Oracle Video Server™ Developer's Guide

Release 3.0

Part No. A53960-02

ORACLE®

Enabling the Information Age

Oracle Video Server Developer's Guide, Release 3.0

Part No. A53960-02

Printed in the U.S.A.

Primary Authors: Gordon Furbush and Denise Stone

Contributing Authors: Judith Latham and Renate Kempf

Contributors: Parker Lord, Dave Pawson, Dirk Pranke, Dan Weaver, Alex Lind, Ming Lee, Matt Prather, Andrew Samuels, Gregg Stefancik, and Dave Robinson.

Preface

This guide is written primarily for system developers who desire to modify or extend the server-side operations of the Oracle Video Server (OVS). This guide also provides information for developers to create customized OVS client applications that run on set-top boxes or other special devices not supported by the Oracle Video Client (OVC).

Important If you are developing a client application, refer to the *Oracle Video Client Developer's Guide* for details on how to use the Oracle Video Client, which is a set of component objects for developing portable video-enabled client applications. Most client applications can be developed using the Oracle Video Client. If you have special needs that cannot be met using the Oracle Video Client, then you can use this guide to learn how to create your own custom client application.

The examples presented in this guide assume that you are already familiar with the basics of Oracle Media Net (OMN) and the UNIX operating system.

How this Manual is Organized

Most of the chapters in this guide describe how to use the existing OVS services. The remaining chapters provide the background necessary for you to write new services or modify existing services that operate as part of the video server.

The video server provides extensive interfaces that allow you to create a variety of services. The services you create or modify depend on the individual needs of your site, so it's not possible for this document to address the needs of every site.

This manual is organized as follows:

- Chapter 1, Overview of the Oracle Video Server — Describes how applications communicate with the Oracle Video Server. All developers should read this chapter.
- Chapter 2, Creating a Session and Using the Stream Service — Describes the operations used by video client applications to stream video data from the Oracle Video Server. All developers should read this chapter.
- Chapter 3, Manipulating Video Content — Provides an overview of the OVS content data model and its uses.
- Chapter 4, Scheduling Content for Broadcast — Provides an overview of the OVS scheduling interfaces and their uses.
- Chapter 5, Writing a Custom Content Service — Describes how to write a custom content service to provide security and/or implement an alternative content data model.
- Chapter 6, Capturing Stream Events — Describes how to write a Media Net event consumer to capture events generated by the stream service.
- Chapter 7, Extending Video Encoders for Real-time Feeds — Describes how to extend a video encoder to allow data to be streamed to clients while it is being encoded.
- Chapter 8, Accessing Files in the Media Data Store — Describes how to create and manipulate files in the Media Data Store (MDS), as well as how to download images and other non-video data from the MDS.
- Appendix A, OVS Interface Reference — Describes the OVS interfaces that are of interest to both client and server applications.
- Appendix B, OVS Data Types Reference — Describes data types used by OVS methods and functions.
- Appendix C, OVS Exceptions Reference — Describes the exceptions that can be raised or errors that can be returned by OVS methods and functions.

- Appendix D, OVS Stream Events — Describes the events that can be generated by the stream service.
- Appendix E, Video Server Metadata — Describes the format of the header and tag information that make up the metadata used by video content encoded using the MPEG-1, MPEG-2, and Raw Key Compression video formats.
- Appendix F, Oracle Generic Framing — Describes the format of the Oracle Generic Framing (OGF) transport header.
- Appendix G, Setting Up OVS Sample Applications — Describes how to set up and run the sample applications described in Appendix H.
- Appendix H, Example Code Reference — Lists the sample applications described throughout this guide.

Which Chapters Should I Read?

Depending on your development goals, you will probably need to read only portions of this guide. This section describes how specific tasks map to the various chapters and appendixes in this guide.

To understand the basic OVS services and how they work together to stream video to a client, read Chapters 1 and 2, and look at the **ovsdemo.c** sample application in Appendix H. These chapters should be read by anyone wishing to gain a general understanding of the OVS API and how it maps to the OVS architecture described in *Introducing Oracle Video Server*. Keep in mind that most of the developer tasks in Chapter 2 are already handled by the Oracle Video Client (OVC), so be sure to read the *Oracle Video Client Developer's Guide* and understand the capabilities of the OVC before making any decision to write a custom client application of your own.

To learn how to write programs that segment video content into "clips" and combine sequences of clips into identifiable units of "logical content," read Chapter 3 and look at the **ctntdemo.c** and **clipdemo.c** sample applications in Appendix H. You can also schedule logical content for general broadcast over specific "channels" by using the scheduling interfaces described in Chapter 4. If the logical content data model described in Chapter 3 does not meet your needs, Chapter 5, along with the **cont.idl**, **contImpl.c**, **rslvImpl.c**, **dcontsrv.c**, and **contclnt.c** files in Appendix H, describe how you can implement your own content service.

Appendix D lists the events that can be generated by the stream service. To capture these events, you can implement a Media Net event consumer, as described in Chapter 6 and demonstrated by the **eclisten.c** sample application in Appendix H.

To extend a video encoder to load video and metadata on the video server in real-time as it is encoded, read Chapter 7 and Appendix E, and look at the **vesdemo.c** sample application in Appendix H.

To learn how to write programs that manipulate video content files directly in MDS or stream non-video BLOB (Binary Large OBject) data to a client, read Chapter 8.

Appendixes A, B, and C provide a complete reference for the publicly available OVS interfaces, datatypes, and the possible exceptions that may be raised.

Related Documents

Developers of OVS applications and services must be familiar with Oracle Media Net. The Oracle Media Net documents are available as PDF files in the OVS document directory (see your *Oracle Video Server Installation Guide* for the location of the OVS document directory).

There titles are file names of the Oracle Media Net documents are listed in the table below.

Media Net Document	**PDF File**
Oracle Media Net Administrator's Guide	omnadmin.pdf
Oracle Media Net Developer's Guide	omndev.pdf
Release Notes	omnnote.pdf

Conventions Used in This Manual

This guide is available online in PDF format for use by the Acrobat Reader. The online version contains numerous hypertext links that greatly simplify navigation through the guide. These links are particularly helpful when exploring the various methods and their datatypes. Refer to the *Oracle Video Server Roadmap* or *Oracle Video Server Installation Guide* for the location of the PDF version of this document.

This table lists the typographical conventions used in this manual.

Feature	Example	Explanation
boldface	**mna.h** **first::query()**	Filenames, objects, and function names when used in text. New terms.
italics	*file1*	Placeholder in command or function call syntax; replace this place holder with a specific value or string.
fixed width	`void DisableDSM()`	Code example or function prototype.

Worldwide Customer Support

When you or someone in your company acquired this Oracle product, you probably also purchased some level of customer support. Oracle then sent you a package that includes telephone numbers, email addresses, and web sites you should use to contact customer support.

Oracle provides web-based support for our Oracle*MetaLink* and Oracle*Mercury* services at **http://support.us.oracle.com**.

If your company did not purchase customer support, you can visit our web site, **http://www.oracle.com/support**, to find out about Oracle's Worldwide Customer Support services.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following postage address, FAX number, or email address.

Oracle Media Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
M/S 6op5
Redwood City, CA 94065
U.S.A.
FAX: 650 506-7615
email: omsdoc@us.oracle.com

Contents

CHAPTER

1

Overview of the Oracle Video Server

This chapter provides a brief overview of the Oracle Video Server and describes some of the features that are of interest to developers of client and server applications.

The topics covered in this chapter are:

What is an Oracle Video Server?

The Oracle Video Server (OVS) is software that allows you to store and manipulate encoded video data, then stream it to applications running on set-top boxes, personal computers, network computers, and other types of devices. OVS can stream video over many types of standard broadband and narrowband networks.

Client/Server Configuration

OVS devices interoperate as **clients** and **servers** in a distributed object environment supported by **Oracle Media Net** (OMN). OMN is the Oracle implementation of the Common Object Request Broker Architecture (**CORBA**), which is an extensible industry-standard architecture for distributing objects between computer systems connected over a network. OMN is discussed in *Using Oracle Media Net* on page 1-10 and the *Oracle Media Net Developer's Guide*.

A **client device** is a piece of hardware used to access the video server, such as a set-top box, a personal computer, or a network computer. Each of these devices must have a network interface to allow communication with a server. The **client applications** that run on the client device communicate through OMN with specialized server applications, or **services**, that perform specific tasks on behalf of the client, such as provide the client with information on available videos. Note that the client and server communicate through OMN to set up and control the streaming of video, but the video data itself is transported as real-time data directly to clients using standard network protocols, and is *not* transported through OMN.

The term **server** is used in this document to describe a group of services that work together to provide clients with access to video and other types of OVS data. These services typically run on one or more server-class computers connected over networks. The various server hardware configurations are described in *Introducing Oracle Video Server*.

Why Write New OVS Applications and Services?

The Oracle Video Server software distribution includes the Oracle Video Client (OVC), which is a set of component objects for developing portable video-enabled client applications. The OVC runs on a variety of hardware platforms, and provides the basic tools you need to stream video from the OVS server. You can use the Oracle Video ActiveX Control, Java, and Web Plug-in features described in the *Oracle Video Client Developer's Guide* to extend the capabilities of OVC and write many kinds of useful and sophisticated video client applications.

