG-2 stream. Thus, taps are returned for the RPC and for the data downstream channels. While the Download Application Portability Interface is simplified by not exposing network flow-control to the application, the DownloadSI Service Inter-operability Interface at the lower layer will perform the details of windowing and flow control as described in chapter 7. The RPC interface provides a reliable Download plus invocation authorization on each message, where higher security is desired.

The pre-conditions for this interface are that a Client can send messages to an object reference for an application Service that supports this interface. The first operation is a **DownloadSI info()**. The reply to this RPC is a sequence of DownloadInfo structures that contain size, private opaque data, and description of each module that must be downloaded in order for the Client to run the application. Following this, a series of **DownloadSI proceed()** messages to be sent to control the Download transfer. The request message body of **DownloadSI proceed()** is the DownloadDataRequest as described in chapter 7. A **DownloadSI cancel()** can be sent to cancel a download in progress.

The Client can recognize an object that supports the Download interface through a **Directory list()** operation at the parent directory where the object's name and object reference are bound. NameComponent kind in the **Directory list()** reply identifies whether the object is a Download Service. Following a resolve of an object reference, **Kind has_a()** may be invoked to test whether Download is supported as an inherited interface.

The Download data encoding of Chapter 7 also supports broadcast download in various forms. This does not require RPC control as described in this section.

CDR Lite encoding will convert the DownloadSI programming interface to exact data encoding specified in Chapters 6 and 7.

Service Inter-operability Syntax: InfoRequest and InfoResponse

```
module DSM {
    struct SubDescriptor {
        octet subDescriptorType;
        octet subDescriptorLength;
        sequence<octet, 255> additionalInformation;
    };
    struct InterfaceDescriptor {
        octet descriptorType;
        octet descriptorLength;
        u_long specifier;
        u_short model;
        u_short version;
        sequence<SubDescriptor, 255> subDescriptorList;
    };
    struct CompatibilityDescriptor {
        u_short length;
        sequence<InterfaceDescriptor, 65535> interfaceDescriptorList;
    };
    struct InfoRequest {
        u_long bufferSize;
        u_short maximumBlockSize;
        CompatibilityDescriptor userCompatibilitiesBytes;
        sequence<octet, 65535> privateDataBytes;
    };
    struct ModuleInfo {
        u_short moduleId;
        octet moduleVersion;
        u_long moduleSize;
        sequence<octet, 255>  moduleInfoBytes;
    };
    typedef sequence<ModuleInfo, 65535> ModuleInfoList;
    //
    // The complete InfoResponse contains Server calculated
    // transport parameters plus application level information
    //
    struct InfoResponse {
        u_long downloadId;
        u_short blockSize;
        octet windowSize;
        octet ackPeriod;
        u_long tCDownloadWindow;
        u_long tCDownloadScenario;
        CompatibilityDescriptor userCompatibilitiesBytes;
        ModuleInfoList modulesInfo;
        sequence<octet, 65535> privateDataBytes;
    };
};
```

Service Inter-operability Syntax: Operations

```
module DSM {
     interface DownloadSI {
#ifdef DSM_CONSUMER
          // Download DataRequest is sent to start the Download or
          // to cause the Server to send the next group of Download DataBlocks
          struct DataRequest {
               u_short moduleId;
               u_short blockNumber;
               octet downloadReason;
          };
          // when the Server receives a Download DataRequest, it
          // will send a series of ackPeriod Download DataBlocks:
          struct CancelRequest {
               u_short moduleId;
               u_short blockNumber;
               octet downloadCancelReason;
               char privateDataLen;
               sequence<octet, 65535> privateDataBytes;
          };
          const AccessRole info_ACR = READER;
          void info (in InfoRequest reqNegotiation,
                    out InfoResponse respNegotiation)
               raises (BAD_COMPAT_INFO, BUF_SIZE, BLOCK_SIZE);
          const AccessRole proceed_ACR = READER;
          void proceed (in DataRequest ackNack)
               raises(BAD_MODULE_ID, TIMEOUT, MPEG_DELIVERY);
          const AccessRole cancel_ACR = READER;
          void  cancel (in CancelRequest cancelReq)
               raises (BAD_MODULE_ID, TIMEOUT) ;
#endif
#ifdef DSM_GENERAL
          struct ModuleInstallInfo {
               u_short aModuleId;
               // path to object containing Download data
               CosNaming::Name n;
          };
          typedef sequence<ModuleInstallInfo> ModuleInstallList;
          };
          const AccessRole install_ACR = OWNER;
          void install (in CompatibilityDescriptor compatInfo,
                     in ModuleInfoList modulesInfo,
                     in ModuleInstallList pathsInfo)
               raises (BAD_COMPAT_INFO, BAD_MODULE_INFO,
                              INV_NAME, NOT_FOUND);
          const AccessRole deinstall_ACR = OWNER;
          void deinstall (in CompatibilityDescriptor compatInfo)
               raises (BAD_COMPAT_INFO);
#endif
     };
};
```