Most client applications can be developed using the Oracle Video Client. If, after reviewing the *Oracle Video Client Developer's Guide*, you don't believe your needs can be met using the Oracle Video Client software, then you can use this guide, the *Oracle Video Server Developer's Guide*, to learn how to create your own custom client application.

This guide addresses the needs of developers implementing customized applications to provide capabilities that cannot be provided by OVC or the existing OVS services. Examples of such custom applications include:

- Client applications for set-top boxes or other client devices not supported by OVC. Other custom video clients may be written to provide tighter integration with custom server applications, such as a custom broadcast scheduler, dynamic content producer, or to allow greater control of the decode and rendering for overlays, image processing, and so on.
- "Back end" applications that enable dynamic manipulation of logical content, dynamic ad insertion into streams, real-time encoding, content usage monitoring, and so on.
- Custom OVS services that enable special control of video content and streams. Such services might be developed to provide the ability to locate, analyze, and manipulate selected portions of video, warehouse and manage large video libraries, and so on.

Summary of OVS Services

The OVS services of interest to most application developers are illustrated in Figure 1-1. You can communicate with these OVS services using the interfaces described in Appendix A, OVS Interface Reference.

The remainder of this chapter describes these services in more detail:

- **Session Service** — establishes and maintains the client/server communication channels and manages a set of OVS resources on behalf of a particular client device.
- **Content Service** — provides the client with information on the content available to it from the OVS system.
- **Stream Service** —instructs the **video pump** to stream video to the client and performs rate control operations. The video pump is described in *Introducing Oracle Video Server*.
- **Media Data Store Services**— store and provide access to all of the media data in the OVS system. Examples of media data include video, **BLOBs** (Binary Large OBjects), and tag files.

  MDS services are typically used internally by other OVS services. However, clients can use MDS services to download application data stored in MDS (for example, BLOBs).

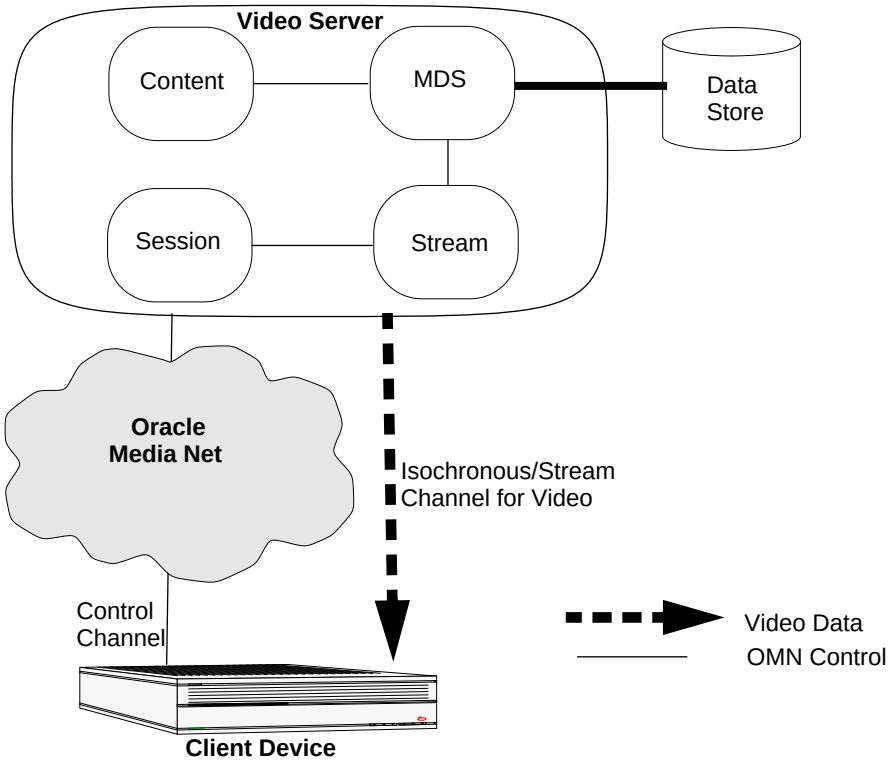


Figure 1-1: Overview of Client Communication with the Oracle Video Server

Session Service

The primary task of the session service is to manage a set of server resources for a particular client device. A client session maintains the mappings between the different components of the system on behalf of the client.

A session can be created in different ways. In the simplest case, the client, such as a PC in an intranet environment, allocates the session and releases it when it is no longer needed. This procedure is described in *Create a Session* on page 2-2.

Only one session is allowed for each client device. If the server allocates the session for that client twice, or if the client tries to allocate the session again, an error is generated.

Circuits

Each session is associated with one or more **circuits**. A circuit is one or more communication channels between the server and a client. A **channel** can be either an **upstream channel** that describes a connection from the client to the server, a **downstream channel** that describes a connection from the server to the client, or a **bidirectional channel** that describes a two-way connection between the client and server. A circuit can consist of one or more channels, which may be either upstream or downstream. The channels contained within a circuit are not necessarily unique to that circuit and may be used by other circuits.

A circuit may be **unidirectional** (up *or* down) or **bidirectional** (up *and* down). A circuit composed of a single channel is a **symmetric circuit**, a circuit composed of two channels is an **asymmetric circuit**.

A circuit can be one of four types:

- **Unidirectional Asymmetric Circuit** — a circuit consisting of a channel in one direction and a null channel in the other direction. (For example, a downstream channel and a null upstream channel.)
- **Bidirectional Asymmetric Circuit** — a circuit consisting of two channels (one upstream and one downstream). This type of circuit is often found in broadband networks with set-top devices communicating with a server over a low-bandwidth upstream channel and a high-bandwidth downstream channel.
- **Unidirectional Symmetric Circuit** — a circuit consisting of a single channel in the same direction as the circuit.
- **Bidirectional Symmetric Circuit** — a circuit consisting of a single bidirectional channel.

A circuit can also be described by one or more of these classifications:

- **Control circuit** — This type of circuit is always bidirectional (that is, it either contains an upstream and downstream channel or it contains a single bidirectional channel). A control circuit is associated with a Media Net transport address and is used to transport all Media Net data.
- **Isochronous circuit** — This type of circuit may be used for isochronous data, such as streamed video/audio. An isochronous circuit is typically associated with a single video pump and may be either unidirectional or bidirectional; in practice, it usually consists of a unidirectional downstream channel for streaming video to the client.
- **Data circuit** — This type of circuit may be either unidirectional or bidirectional and does not necessarily have an associated Media Net transport address.

Note Because there is always one session per client, if a device has multiple network interfaces, the device doesn't create multiple sessions, but instead creates multiple circuits for the session.

Content Service

Clients query a **content service** in the video server for available content. The video server associates each piece of content with a server-defined structure, called an **asset cookie**. The contents of the asset cookies are opaque to the client application and are used only by the server.

A client application obtains asset cookies in response to queries on the content service. To obtain video or other types of content from the video server, the client passes the asset cookies associated with the desired content back to the server, which then uses a **content resolver** to "resolve" the asset cookie into the actual pieces of content in the MDS.

Figure 1-2 illustrates how a client retrieves video content from the **vscontsrv** service, which implements both the content query and content resolver interfaces. Other content services and content resolver services may operate differently.

1. OVS client application queries the content service for the presence of one or more video titles.
2. OVS content service returns the asset cookies associated with the desired video content.
3. Client passes the asset cookies to the stream service to prepare the video content to be streamed.
4. Stream service uses the content resolver to resolve the asset cookies
5. Stream service locates the associated video content in MDS.
6. Stream service directs the video pump to stream the video content to the client.

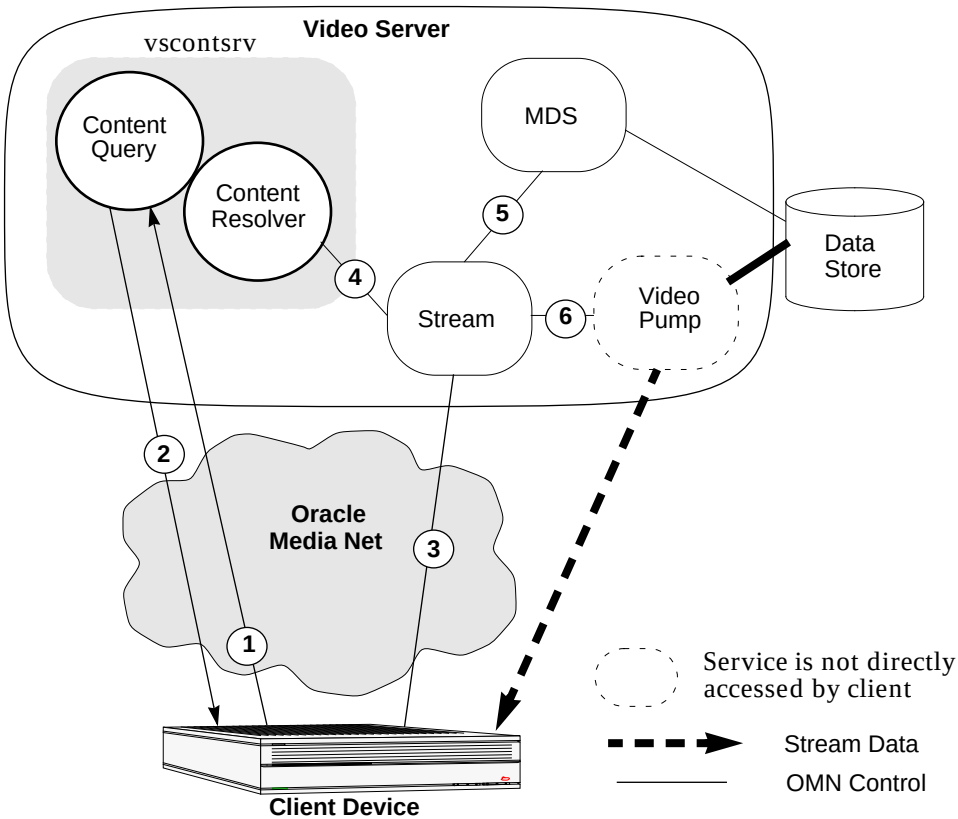


Figure 1-2: Client Obtaining Video Content From OVS

The content service you use may vary, depending on the needs of your OVS system. Some content services may return a reference to an authentication service to authenticate the client. If an authentication service is used, OVS will not deliver any content to the client unless a successful authentication is received.