TapUse

RPC_USE	operation requests
DOWNLOAD_DATA_DOWN_USE	Download DataBlocks

Service Inter-operability Semantics

DownloadSI info() is used to obtain download module information and negotiate flow-control parameters for a Download. **DownloadSI info()** enables a Client to obtain information about modules that must be downloaded in order for an application to proceed. It is also used to negotiate flow-control parameters based on buffer size, block size and network reliability. For efficiency reasons, it is recommended that the first InterfaceDescriptor in the CompatibilityDescriptor identify the dominant Client hardware specifier, model and version. The **InfoRequest** and **InfoResponse** fields are described in Chapter 7.

DownloadSI proceed() is used iteratively to transfer a series of DownloadDataBlocks to the Client. The DownloadDataBlocks are normally transfered over the channel identified by the DOWNLOAD_DATA_DOWN_USE Tap. The **DataRequest** fields are described in Chapter 7.

DownloadSI cancel() is used to cancel a Download in progress. The **CancelRequest** fields are described in Chapter 7.

Informative: A new interface type that inherits both File and Download is possible. In this case, the transport-level download protocol described in Chapter 7 can be used for large transfers requested by **File read()**. To accomplish this, the **InfoRequest** is carried in the ServiceContextList of the associated operation. The **InfoResponse** and **ConnBinder** are carried in the ServiceContextList of the RPC reply. The data to be transferred is treated as a single module to be downloaded. The negotiation of flow-control parameters is the same as the DSM-CC Download protocol. Since the Client is making the selection, there is no compatibility matching. This mechanism may also be used for other objects that have operations that potentially require large data transfers. The Download interface must be supported by the target object.

DownloadSI install() enables an owner to bind download configuration to the Server. The configuration consists of the compatibility information that describes the Client configuration, module information for the modules to be downloaded, and the path name to download module data for each module. The **CompatibilityDescriptor** fields are defined in Chapter 6.For efficiency purposes, it is recommended that the first InterfaceDescriptor in the CompatibilityDescriptor identify the dominant Client hardware specifier, model and version. The ModuleInfoList contains information for each module to be downloaded. ModuleInfo fields are defined in Chapter 7. The ModuleInstallList contains the pathNames to objects at the Server that contain the download module data.

DSM DownloadSI deinstall() enables an OWNER to unbind a Download configuration from the Server. The Client CompatibilityDescriptor identifies the download configuration to be removed.

5.6.6.6 Event

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface, except for **Event notify()**, which is a local operation. Please refer to the Event API for individual operation descriptions. Note that **Event notify()** is implemented as a local library operation at the Client. To the IDL Compiler, defining DSM_SERVER will cause **Event notify()** to not be compiled for the Server Stub.

Application Portability / Service Inter-operability Syntax

```
module DSM
    interface Event {
        // In addition to the other descriptor fields, the stream object places the
        // StreamEvent in the private data section of the media stream:
        const u_short NULL_EVENT_ID = 0;
        typedef sequence<char, 255> eventName;
        typedef  sequence<eventName, 65535> EventList_T;
        const AccessRole EventList_get_ACR = READER;
        const AccessRole EventList_put_ACR = OWNER;
        attribute EventList_T EventList;
        // the following struct is sent in the MPEG stream
        //
        struct StreamEvent {
                u_short eventId;
                AppNPT rAppTime;
                sequence<octet>  rPrivateData;
        };
        const AccessRole subscribe_ACR = READER;
        void subscribe(
                in string aEventName,
                out u_short eventId)
                raises(INV_EVENT_NAME);
        const AccessRole unsubscribe_ACR = READER;
        void unsubscribe(
                in u_short eventId)
                raises(INV_EVENT_ID);
#ifdef DSM_PSEUDO
        void notify (out StreamEvent rStreamEvent);
#endif // DSM_PSEUDO
     };
};
```

Possible TapUses

RPC_USE	operation requests
STREAM_EVENT_USE	Stream Event Descriptors
STR_STATUS_AND_EVENT_USE	Stream Mode and Event Descriptors

Service Inter-operability Semantics

The local Event object shall monitor the MPEG-2 Stream for Stream Event descriptors that match the eventId for subscribed events. It shall hold event information for each eventId until such time an application level **Event notify()** is invoked or the object is closed.

5.6.6.7 Composite

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Composite API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    typedef u_long BindingType;    // extends CosNaming BindingType
    const BindingType cobject = 2;
    interface Composite {
        struct ChildInfo {
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
        };
        typedef sequence<ChildInfo> ChildInfos;
        //
        const AccessRole list_subs_ACR =READER;
        void list_subs (
             in CosNaming::Name name,
             out ChildInfos infos)
            raises (NOT_FOUND, INV_NAME);
#ifdef DSM_GENERAL
        struct ChildBinding{
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
            ObjRef obj;
        };
        typedef sequence<ChildBinding>  ChildBindings;
        const AccessRole bind_subs_ACR = WRITER;
        void bind_subs (
            in CosNaming::Name name,
            in ChildBindings rChildBindings)
            raises (NOT_FOUND, INV_NAME, ALREADY_BOUND);
        const AccessRole unbind_subs_ACR = WRITER;
        void unbind_subs (in CosNaming::Name name)
            raises (NOT_FOUND, INV_NAME);
#endif
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

Directory open() with **aPathType** = DEPTH must be used to resolve a Composite object. The ObjRefs returned are the sequence of object references whose corresponding **aPathSpec Step Process** flags areTRUE (the last of which being the Composite Parent object reference), followed by the Child object references.

While **Directory open()** at the Application Portability Interface returns the parent Composite to the Client Application, the corresponding **Directory open()** at the Service Interoperability Interface returns the object references of the Composite parent and the required child objects over the network. The child objects are associated as being compatible versions, such that when the **Directory open()** is invoked, the set of required compatible sub-objects are resolved.

Opening a required child is a local operation, because the object reference is present from the previous **Directory open()**. Opening an optional child causes an object reference for the child to be returned over the network.

With **Composite bind_subs()**, subset of child objects are labeled as required, i.e., they should be opened together by a **Directory open()** operation. Another subset are labeled optional, i.e., they may be opened individually at any time during the running of the application. The optional sub-objects are included with the composite set for version compatibility reasons.

Each child object reference shall have a NameId in its IOR to provide the child's symbolic name.

5.6.6.8 View

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the View API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax: View Styles

```
module DSM
    interface View{
        // View Style identifies the Query set supported by the View
        // NON_DB indicates service is not a Database but performs minimal searches,
        // filters and sorts using SELECT as described in the DSM-CC spec
        // SQL89 indicates the View is a SQL89-compliant database
        // SQL92 indicates the View is a SQL92-compliant database
        // SQL3 indicates the View is a SQL3-compliant database
        const char NON_DB = ‘N’;
        const char SQL89 = ‘1’;
        const char SQL92 = ‘2’;
        const char SQL3 = ‘3’;
        const AccessRole Style_get_ACR = READER;
        readonly attribute char Style;
    };
};
```

Application Portability / Service Inter-operability Syntax: Statement and Result

```
module DSM {
    interface View {
        typedef string SQLStatement;
        //
        typedef u_short FieldCode;
        struct FieldDescribe {
                string fieldName;           // name of the field
                FieldCode aType;                    // type of the field
                opaque typeParameters;              // parameters related to the given type
        };
        typedef sequence<FieldDescribe> ResultDescribe;
        //
        // FieldCodes for standard SQL types
        const FieldCode VTC_CHAR =                  1;
        const FieldCode VTC_SMALLINT =              2;
        const FieldCode VTC_INTEGER =               3;
        const FieldCode VTC_FLOAT =                 4;
        const FieldCode VTC_SMALLFLOAT =            5;
        const FieldCode VTC_DECIMAL =               6;
        const FieldCode VTC_REAL =                  7;
        const FieldCode VTC_DOUBLEPRECISION =       8;
        const FieldCode VTC_CORBA_TYPECODE =        9;
        //
        // these structs are placed into the opaque typeParameters field of
        // the FieldDescribe struct depending on the value of the FieldCode
        struct InfoChar {
                u_short length;
                boolean nullTerminated;
        };
        struct InfoFloat {
```

```
                u_short precision;
         };
        struct InfoDecimal {
                u_short precision;
                u_short scale;
        };
        //
        // type definitions for the returned data
        typedef opaque Field;
        typedef sequence<Field> Row;
        typedef sequence<Row> Result;
    };
};
```