Content services may also present the content in different ways. Some content services describe content simply as MDS files. Other content services, such as **vscontsrv**, may add layers of abstraction that allow you to construct **logical content** entities that consist of **clips**, which represent selected portions of the content in MDS.

Streams and BLOBs

Clients can receive data from OVS in the form of a real-time stream or a binary object, called a **BLOB** (Binary Large OBject). The server components used to download streams and BLOBs to the client are illustrated in Figure 1-3. Note that the client receives BLOB data over a Media Net stream and real-time data directly over an isochronous stream from the server.

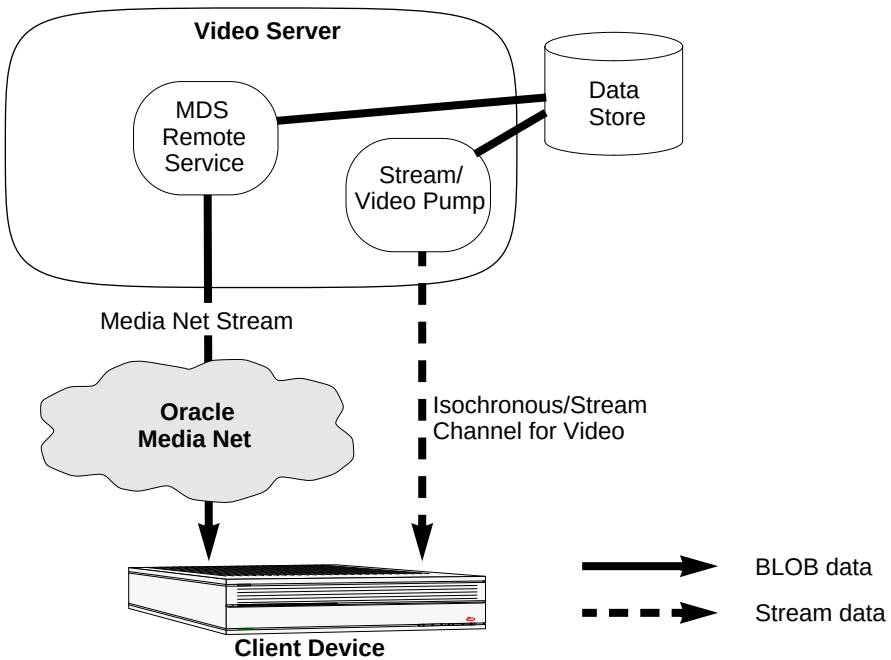


Figure 1-3: Stream Service and MDS Remote Service

Isochronous, or “real-time,” streams are used mostly to transfer large quantities of time-critical data, such as video, that is decoded and displayed in real-time by clients. The video server and networks may have to compromise the reliability of the data to deliver it as a real-time stream, so you should assume that data transported in this manner is inherently unreliable. (See *Detecting Dropped OGF Packets and other Content Discontinuities* on page 2-22 for information on how a client can manage any “gaps” in the video stream.)

BLOBs, on the other hand, are used mostly for data that must be transported in a reliable manner. BLOBs typically transport bitmaps, image stills, or any other data stored on the server that client applications might want to download for local access. Applications running on client devices with limited memory resources can implement their own special callback routines that allocate and deallocate memory for BLOB data as it arrives and is used.

Stream Service

The stream service directs a video pump to deliver OVS data to the client device in the form of a real-time stream. Streams are delivered directly from the server to the client device over an isochronous circuit.

To receive a stream, the client application creates a downstream isochronous circuit from OVS to itself and passes the stream service an asset cookie that specifies the content to be streamed. As described in *Content Service* on page 1-6, the stream service takes care of all of the details of resolving the asset cookie to obtain the correct data from MDS and streaming it to the client.

The stream service currently supports the following video formats:

- MPEG-1 system streams
- MPEG-2 transport streams
- Raw Key Frame, including Oracle Streaming Format (OSF)

See the *Oracle Video Server Content Administrator’s Guide* for more information on video encoding formats.

Media Data Store Services

The MDS and MDS remote services provide file management capabilities.

BLOB support is provided by the Oracle Media Data Store (MDS) remote service. BLOBs are transferred over a Media Net stream. (Do not confuse a Media Net stream with an OVS video stream, which does not use Media Net.)

Most applications that need to store application data in MDS on the Oracle Video Server use the BLOB interface. However, if an application wants to create, modify, or destroy content stored in MDS, they can use the MDS interface. For example, an application that wants to load BLOBs onto MDS, create their own BLOB libraries, examine BLOB or video files, and so on would use MDS (that is, the library described in **mds.h** and **mdsnm.h**).

Note that the MDS interface is very low-level. When an application uses the stream or BLOB interface to access data, content is referenced using asset cookies so that the application does not need to know the name of the associated MDS file(s) or where the pertinent data is located within these files. However, when using the MDS interface, the application has to reference the file by its full pathname, which has the format /mds/*volume_name*/*file_name*.

MDS allows free access to all of its files, so an application should be very careful not to destroy or modify MDS content unintentionally.

For more information on MDS, see Chapter 8, Accessing Files in the Media Data Store and *Introducing Oracle Video Server*.

Using Oracle Media Net

The OVS interfaces described in Appendix A consist of standard C header files (**.h** extension) and CORBA Interface Definition Language files (**.idl** extension). You invoke the functions in the **.h** files as you would any other C-language function. To invoke the methods described in the **.idl** files, you must be familiar with the basics of Oracle Media Net. If you are unfamiliar with how to set up and use Media Net to invoke a method on a remote server, read the first three chapters of the *Oracle Media Net Developer's Guide*.

The code samples used in this guide are based on the sample applications listed in Appendix H. These sample applications use the Media Net functions provided by the Common Object Adapter (COA), which is Oracle's CORBA object adapter implementation.

This section reviews the fundamentals of initializing Media Net, binding to an interface, and invoking methods on that interface. Chapter 5, Writing a Custom Content Service, describes how to create and implement your own Media Net interfaces.

Initialize Media Net

Before you can communicate with Media Net, you must first call the **mtuxInit()** or **mtuxSimpleInit()** function to initialize the Media Net service layer, the network transport layer, and the object manager layer.

The **mtuxSimpleInit()** function initializes the Media Net layers. The **mtuxInit()** function, in addition to initializing Media Net, allows you to pass in an **argument map** that maps command-line options to resource names and values. (See *mtuxInit* on page A-6 and the **ysr.h** file for more information.)

Examples The **mtuxSimpleInit()** function initializes Media Net. The *ctxbuf* is a pointer to memory reserved for Media Net resources, *myprog* is the name of the client application, and *mnLogger* is a pointer to the Media Net logger function.

```
ub1  ctxbuf[SYSX_OSDPTR_SIZE]; /* global context storage space */

if (mtuxSimpleInit(ctxbuf, myprog, (mnLogger)0) != mtuxLayerSuccess)
    exit( 1 );
```

The **mtuxInit()** function initializes Media Net for the program, *myprog*, and accepts an **argument map** named *ovsdemoArgs*. This argument map is created by the **ysargmap()** function to map command-line options to certain resource names and values. For example, the first line in the argument map below maps the command-line option **b** to the resource name "**ovsdemo.bitrate**" and the value **1**. (See the definition for **ysargmap()** in the **ysr.h** file for information on the resource values.)

```
static struct ysargmap ovsdemoArgs[] =
{
  { 'b', "ovsdemo.bitrate", 1 },
  { 'c', "ovsdemo.control-address", 1 },
  { 'C', "ovsdemo.control-protocol", 1 },
  { 'd', "ovsdemo.data-address", 1 },
  { 'D', "ovsdemo.data-protocol", 1 },
  { 'i', "ovsdemo.clientDeviceId", 1 },
  { YSARG_PARAM, (char *)"ovsdemo.blob-name", 1 },
  { YSARG_PARAM, (char *)"ovsdemo.video-tagfile", 1 },
  { 0, "", 0 }
};

ub1  ctxbuf[SYSX_OSDPTR_SIZE]; /* global context storage space */

if (mtuxInit(ctxbuf, myprog, (mnLogger)0, (sword)(argc - 1), argv + 1,
        ovsdemoArgs) != mtuxLayerSuccess)
    exit( 1 );
```

The *ctxbuf* is a pointer to memory reserved for Media Net resources, *ovsdemo* is the name of the client application, and *mnLogger* is a pointer to the Media Net logger function.