Application Portability / Service Inter-operability Syntax: Operations

```
module DSM {
    interface View {
        // query() executes an SQL query (select) in the server and
        // returns the first set of rows
        const AccessRole query_ACR = READER;
        void query(
                in SQLStatement aSQLStatement,
                in u_short maxRows,
                out ResultDescribe describe,
                out Result aResult,
                out View iterator)
                raises (ILLEGAL_SYNTAX);
        // read() returns additional rows from the query
        const AccessRole read_ACR = READER;
        void read(
                in u_short aCursor,
                in u_short maxRows,
                out Result aResult)
                raises(NO_QUERY, INV_CURSOR);
        // execute() executes an SQL statement in the server (for example “update”)
        const AccessRole execute_ACR = WRITER;
        void execute(
                in SQLStatement aSQLStatement)
                raises(ILLEGAL_SYNTAX);
    };
};
```

TapUse

RPC_USE

5.6.6.9 State

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the State API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface State {
        const AccessRole suspend_ACR = READER;
        void suspend(
                in boolean          aRelease,
                out UserContext  savedContext)
                raises (NO_SUSPEND);
        const AccessRole resume_ACR = READER;
        void  resume (
                in UserContext    savedContext,
                out ObjRefs        restoredRefs)
                raises (NO_RESUME);
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

In addition to suspending application state, **State suspend()** provides the **aRelease** flag for management of network resources. If **aRelease** is true then the object instance forces network resources to be released immediately. When **aRelease** is not set to true the object instance preserves the association tags but the resources might be reassigned or released after a local time-out. In either case, if this object is a parent Composite with open child objects, **State suspend()** cascades to the child objects.

A **ConnBinder** must be present in the object reference (IOR) of the **State resume()** RPC reply.