Initialize the Object Request Environment

After initializing Media Net, use the **yoEnvInit()** function to initialize a variable (*env*) to represent the object request environment, **yoenv**.

```
yoenv       env; /* active ORB environment */

yoEnvInit(&env)
```

Note The *yoenv* environment variable set by **yoEnvInit()** is used differently by the Media Net Common Object Adapter (COA) than by the Basic Object Adapter (BOA) that is standard in other CORBA implementations. In COA, the environment does not contain any information visible to the user. Also, exceptions in COA are "thrown" and "caught," rather than propagated through the environment as they are in BOA. See the *Oracle Media Net Developer's Guide* for more information on COA exception handling.

You can initialize multiple environments for your application, but one environment is sufficient for the purposes of understanding the code samples in this guide.

Bind to an Interface

Before you can invoke methods on an interface implemented by some remote server, you first must **bind** to an implementation of that interface and obtain an **object reference** from the Media Net Object Request Broker (ORB). In the COA environment, you bind using the **yoBind()** function:

```
object_reference = yoBind (interface_id, impl, ref_data, ignored)
```

The **yoBind()** function tells the ORB to locate a server that implements the interface identified by *interface_id*. The ORB returns an object reference on which you can invoke the methods described by that interface.

The **yoBind()** parameter of most interest is *interface_id*, which is the ID of the interface that was generated by the IDL compiler and registered in the ORB's **interface repository**. For information on the other **yoBind()** parameters and the IDL compiler, see the *Oracle Media Net Developer's Guide*.

Example The **ovsdemo.c** client application described in Chapter 2 first binds to an implementation of the session factory interface (**mzz::factory**) and returns an object reference, *zfac*:

```
mzz_factory zfac;

zfac = (mzz_factory) yoBind( mzz_factory__id, /* interface name */
                             (char *)0,
                             (yoRefData *)0,
                             (char *)0 );
```

Invoke a Method

Every method invoked through COA must include a reference to a remote object that implements the method's interface and a pointer to the ORB environment:

method (*object_reference*, *environment*, *other_parameters*);

The *object_reference* is the reference to the method's interface returned by **yoBind()**, and *environment* is the environment variable initialized by **yoEnvInit()**. Any remaining parameters are specific to the method.

Example After binding to the session factory interface, the **ovsdemo.c** client in Chapter 2 allocates a new session, *ses*:

```
mzz_session ses; /* session object */

ses = mzz_factory_AllocateSession( zfac,         /* object ref */
                                   &env,         /* ORB env */
                                   mzz_sessNull,
                                   &cid,
                                   &spec);
```

Freeing Media Net Resources

All memory allocated to service a client request (such as arguments, return values, and exception data) belongs to the client. The memory may be allocated directly by the client (such as for a context input parameter) or implicitly by the server (such as for a return value or exception). Regardless of who allocated the memory on behalf of the client, it is the client's responsibility to release the memory when it is no longer needed.

Releasing Object References

When you no longer need to invoke methods on a remote object, call **yoRelease()** on your object reference to release any related resources.

Example After obtaining an object reference to the **mzz::ses** object (*ses*), you can release the object reference to the **mzz::factory** object (*zfac*).

```
yoRelease( (dvoid *)zfac );
```

Releasing Complex Datatypes

All complex datatypes returned by the OVS methods must be explicitly freed using a Media Net **__free()** function. (This function is described in the *Oracle Media Net Developer's Guide*.)

Example Use the **mzz::factory::AllocateSession()** method to allocate the session. This method returns an **mzz::session** structure, which contains an object reference for the session object and a structure that contains the attributes of the session.

```
mzz_session ses;

ses = mzz_factory_AllocateSession(zfac, &env,
                                  mzz_sessNull,
                                  &cid,
                                  &spec);
```

When you no longer need the session, free the **mzz::session** structure by invoking an **mzz_session__free()** function, specifying a free function of **yoFree()** to free the *contents* of the structure.

```
mzz_session__free( &ses, yoFree );
```

Releasing Media Net Environment Resources

Before terminating your program, release the Media Net resources in the opposite order in which you initialized them at the beginning of your program.

Example Call **yoEnvFree()** to free the resources used by the object request environment and **mtuxTerm()** to shut down Media Net and release any related resources.

```
yoEnvFree( &env );
mtuxTerm( ctxbuf );
```

CHAPTER

2 Creating a Session and Using the Stream Service

This chapter describes how a typical client application obtains video data from the Oracle Video Server and streams the data to the client device.

The examples presented in this chapter illustrate how to create clients that receive video from OVS over an Internet Protocol (IP) network. However, keep in mind that clients can also be created to receive video over networks based on other protocols, such as Asynchronous Transfer Mode (ATM).

Note Users of the Oracle Video Client (OVC) do not need to be concerned with the APIs described in this chapter. OVC users instead should extend their applications using the ActiveX Control, Java, and Web Plug-in features described in the *Oracle Video Client Developer's Guide*.

Here's a summary of the tasks described in this chapter.

Task	Action	Details on Page
1.	**Create a Session** (and set up the control circuit)	2-2
2.	**Query the Content Service**	2-11
3.	**Prepare Video Content**—Pass the asset cookies for the video content to the stream service	2-15
4.	**Play Video Content**	2-17
5.	**Delete the Session**	2-25

The code samples used in this chapter are based on the **ovsdemo.c** sample application listed on page H-2. Unlike the actual working code presented on page H-2, the code samples show the type declarations where they are used.

Important The **ovsdemo.c** application requires that you install the Media Net run-time libraries and selected IDL files from the server distribution. See the include statements at the start of the **ovsdemo.c** application on page H-2 for the complete list of files that must be installed.

Create a Session

A session is a collection of resources maintained by the server on a client's behalf. A client device may only have one session active at a time on a given server (the client device's unique ID is the session's primary key). If the client has more than one network connection, it can add a new circuit for each connection. Clients can use a single session to manage any number of circuits and other server resources.

A session must:

- Have a client device ID.
- Have at least one control circuit that is operational on the network. This circuit is created when the session is allocated. Additional circuits can be created but are not required.

Summary of Operations

To create a new session for your client, perform the following actions, which are discussed in detail in this section:

Step	Action	Details on Page
1.	**Set the Client Device ID** (a unique ID for the client) by setting the **mzc::clientDeviceId**	2-3
2.	**Create a Network Socket to Receive Video**	2-4
3.	**Allocate a Session and Build a Control Circuit**: • Create an **mzc::cktspec** circuit specification to define the control circuit for the session. • Call **mzz::factory::AllocateSession()** to allocate a new session.	 2-7 2-7
4.	**Create Additional Circuits for the Session** Once a session has been created, call the **ses::AddCircuit()** method to add a new downstream circuit for streaming video.	2-8

Table 2-1: Creating a Session

Note The **mzz::factory::AllocateSession()** and **ses::AddCircuit()** operations in Steps 2 and 3 can be consolidated into a single call to **mzz::factory::AllocateSessionEx()**. See *AllocateSessionEx* on page A-137 for details.

Set the Client Device ID

Every client that participates in an OVS network must have exactly one session allocated. Sessions are keyed by **mzc::clientDeviceId**, which is typed as an octet sequence of arbitrary length. This means that clients can choose an **mzc::clientDeviceId** that works best for them; the only restriction is that the ID has to be unique. Two **mzc::clientDeviceId**s are equal only if they are the same length and have the same contents.

Example This code fragment sets the client device ID to “demo.”

```
mzc_clientDeviceId cid;
id = (char *)"demo";

cid._length = cid._maximum = (ub4) (strlen(id)+1);
cid._buffer = (ub1 *)id;
```

Create a Network Socket to Receive Video

In order for the OVS to transport the video data over the network, the video pump must convert the video stream into a series of **packets**. For streams consisting of data encoded using MPEG-1 System Streams or Oracle Streaming Format (OSF), the video pump divides the video stream into a series of chunks and wraps each chunk with an Oracle Generic Framing (OGF) header. The client uses the information in the OGF headers to reassemble the video data back into a contiguous stream. (Video data encoded using MPEG-2 transport streams does not require OGF headers, since it has similar information already built into the format.)

As illustrated in Figure 2-1, the video pump packages the video data into a series of OGF packets for transport over the network. The client implements a "data reader" that receives the OGF packets from the network, strips off the headers, and passes the video data to the decoder.