5.6.6.10 Interfaces

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Please refer to the Interfaces API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
#ifdef DSM_GENERAL
    // A TypeCodeBuf holds a Corba 2.0 TypeCode
    typedef opaque TypeCodeBuf;
    typedef sequence<TypeCodeBuf>  TypeCodeList;
    //
    interface Interfaces {
        typedef opaque ReferenceData;
        typedef string InterfaceDef;
        const AccessRole show_ACR = READER;
        void show (
                in string aStrKind,         // DSM-CC format, "Module::Interface Type"
                in IFKind anIFKind,         // IFKind
                out string rIDL,            // IDL used in previous define
                out IntfCode rIntf,         // DSM-CC interface code with included IFKinds
                out TypeCodeList  rTypes)// TypeCodes required at this level of interface
                raises (NOT_DEFINED, INV_KIND);
        const AccessRole define_ACR = MANAGER;
        void define (
                in ReferenceData id,        // unique Identifier
                in InterfaceDef  rIDL,      // IDL definition
                out IntfCode rIntf,         // DSM-CC interface code with included IFKinds
                out TypeCodeList  rTypes)// TypeCodes required at this level
                raises (PREV_DEFINED, ILLEGAL_SYNTAX, NO_REF_TYPE);
        const AccessRole check_ACR = MANAGER;
        void check (in IFKind  anIFKind)
                raises (NO_REF_TYPE);
        const AccessRole undefine_ACR = MANAGER;
        void undefine (in ReferenceData id, out IFKindList usedBy);
    };
#endif
};
```

TapUse

RPC_USE

5.7 Application Boot Process

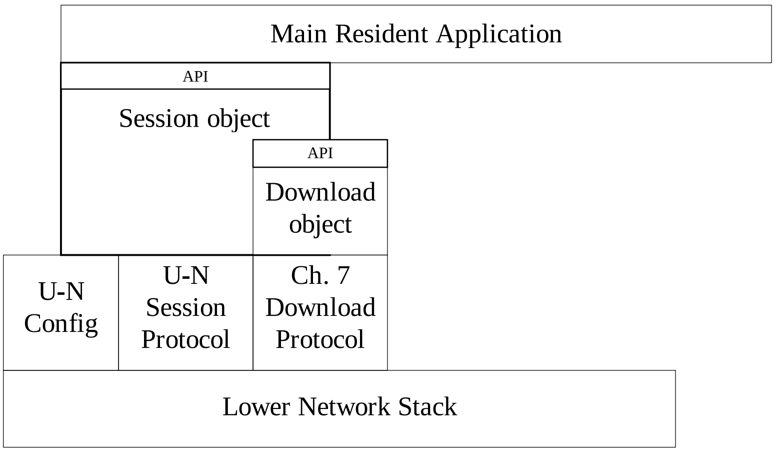
This section describes the process that establishes a Session, performs Download, and starts a Client application. The process consists of a series of steps, each with pre and post-conditions. It is possible that certain Client configurations will satisfy the post-conditions for a step through previously resident functional elements. Thus, if post-conditions for a given step are already satisfied, that step need not be performed.

The Application boot process described here is informative. The process conforms to the U-N messages defined in Chapter 4, the Download messages defined in Chapter 7, and the User-to-User operations defined in this chapter. The operations and their supporting IDL definitions are normative. The pre and post-conditions are normative.

The Application Boot Process encompasses the following lifecycle phases:

1. **Session attach()** is invoked by the Main Resident Application on a local Session object, in order to establish a Session, set up connections to initial remote Service objects, download the functional elements needed to run the application, and decode the parameters that will be used by application.
2. Following **Session attach()** , the Client application is launched.
3. The newly started application invokes **First service()** to obtain the object reference for its first remote Service. The application may also invoke **First root()** to obtain the object reference for the ServiceGateway. During the Session, it may resolve new object references, and invoke object operations on local Object Implementations and active object references.
4. When finished, **Session detach()** is invoked by the Client Application on the object reference of the ServiceGateway, in order to teardown the Session and disconnect from remote Service objects.

5.7.1 Session attach() Pre-conditions



The following pre-conditions must be satisfied in order to call **Session attach()** for the U-N Case.

- Lower Network Stack is present. It includes the Physical, Data link, Network and Transport layers of the OSI Model.
- Session input parameters are present.
    1. ClientId. ClientId is carried in the U-N protocol to globally identify the Client device. ClientId is a 20 octet value.
    2. Service location information. These include ServerId, optional path name, and optional user context. The 20 octet ServerId globally identifies the Server-side device that will negotiate the U-N messages. There is a 1-1 relationship between ServerId and ServiceGateway, i.e., there may only be one ServerId for a given ServiceGateway instance. The path name is the complete logical path from a ServiceGateway to a Service. The user context is an opaque value that holds application initial parameters