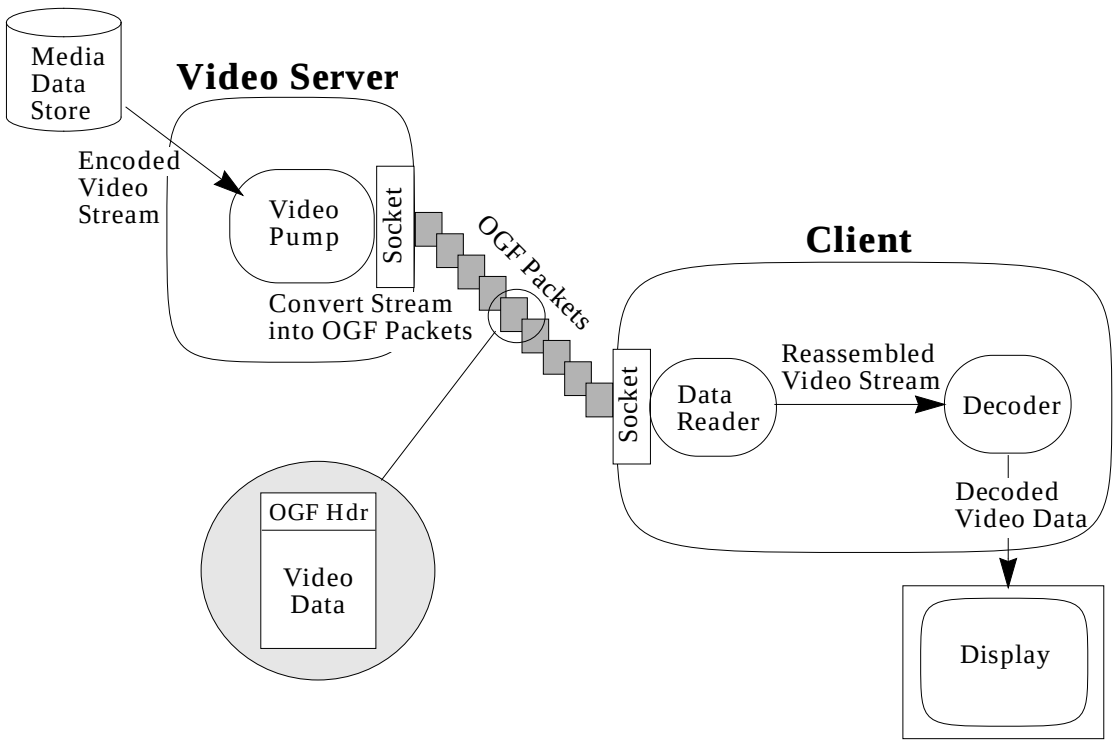


Figure 2-1: Creating, transporting, and reassembling OGF packets

The OGF headers also contain information that can be used by the client data reader to detect network transmission errors, such as dropped or out-of-order packets. You can write your data reader to make use of this OGF information to manage the impact of such transmission errors prior to passing the video data to the decoder.

The OGF header format is described in detail in Appendix F. The purpose of this section is to describe how to set up a socket to receive the OGF data packets from the network. The section, *Read Video Data from the Socket and Decode* on page 2-21, describes how to receive the OGF packets on the socket and use the OGF header information to correctly reassemble the video data for decoding and playback.

Example This code example illustrates how to create a socket to receive video data from the OVS server. The video data in this example can arrive in the form of UDP datagrams or a TCP byte stream.

Note All functions using the **(3N)** designator are specific to the Solaris operating system and defined in the Solaris **/sys/socket.h** file. Though you will be using different functions for other client platforms (such as WinSock for Windows clients), the procedures are similar.

The Solaris **socket(3N)** function creates a socket that recognizes a specific network protocol, such as SOCK_DGRAM for UDP datagrams and SOCK_STREAM for a TCP byte stream. This discussion focuses on setting up a UDP connection. For details on creating and maintaining a TCP connection, see the **ovsdemoSetupDataPort()**, **ovsdemoConnectDataPort()**, and **ovsdemoReadDataPort()** functions in the **ovsdemo.c** sample application listed on page H-2.

```
dproto = (char *)"UDP";
boolean dgram;
int s;

if(strcmp(dproto,"UDP") == 0)
{
  s = socket(PF_INET, SOCK_DGRAM, 0);
  dgram = TRUE;
}
else if(strcmp(dproto,"TCP") == 0)
{
  s = socket(PF_INET, SOCK_STREAM, 0);
  dgram = FALSE;
}
else
  yslPrint( "don't know how to build a %s port\n", dproto);
```

Obtain the address of the client. Use the **ysGetHostName()** function to return the name of the client. Pass this name to the **gethostbyname(3N)** function to return the client address in a **hostent** structure. (The **hostent** structure is defined in the **netdb.h** file.)

```
struct hostent *ent;

ent = gethostbyname(ysGetHostName());
if(!ent)
{
  /* unable to get host's address */
  return -1;
}
```

The **bind(3N)** function assigns the client address and "address family" described in the **sockaddr_in** structure to the socket. (See the Solaris **netinet/in.h** file for a description of the **sockaddr_in** structure.)

```
int sinlen;
struct sockaddr_in sin;

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = ((struct in_addr *)ent->h_addr_list[0])->s_addr;
sin.sin_port = 0;
sinlen = sizeof(sin);

if(bind(s, (struct sockaddr *) &sin, sinlen) == -1)
{
  /* bind failed */
  return -1;
}
```

The **getsockname(3N)** function returns the socket's port id in the **sockaddr_in** structure's 'port' field.

```
sinlen = sizeof(sin);
if(getsockname(s, (struct sockaddr *) &sin, &sinlen) == -1)
{
  /* unable to get socket name */
  return -1;
}
```

The **sprintf()** function constructs the UDP address for the newly created port (this will look something like 127.0.0.1:2856), and stores the address in the *data* variable. The *data* variable is used later to construct the downstream channel, as described in *Create Additional Circuits for the Session* on page 2-8.

```
/* construct a port address for OVS */
sprintf(data,"%s:%d",inet_ntoa(sin.sin_addr),ntohs(sin.sin_port));
```

Allocate a Session and Build a Control Circuit

To allocate a session, the client calls the **AllocateSession()** method. In response to the call, the OVS server creates a session for the client device and sets up the first **circuit** for the device. This first circuit is always a **control circuit**. The client must define an **mzc::cktspec** circuit specification to identify the properties of the control circuit when allocating the session. The **mzc::cktspec** specification is described in detail in *Create Additional Circuits for the Session* on page 2-8.

Example This code fragment illustrates how to create a new session and control circuit.

Bind to the session factory interface and return a reference, *zfac*.

```
mzz_factory zfac;

zfac = (mzz_factory) yoBind(mzz_factory__id,
                            (char *)0,
                            (yoRefData *)0,
                            (char *)0 );
```

Create an **mzc::cktspec** circuit specification to describe a new control circuit for use by the session. The **mzc::cktreq** value of *cktreqTypeSymmetric* identifies the circuit as symmetric. The communication properties (*props*) defined in the **mzc::commProperty** bitmask for this circuit specify that it contains a persistent (*propPersistantConnect*), point-to-point (*propPointcast*) bidirectional channel (*propDown* and *propUp*) between the OVS server and the client device. (No bit rate is defined because it has no meaning for this particular type of circuit.)

```
mzc_commProperty props;
mzc_cktspec spec;

cproto = (char *)"UDP"; /* channel uses UDP protocol */
ctrl = (char *)"127.0.0.1:0"; /* UDP address of client device */
props = mzc_propDown | mzc_propUp | mzc_propPointcast | mzc_propControl |
        mzc_propPersistantConnect;

spec._d = mzc_cktspecTypeRequest;
spec._u.req._d = mzc_cktreqTypeSymmetric;
spec._u.req._u.sym.props = props;
spec._u.req._u.sym.chnl._d = mzc_chnlspecTypeRequest;
spec._u.req._u.sym.chnl._u.req.props = props;
spec._u.req._u.sym.chnl._u.req.protocol.name = (char *)cproto;
spec._u.req._u.sym.chnl._u.req.protocol.info = (char *)ctrl;
spec._u.req._u.sym.chnl._u.req.bitrate = 0;
```

Call the **mzz::factory::AllocateSession()** method to allocate the session.

```
mzz_session ses;

ses = mzz_factory_AllocateSession(zfac, &env, mzz_sessNull, &cid, &spec);
```

Release the **mzz::factory** object.

```
yoRelease((dvoid *)zfac);
```

Create Additional Circuits for the Session

When you call **mzz::factory::AllocateSession()** to create a session, one control circuit is allocated to the session. Clients can create additional circuits for a session by calling **mzz::ses::AddCircuit()**.

A circuit is deleted automatically when the session it belongs to is deleted. A client may delete a circuit explicitly by calling **DelCircuit()**, which removes the circuit from the list of circuits associated with the session. No additional cleanup is needed; any channels associated with the circuit are automatically deleted.

Note The client rarely needs to interact with the circuit itself, but should instead carry out its operations using methods in the **ses** (session) interface. For example, to allocate a circuit, the client usually does not need to call the circuit factory's **mzc::factory::BuildCircuit()** method, but instead calls the **mzz::ses::AddCircuit()** method in the session (**mzz::ses**) interface. The circuit API is mainly used by the server and is provided as a convenience for special cases.

cktspec Circuit Specification

When a client calls **mzz::ses::AddCircuit()**, it must pass an **mzc::cktspec**, which is a discriminated union that specifies the properties of the resources used by the client device. The session service allocates the circuit using the information provided in the **mzc::cktspec**, associates the circuit with the session, and returns the circuit that was allocated to the client.

The possible ways of using an **mzc::cktspec** to specify the properties for the client device are illustrated in Figure 2-2.

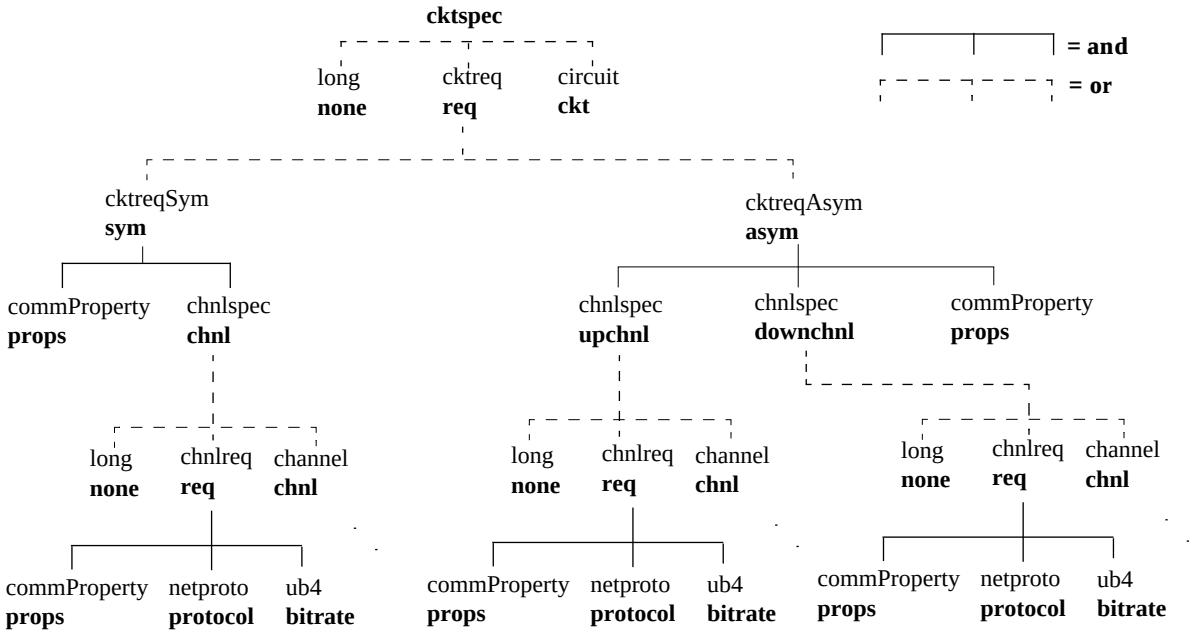


Figure 2-2: Specifying a Circuit

As shown at the top of Figure 2-2, the **mzc::cktspec** can either:

- point to nothing (*none*)
- point to an already existing circuit (**mzc::circuit**)
- contain a request for a new circuit (**mzc::cktreq**)

The new circuit (**mzc::cktreq**) can be symmetric (**mzc::cktreqSym**) or asymmetric (**mzc::cktreqAsym**). A symmetric circuit includes a single channel specification (**mzc::chnlspec**). An asymmetric circuit includes two channel specifications for both an upstream channel (*upchnl*) and a downstream channel (*downchnl*). Both symmetric and asymmetric circuits have specific properties that are defined by setting the **mzc::commProperty** bitmask described on page B-60.

A channel specification (**mzc::chnlspec**) may either be *none*, point to an existing channel (**mzc::channel**), or refer to a new channel (**mzc::chnlreq**). The new channel, in turn, consists of the properties (**mzc::commProperty**) for that channel, information about the protocol used by the channel (**mzc::netproto**), and the *bitrate* for that channel.

Example This code example illustrates how to add a new data circuit to the session, **ses**.

Create an **mzc::cktspec** circuit specification to specify the communication properties of the resource for the client device. All of the possible communication properties are defined by setting the **mzc::commProperty** bitmask.

The communication properties (*props*) defined for the circuit in this example are:

propDown	Channel / circuit is downstream from the video pump to the client device.
propPointcast	Channel / circuit is point-to-point between the video pump and the client device.
propData	Channel / circuit may transport non-isochronous data.
propIsochronousData	Channel / circuit is capable of transporting isochronous data, such as video. An isochronous circuit will have an associated **mzs::stream** reference (which may be NULL, if unbound).
propPersistantConnect	Channel remains connected throughout the life of the circuit.

The circuit specification defines the circuit as asymmetric (*cktreqTypeAsymmetric*) and specifies that it is to operate as a downstream-only (*downchnl*) UDP channel (*dproto*) from the video pump to the client at the *data* address returned for the downstream port (see *Create a Network Socket to Receive Video* on page 2-4).

```
mzc_commProperty props;
mzc_cktspec spec;

bitrate = (char *)"2048000"; /* bit rate is 2 Mbps */
dproto = (char *)"UDP"; /* channel uses UDP protocol */
props = mzc_propDown | mzc_propPointcast | mzc_propData |
          mzc_propIsochronousData | mzc_propPersistantConnect;

spec._d = mzc_cktspecTypeRequest;
spec._u.req._d = mzc_cktreqTypeAsymmetric;
spec._u.req._u.asym.props = props;
spec._u.req._u.asym.upchnl._d = mzc_chnlspecTypeNone;
spec._u.req._u.asym.upchnl._u.none = 0;
spec._u.req._u.asym.downchnl._d = mzc_chnlspecTypeRequest;
spec._u.req._u.asym.downchnl._u.req.props = props;
spec._u.req._u.asym.downchnl._u.req.protocol.name = (char *)dproto;
spec._u.req._u.asym.downchnl._u.req.protocol.info = (char *)data;
spec._u.req._u.asym.downchnl._u.req.bitrate = bitrate;
```

Call the **mzz::ses::AddCircuit()** method to add a downstream circuit for the session, passing it the **mzc::cktspec** circuit specification, *spec*.

```
mzc_circuit dcirc; /* downstream circuit object */

dcirc = mzz_ses_AddCircuit(ses->or, &env, &spec);
```

Query the Content Service

Before requesting a piece of content from the server, you typically have to query the content service to obtain a list of the available content, along with the **mkd::assetCookie**s necessary to identify the content to the **stream** service or the MDS service.

If, for example, you are using the **vscontsrv** content service, you can obtain the content information for named content, along with its respective **mkd::assetCookie**s, by calling the **mza::LgCtntMgmt::lstAtrByNm()** method. The content information is returned in one or more **mza::LgCtntAtr** structures (depending on whether wildcards are specified in the name). The **mza::LgCtntAtr** structure returned contains a tightly-bound object reference to the **mza::LgCtnt** object matching the name, which you need to invoke any of the methods listed under *mza::LgCtnt* on page A-54.

Some content services may also return an object reference to an authorization object to be used to authenticate the client, as described in *Authorize the Client* on page 5-12. The **vscontsrv** content service, however, does not return an authorization object. The **mza::LgCtnt** interface, and all related interfaces, are described in detail in Chapter 3, Manipulating Video Content.

Example This code example illustrates how to query the logical content service to obtain the asset cookie and segment information for the requested *file*.

Obtain a loosely bound object reference to the logical content manager object, **mza::LgCtntMgmt**.

```
mza_LgCtntMgmt  fobj;     /* content service obj ref */

fobj = (mza_LgCtntMgmt) yoBind( mza_LgCtntMgmt__id, (char *)0,
                                    (yoRefData *)0, (char *)0);
```

Call the **mza::LgCtntMgmt::lstAtrByNm()** method to obtain information on the specified file. In order to support wildcards in names, the **mza::LgCtntMgmt::lstAtrByNm()** method returns a sequence of **mza::LgCtntAtr** structures in an **mza::LgCtntAtrLst**. In this example, a single piece of logical content, named *myvideo*, is specified, so only one **mza::LgCtntAtr** structure is returned in the sequence.

You use the **mza::Itr** structure to keep track of the current position in the **mza::LgCtntAtrLst**. If you were using wildcards to describe the *file* (**video*, for example), then you would probably set the **itr.NumItems** to a value greater than 1. (See *Using an Iterator* on page 3-12 for more information on how to use the **mza::Itr** structure.)

```
mza_LgCtntAtrLst listings;
const char *file = "myvideo";
boolean tagfile = FALSE;
mza_Itr itr;

itr.Position = 0;
itr.NumItems = 1;

listings = mza_LgCtntMgmt_lstAtrByNm( fobj, env, (char *)file, tagfile,
                       &itr );
```

Obtain the **mkd::assetCookie** for the logical content from the first (and only) **mza::LgCtntAtr** in the returned **mza::LgCtntAtrLst**.

```
*cookie = (mkd_assetCookie) ysStrDup( listings._buffer[0].cookie );
```

The **mza::LgCtntAtr** structure contains an **mza::CtntAtrLst**, which is a sequence of **mza::CtntAtr** structures containing the bit rate and other content information for each **mza::Ctnt** object used by this logical content. Check the **mza::CtntAtrLst** for the encoding rate of each **mza::Ctnt** object to determine the maximum bit rate among the clips.

```
int i;
ub4 bitrate = 0;

for(i = 0; i < listings._buffer[0].ctntAtrLst._length; ++i)
{
   if(listings._buffer[0].ctntAtrLst._buffer[i].rate > bitrate)
           bitrate = listings._buffer[0].ctntAtrLst._buffer[i].rate;
}
```

Free the resources used by the **mza::LgCtntAtrLst**.

```
mza_LgCtntAtrLst__free( &listings, yoFree );
```

Prepare and Stream Video Using the Stream Service

Once a client has established a session with the Oracle Video Server, it can use the **stream** service to initiate and control streams from the server.