    3. CompatibilityDescriptor. The CompatibilityDescriptor contains configuration information to clearly distinguish the Client profile, e.g., system hardware descriptor, system software descriptor, and descriptor(s) identifying the network protocol(s) the Client supports. This descriptor can identify the protocol to be used for Download. In the absence of such identification, the Download message set of Chapter 7 shall be used as the default.
    4. Transaction timeout values. Timeouts can occur on the U-N and Download message sequences, resulting in a cancellation of the transaction. The Timeouts are specified in the U-N and Download Chapters.
- Other parameters are also optionally present: PrincipalId to identify the End-User to a Service Domain (there may be more than one PrincipalId for more than one Service Domain). Service Context information such as preferences. DSM-CC does not define how these are obtained. They may be included in the Server Binding.

Additionally, a display name may be associated with the path name. The display name is the End-User's alias for an application or Service name. It is recommended that the display name be formated as a UNIcode string, to accommodate alphabets with more than 255 characters.

- U-N Session Protocol is present. The U-N Session protocol includes U-N Session state machine and the ability to send/receive U-N messages.

- Download Protocol is present. The Download protocol includes the encoded Download message set. The Download message set is replaceable by other protocol, such as broadcast Download or RPC. A Download InfoRequest message including CompatibilityDescriptor is pre-encoded.

- Session Local Object Implementation is present. The Session object provides attach() and detach() operations to the upper programming layer, the Session state machine, and U-N Session encoded protocol to the lower layer. The Session object is replaceable by other Session objects that could interface with different lower layer protocols (such as broadcast or RPC).

- Download Local Object Implementation object is present. The Download object provides info(), alloc(), start() and cancel() operations to the upper programming layer, the Download state machine, and Download encoded protocol to the lower layer. The Download object may also interface to other lower layer protocols, such as broadcast Download or RPC-based Download.

- A Main Resident Application is running. This application shall invoke **Session attach()**, and then launch the newly loaded Client application.

5.7.2 Session attach() Procedure

For an interactive Session based on DSM-CC User-to-Network signaling, **Session attach()** performs the following steps:

1. Resolve path-specific parameters to be used in U-N Session Establishment, if necessary.
2. Use **SessionUU attach()** to populate the uuData of Session Establishment messages, and establish the U-N Session. The syntax is defined in the Service Inter-operability Interfaces section of this Chapter.
3. Perform a Download sequence to download DSM-CC Library stubs, application Client executable and other functional elements needed to run the application.
4. Decode Downloaded data to produce progamming level structs and parameters, and bind ObjectReferences with network resources.

5.7.2.1 Resolving Path-specific Parameters

Path-specific parameters may be known in advance by the calling application, or may be stored in a local binding list in the Session object. This sub-section is informative, and describes the use of the Server Binding List.

The Server Bindings can be used to hold various path-specific parameters required for U-N Session Establishment. The Session object maintains this state to enable resolution of a ServerId and SavedContext, given a PathName. At a minimum each Server Binding contains fields for ServerId, CosNaming::Name, and SavedContext. A default ServerId can exist with a NULL (length 0) PathName.

Server Binding List:

ServerId 20 octet	PathName CosNaming::Name	SavedContext opaque

A default Server Binding with a ServerId and NULL PathName is used if the caller of Session attach() does not have a name for a remote Service.

A display name, if present in the ServerBinding, can be used as an End-User alias for the Server Binding. In this case, the Session attach() may accept either the display name alias of the Service (in the NameComponent id) or the full PathName to the Service.

PrincipalId (End-User identification) and ServiceContextList parameters (such as user preferences), may also be present in the Server Binding, or may be global Client parameters common to many Service paths. The default values for these parameters are NULL (zero-length).