The **mzs::stream** interface expects the client to provide an asset cookie that identifies the server content to be streamed. The client obtains the asset cookie from a content service, as described in *Query the Content Service* on page 2-11. If the client is not authorized to access the requested services, the stream service returns an error.

Summary of Operations

To set up a stream object and use it to send content, perform these actions:

Step	Action	Details on Page
1.	**Allocate a Stream** context by calling the **mzs::factory::alloc()** method.	2-14
2.	**Prepare Video Content**: Call **mzs::stream::prepare()** or **mzs::stream::prepareSequence()**, passing an **mkd::assetCookie**, to prepare a stream to play a particular piece of content.	2-15
3.	**Play Video Content**: Call **mzs::stream::play()** to play the video content. Call this method several times if you want to change the rate at which the content is playing or the portion of the content that's being sent.	2-17
4.	**Read Video Data from the Socket and Decode**, extract the data from the OGF packets, and forward to the decoder.	2-21
5.	**Finish With Video Content**: When you no longer need the piece of content, call **mzs::stream::finish()**. This method cleans up any cached information at the server. Also, if the content is currently playing, **finish()** stops playback and sends any necessary end-of-stream information to the client.	2-24

Table 2-2: Setting Up and Using a Stream

Step	Action	Details on Page
6.	To play another piece of content, query the content service again and go back to **Step 2**. Otherwise, continue with **Step 7**.	
7.	**Deallocate the Stream**: Call **mzs::stream::dealloc()** to destroy the stream object and free all resources created by **alloc()**.	2-24

Table 2-2: Setting Up and Using a Stream

Allocate a Stream

The **mzs::factory::alloc()** method allocates a stream object for sending content to a client. You must allocate a stream object before you can stream video data to a client.

You can use the same stream to send several different pieces of content, one after another, so you need only to allocate one stream per session. If your client can handle several simultaneous streams of data, you can allocate several streams, calling **mzs::factory::alloc()** once for each. In that case, make sure that the total bandwidth of all streams allocated does not exceed the capabilities of the client or the network.

A given network might support a variety of clients with widely different capabilities. When you allocate a stream, you must indicate the playback capabilities of your client device to the **stream** service. You do this by passing an **mzs::capMask** to the **mzs::factory::alloc()** method. This mask has a flag for each of the client capabilities the server might need to be aware of. The **mzs::capMask** flags are listed in *mzs::capMask* on page B-69.

Example This code example demonstrates how to allocate a stream to play MPEG-1 and MPEG-2 video.

Obtain an object reference to the stream **factory** object.

```
mzs_factory sfac;

sfac = (mzs_factory) yoBind( mzs_factory__id, (char *)0, (yoRefData *)0,
       (char *)0 );
```

Set up the **mzs::capMask** to indicate that the client can receive both MPEG-1 and MPEG-2 video and can pause and blindly seek but not scan (rate control).

```
mzs_capMask caps;

caps = mzs_capAudio | mzs_capVideo | mzs_capMpeg1 | mzs_capMpeg2 |
                      mzs_capSeek | mzs_capPause;
```

Call **mzs::factory::alloc()** to allocate a stream and associate it with the down stream circuit (*dcirc*) created for this session by the **mzz::ses::AddCircuit()** method described in *Create Additional Circuits for the Session* on page 2-8.

```
mzs_stream strm;
ub4   bitrate;
/* dcirc is the object reference to the down stream circuit */

/* maximum bitrate that the client can support */
bitrate = (ub4) 2048000;                    /* 2 Mbps */

strm = mzs_factory_alloc( sfac, env, dcirc, caps, bitrate );
```

Release the stream (mzs) **factory** object (no longer needed).

```
yoRelease((dvoid *)sfac);
```

Prepare Video Content

Once you have obtained the **mkd::assetCookie**s for the selected content and an object reference to a stream object, call the **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** method to prepare the selected content to be streamed to the client (the difference between the two is discussed in this section).

The contents of the **mkd::assetCookie**s returned by the content service is of interest only to the **stream** service; the client simply passes the cookies associated with the desired content to the **prepare()** or **prepareSequence()** method. The **stream** service then prepares the content by locating the appropriate **content resolver** to resolve the **mkd::assetCookie**s into physical content and, if an authentication object is specified, determine whether the client is authorized to play the content. If the client is authorized to play the content, the content resolver returns to the **stream** service one or more records, including file names, start times, and stop times. If an authentication object is specified, the **stream** service will not deliver any of the requested content unless a successful authentication is received.

The **prepare()** method prepares a single piece of content, whereas the **prepareSequence()** method prepares several pieces of content in sequence as a single stream. The **prepareSequence()** method takes an array of

mkd::assetCookies, each of which describes one or more segments of content to be played. When the combined sequence is played, the server plays the segments as one continuous piece of content, without pauses or glitches between segments.

The **prepare()** and **prepareSequence()** methods return immediately, even if it will take time to prepare the content. The prepare methods return an **mzs::instance** that identifies the stream of prepared content assets. The returned stream instance must be used to identify the stream in any subsequent calls to the stream server.

When you prepare the stream, you can set the **playNow** flag to direct the stream server to play the content as soon as it's prepared, or you can explicitly play the stream by calling the **mzs::stream::play()** method. You can call **play()** at any time after **prepare()** or **prepareSequence()** returns; if the content is not yet ready, **play()** may block until it's available. Once a stream has been prepared, you can play it on the client device any number of times without any further preparation.

Example This code example demonstrates how to call the **mzs::stream::prepare()** method to prepare a specific piece of content to be streamed to the client. The content is identified by the *cookie* obtained from the content service using the **mza::LgCtntMgmt::lstAtrByNm()** method described in *Query the Content Service* on page 2-11:

- The *asset_bitrate* parameter, which was also returned by the **mza::LgCtntMgmt::lstAtrByNm()** method, indicates the rate at which the stream is to be played.
- Specifying **mkdBeginning** and **mkdEnd** indicates that the stream is to play from beginning to end.
- Setting the **playNow** flag causes the movie to start immediately, so no **play()** method is necessary.
- The **prepare()** method returns an object reference, *inst*, to the newly created stream instance.

```
mzs_stream_instance inst;
mkd_segInfoList status;

inst = mzs_stream_prepare(strm, &env, cookie,
                          (mkd_pos *)&mkdBeginning,
                          (mkd_pos *)&mkdEnd,
                           asset_bitrate,
                           mzs_stream_playNow,
                           &status,
                          (dvoid *)0);
```

The stream service provides a looping feature that allows you to repeatedly play a portion of content, known as a **segment**, or a sequence of segments as a loop. For example, you might wish to display a video logo in between showings of an asset. Since a video logo consists of a limited number of frames, it is best implemented as a loop.

The stream service provides three flags that are related to looping:

- **playLoop**
- **playLoopLast**
- **finishLoop**

The **playLoop** and **playLoopLast** flags are used by the **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** method when the asset is prepared. The **playLoop** flag is used to loop an entire sequence (or a single segment); **playLoopLast** is used to loop the last segment in a sequence. (The **playLoopLast** flag is currently not supported.)

The **finishLoop** flag is passed to the **mzs::stream::finish()** method to cleanly terminate the loop.

Play Video Content

The **mzs::stream::play()** method is used to instruct the stream service to start playing a prepared piece of content. It's also used to change the playback characteristics of a stream currently being played. For example, you can call **play()** to pause playback, to fast-forward, or to rewind.

When you call **play()**, you can specify the beginning and / or ending locations by passing *startPos* and *endPos* arguments. This is discussed in *Play Positions: startPos, endPos, and curPos* on page 2-18.

There are a few common ways to use the **play()** method. For example, you may make frequent calls to pause playback or play at normal speed from the current position.

Note You must synchronize your **play()** operation with the data reader / decoder operations that receive the video data from the network and decode it for display. See *Read Video Data from the Socket and Decode* on page 2-21 for details.

Several play helper methods listed in **Table 2-3** are provided for common tasks.

Play Helper Method	Description
pause()	Pauses the specified content's playback.
playFwd()	Plays the specified content forward, starting at the current position.
playRev()	Plays the specified content backwards, starting at the current position. (This method is currently not supported.)
frameFwd()	Advances the content one frame from the current position. (This method is currently not supported.)
frameRev()	Rewinds the content one frame from the current position. (This method is currently not supported.)

Table 2-3: play() Helper Methods

Play Positions: startPos, endPos, and curPos

The **play()** method has parameters that allow you to specify the start and end position for play. If you call **play()** with a playback rate other than zero (pause), the *startPos* parameter specifies the starting point and *endPos* the ending point. The *curPos* parameter is a hint to the stream server about the current position, which might be different from *startPos* if, for example, the client device is seeking or buffering data.

You specify play positions by passing pointers to **mkd::pos** structures. The way you use these structures depends on whether you are calling **play()** to pause playback, or to start (or change) playback.