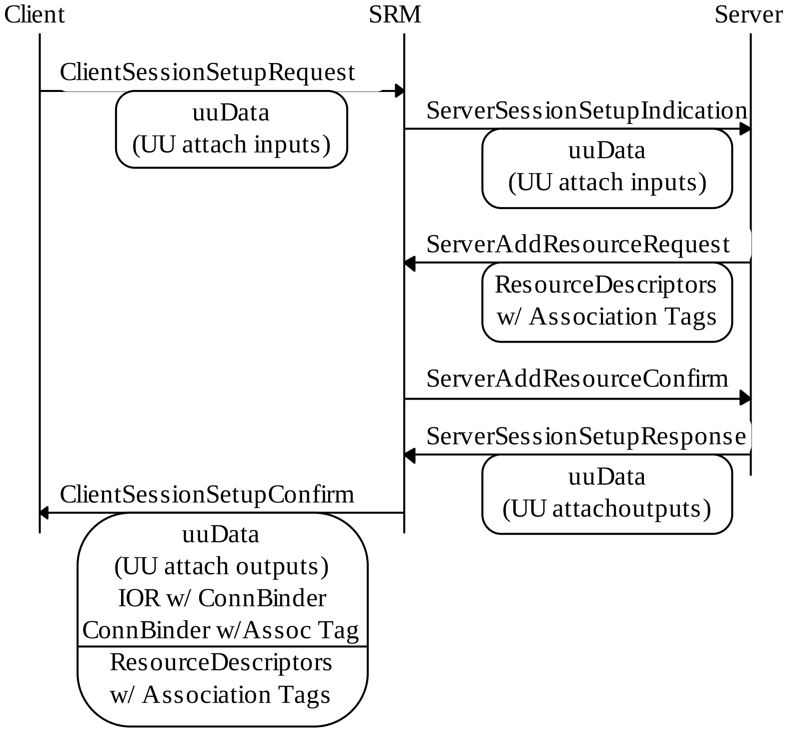
5.7.2.1.1 Post-condition:

These values are known:

ServerId, pathName, savedContext, PrincipalId, ServiceContextList, DownloadInfoRequest

5.7.2.2 Establishing the U-N Session

The **uuData** for all U-N messages is encoded as big-endian CDR. The U-N Session Establishment sequence with **uuData** is shown below:



Note that **_long** in following CDR big-endian data encoding diagrams represents the count of elements in a sequence, according to CORBA terminology.

5.7.2.2.1 ClientSessionSetupRequest

The marshaled input parameters of the **SessionUU attach()** are the **uuData** of ClientSessionSetupRequest, as described below. The resulting **uuData** field of **ClientSessionSetupRequest** is as follows.

- A Download InfoRequest structure.
- A CosNaming::Name identifying a path to the desired Service. The Name extends logically from the remote ServiceGateway. The terminal leaf shall contain a name identifier that identifies the desired initial Service. This is placed in the **pathName** parameter used in ClientSessionSetupRequest.
- A UserContext value, for resumption of previously suspended application state. This is placed in the input **savedContext** parameter.

- An End-User identification of the consumer, unique within the context of the ServiceGateway Service Domain. This is placed in the **aPrincipal** parameter.
- Optional ServiceContextList information, placed in the **inSC** parameter. The ServiceContextList defined later in this section.

The Network shall take the **uuData** of **ClientSessionSetupRequest** and place it in the **uuData** of **ServerSessionSetupIndication**, as part of the Session establishment sequence.

U-U Data Client Session Setup Request	opaque downloadInfoReq		
	CosNaming::Name pathName sequence<>		u_long _length
		CosNaming:: NameComponent	string id string kind
		<other NameComponents>	
	UserContext savedContext		
	Principal aPrincipal		
	IOP::ServiceContextList inSC		

The Server may use the Client CompatibilityDescriptor, Service Name, PrincipalId and ServiceContextList to determine parameters for negotiating session resources with the SRM.

5.7.2.2.2 ClientSessionSetupConfirm

Near the completion of the User-to-Network session establishment, the Server shall return **ServerSessionSetupConfirm** to the network, with **uuData** containing the marshaled output parameters of **SessionUU attach()**. This **uuData** is forwarded to the Client in the **ClientSessionSetupResponse**. The output parameters of **SessionUU attach()** are one of two choices:

1. If the U-N **response** field indicates the session is established:

- Download InfoResponse, placed in the **downloadInfoResp** parameter.
- Connection information for Download data and control channels, placed in the **downloadTaps** parameter.
- Resolved Interoperable Object References(IOR) for the ServiceGatewayUU and optionally for the first Service, as were identified in **pathName**. These are placed in **resolvedRefs** parameter.
- Optional ServiceContextList information, placed in the **outSC** parameter.

U-U Data Client Session Setup Response U-N response field indicates a valid session	opaque downloadInfoResp	
	ConnBinder downloadTaps	
	ObjRefs resolvedRefs sequence<IOP::IOR>	u_long _length
		IOP::IOR string id u_long ProfileId opaque encapsulation
		<other IOP::IORs>
	IOP::ServiceContextList outSC	

2. If the U-N **response** field indicates the session is NOT established, uuData shall contain:

- Exception. This shall contain exception type and value.

U-U Data Client Session Setup Response U-N response field indicates an invalid session	any exception	DSM Typecode	string kind string repositoryId u_long TCKind
		opaque value	

5.7.2.2.3 Session Establishment Post-conditions

A table of tag/ network resource associations contains the Resource Association Tags and Resource Descriptors that were returned in the **ClientSessionSetupResponse**.

The **uuData** is not decoded. **DownloadInfoResponse** and **downloadTaps** are present in CDR big-endian format.The encoding of other **uuData** parameters is not yet known.

The undecoded **resolvedRefs** contains an IOR for the ServiceGatewayUU, and may contain IORs of the first Service. If the first Service is a Composite, IORs are present for the parent and child references. If not, the first Service is referenced by a single IOR.

5.7.2.3 Download

The Download sequence returns all functional elements needed to initially run the application, including the Client Application itself, DSM-CC Library Stubs for objects the application will reference, and the RPC Protocol, if it is not already present.

As part of the Application Boot Process, the execution of Download does not require the RPC to be present. Following Download, the RPC must be present.

downloadTaps from **uuData** of **ClientSessionSetupResponse** provide the connection information for Download data channels. This ConnBinder shall be encoded as CDR, big-endian.

DSM::ConnBinder		u_long _length
	Tap	u_short id u_short use u_short assocTag sequence<octet, 255> selector
	<other Taps>	

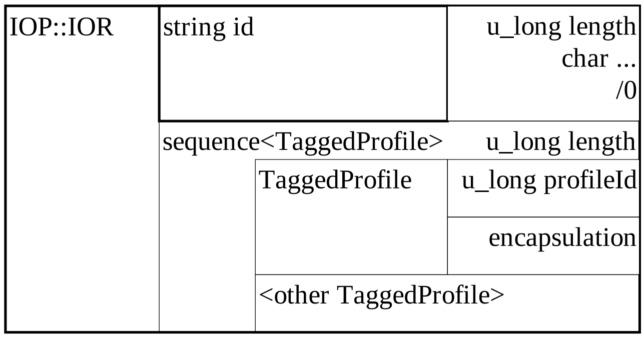
The Download Protocol receives **DownloadInfoResponse** from **uuData** of **ClientSessionSetupResponse.** The Download Client state machine of Chapter 7 is DCActive.

Download start() is invoked by the **Session attach()** procedure to send a series of DataRequest messages to retrieve all required DownloadDataBlocks, as described in Chapter 7.

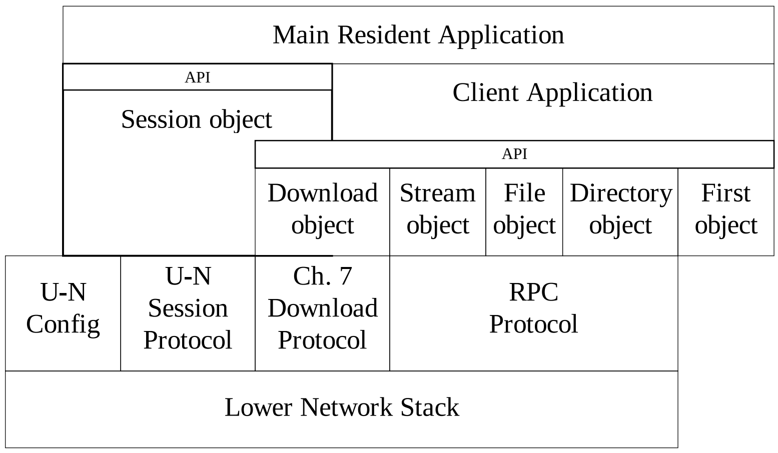
The Download process retrieves byte sequences that must be decoded in order for Client Application to access object structures at a programming level. Following Download, the encoder/decoder must be present, either in the RPC or in the DSM-CC Library stubs.

The IORs (**resolvedRefs**) from **SessionUU attach()** are now decoded to determine to the Protocol ProfileId (and hence the encoding/decoding specification) for each Client/object binding. The IOR contains a string identifier for the object type, a length indicating a number of TaggedProfiles, and an array of TaggedProfiles (DSM-CC typically uses 1 TaggedProfile). The string identifier shall be either the NameComponent kind format (“<Module>::<Interface>”) or an alias (see section ‘Entity Identification’). The Tagged Profile contains a ProfileId and a CORBA 2.0 encapsulation.

The profileId specifies the RPC, the data encoding, and the structure of the encapsulation. The top format of the IOR shall be be big-endian CDR. The encapsulation encoding and all other encodings for the object shall be determined by the ProfileId. The IOR encoding is shown below:



The Downloaded data encodings are now converted to programming level constructs. Download returns, **Session attach()** returns, and the Main Resident Application launches the new Client Application.



5.7.3 Session Tear-down

1. The marshaled input parameters **SessionUU detach()** specify the **uuData** field of **ClientReleaseRequest**. This user data is then forwarded by the network to the **Server** in **ServerReleaseRequest**. The input parameters in **SessionUU detach()** are:
    - A boolean suspend indication which indicates to the Server that application state should be preserved for later resumption. This is placed in the **aSuspend** parameter.
    - Optional ServiceContextList information, placed in the **inSC** parameter.
2. The output parameters of **ServiceGateway detachUU()** are marshaled together and placed in the **uuData** field of ServerReleaseConfirm. This user data is then forwarded by the network to the Client in **ClientReleaseConfirm**. The output parameters of **ServiceGateway detachUU()** are:
    - A UserContext value containing state for later resumption of this session, placed in the **savedContext** parameter. A value of zero will indicate an exception occurred in the process of suspending the session state, and that state from this session is no longer valid.
    - Optional ServiceContextList information, placed in the **outSC** parameter.

5.7.4 Session Transfer Implications

While the User-to-User does not preclude the use of the User-to-Network Session Transfer, a transfer to a new Service Domain implies that object references must be updated in order for the Client to communicate with the new Service. No mechanism is described in DSM-CC for notifying the Client of the new object references, nor for notifying that a transfer is about to take place. In addition no provision has been made in DSM-CC for the Client to download an application from the new Service Domain.

In order to meet the requirement for transferring a Client between Service Domains with the existing User-to-User specification of DSM-CC, User-to-User Service Transfer is defined in the Session interface in this Chapter.

Please refer to Informative Annex L, Service Transfer Message Flows, for further information.