Pausing Playback

If your client device is capable of reporting a precise playback position, pause playback by constructing an **mkd::pos** structure to encode the precise playback position where you want to pause and then pass a pointer to this structure as the *curPos* argument of the **pause()** method. This tells the server precisely where in the stream you want to pause and, presumably, where you'll want to resume.

If your client device cannot report a precise playback position (typical in a corporate LAN environment), you can stop display output, and specify **&mkdCurrent** for *startPos* and *curPos*. This refers to the server's output position in the stream when it receives the **pause()** request. Because of network latencies, you must be able to buffer data that the server sends between the time you issue the **pause()** request and the time it's received by the server.

Example You can use either the **pause()** or **play()** method to pause a stream.

The **pause()** method simply specifies the stream instance, *inst*, and the current position, *&mkdCurrent*.

```
mzs_stream_pause (strm, &env, inst, (mkd_pos *)&mkdCurrent);
```

To pause the stream using the **play()** method, specify the stream and the current position for *curPos*, *startPos*, and *endPos*.

```
mzs_stream_play(strm, env, inst,
                (mkd_pos *)&mkdCurrent,
                (mkd_pos *)&mkdCurrent,
                (mkd_pos *)&mkdCurrent,
                 mzs_stream_ratePause,
                 asset_bitrate);
```

Starting or Resuming Playback

If playback is paused when you call **mzs::stream::play()**, and you pass **&mkdCurrent** for both *startPos* and *curPos*, the server resumes output at the position it stopped after receiving the **pause()** request.

Because of network latencies, **&mkdCurrent** is not necessarily the same point at which the content was last displayed to the viewer. You must therefore have buffered the data the server sent between the time you stopped display output and the time the server stopped its output to you; this allows you to resume display output without glitches or dropouts when the server resumes output.

Example To resume playback of the stream, specify the **play()** method with **&mkdCurrent** for both *startPos* and *curPos*.

```
mzs_stream_play(strm, env, inst,
               (mkd_pos *)&mkdCurrent,
               (mkd_pos *)&mkdCurrent,
               (mkd_pos *)&mkdEnd,
                mzs_stream_rate1x,
                bitrate );
```

Rate and Direction Control

The **play()** method allows you to specify the replay rate by passing an integer in the *playRate* parameter. A positive value indicates forward playback and a negative value indicates reverse playback. The base number is 1,000 for normal replay rate. So a rate of 2,000 means "play forward at twice normal speed;" similarly, a rate of -1,000 means "play in reverse, at normal speed."

A rate of zero means "pause playback." If you later resume playback without specifying a start position, the stream server will resume playing from the position at which it was paused. The **play()** method's *curPos* parameter is used to help the stream server cache data; however, it will *not* change the position from which the stream server starts playing.

Get the Current Position in a Stream

Use the **mzs::stream::getPos()** method to determine the server's last recorded playback position in a piece of content or a sequence of content segments. The **getPos()** method passes back an **mkd::pos** value that specifies the server's output position within the most recently prepared piece of content or sequence at the time the server receives the **getPos()** request. Positions within a sequence are given from the beginning of the entire sequence, rather than from the beginning of the current segment.

Because of network latencies, the position returned by **getPos()** is not necessarily the same point at which the content was last displayed to the client. There is also some latency in the processing of **getPos()**, which may make the value less than exact. The current position of a playing stream may indicate the stream has stopped when video is still being displayed to the client. Such latencies are entirely dependent on the network configuration.

You would generally only want to call **getPos()** on a playing stream. A typical application is a "bookmarker." Every five minutes, it asks the server where it is in the playing content. The application stores this answer somewhere (probably a database), so if it's disconnected for some reason, it can pick up within a few minutes of where it stopped. That is why **getPos()** returns something like, "You are 5 minutes, 3 and 32/100 seconds into the content."

Note that only the client device knows the true position of a playing stream. Due to network latencies, the **getPos()** method can only return the position last delivered by the video pump.

Suppose you prepared a sequence composed of 10 minutes of a movie, 2 minutes of car commercials, the next 10 minutes of the movie, 2 minutes of soap commercials, and the final 10 minutes of the movie. If you call **getPos()** after the sequence has been playing for 23 minutes, you are told you are 23 minutes in, not that you are 1 minute into the soap commercials.

Read Video Data from the Socket and Decode

When you play the video data, the data reader in your client must reassemble the OGF packets received from the OVS server back into a stream. See *Create a Network Socket to Receive Video* on page 2-4 for an overview of this process.

If the socket is set up to receive TCP byte stream data, you can use the information in the OGF header to extract the packets from the byte stream. You do this by reading the byte stream, searching for the magic number (F0F0F003) to locate the beginning of an OGF packet, and using the OGF data offset (*DP*) and data size (*Size*) fields to locate the packet's video data. For an example, see the **ovsdemoReadDataPort()** function in the **ovsdemo.c** sample application listed on page H-2.

Example If the socket is set up to receive UDP data, use the Solaris **recvfrom(3N)** function to receive each packet from the socket. Read the OGF magic number to confirm that this is an OGF packet, and read the OGF Size (*sz*) field to determine the length of the data in the packet.

```
int  rlen, l, len, sinlen;
struct sockaddr_in sin;
ub1  buf[8216], *p, uc;
ub4  sz;

sinlen = sizeof(sin);
l = recvfrom(port,(char *)buf,sizeof(buf),0,(struct sockaddr *)&sin,
             &sinlen);
if(l == -1)
{
  /* error */
}
if(l < 20)
{
  /* short packet */
}
if(sysxGetB4(buf) != 0xf0f0f003)
{
  /* unknown packet */
}
sz = sysxGetB4(buf+4);
```

Read the remaining fields in the OGF header.

```
ub2  fseq, cseq, cnum;
ub1  rev, offset;
ub4  type;

fseq = sysxGetB2(buf+8);       /* Frame sequence number*/
rev  = buf[10];                /* Revision level of frame */
offset = buf[11];              /* Start of data offset */
type = sysxGetB4(buf+12);      /* Type of data*/
cseq = sysxGetB2(buf+16);      /* Current sequence value*/
cnum = sysxGetB2(buf+18);      /* Discontinuity indicator*/
```

See *Detecting Dropped OGF Packets and other Content Discontinuities* on page 2-22 for information on how to use the Frame Sequence (*FSeq*), Current Sequence (*CSeq*), and Command Number (*CNum*) fields.

After reading the contents of each OGF header, extract the video data (the contents of the packet following the *offset* for the length of *l*) and forward the data to the decoder.

Repeat the steps in this example for each OGF packet received.

Detecting Dropped OGF Packets and other Content Discontinuities

The Frame Sequence (*FSeq*), Current Sequence (*CSeq*), and Command Number (*CNum*) fields in the OGF header can be used to detect dropped or out-of-order OGF packets, as well as "discontinuities" in the content originating in the server.

Using the Frame Sequence Number

The OVS video pump increments the frame sequence number (*FSeq*) for each new OGF packet sent. Your client can use this value to detect packets that have been dropped or have arrived out of order.

Example In this example, the *fs* variable was set to 1 prior to reading any packets. The *fs* and *fseq* values should be the same when reading the first packet and all the following packets, unless a packet arrives out of order or is missing. How you deal with dropped and reordered packets is up to you and the requirements of your decoder.

```
ub2 fs

if(fseq != fs)
{
  yslPrint("Frame drop: expected %d received %d.\n",*fs,fseq);
}
fs = fseq + 1;
```

Using the Current Sequence Number

The current sequence (*CSeq*) value is reset to the current *FSeq* value each time a "discontinuity" in the content delivery occurs. Such discontinuities can result from the server's response to a client command to seek to a different location in the content, or from a transition between the sequences that make up the individual clips in a piece of logical content.

Example This example builds on the previous example and uses the *CSeq* value to determine if a dropped packet is within 5 packets following a discontinuity in the content.

```
ub2 fs

if(fseq != fs)
{
  yslPrint("Packet drop: expected %d received %d.\n",fs,fseq);
  if(fs < cseq + 5)
     yslPrint("Warning: Drop within first 5 packets of command.\n");
}
fs = fseq + 1;
```

Using the Command Number

The command number (*CNum*) field is similar to the *CSeq* field in that its purpose is to also detect discontinuities in the content. The difference is that the *CNum* field is incremented by 1 following each discontinuity. You can use the *CNum* field to determine if the OGF packet contains the video data sent by the server in response to a previous seek command issued by the client.

The *CNum* value can also be used to detect if a short sequence of packets was dropped and not identified by the *CSeq* field. See the description for the *CNum* field in Appendix F for details.

Example In this example, any difference detected between the *cn* and *cnum* values identifies an OGF packet sent by the server in response to a previous client command.

```
ub2 cn

if(cnum != cn)
{
  yslPrint("Command response received.\n");
  cn = cnum;
}
```

Note The *CNum* value is not guaranteed to identify data sent in response to commands. This is because the *CNum* value may also be incremented by the server in response to a sequence transition or other discontinuity in the content. See Appendix F for details.

Finish With Video Content

Once you have finished using a piece of content, indicate this by calling **mzs::stream::finish()**. This method is the counterpart of **mzs::stream::prepare()** and **prepareSequence()**; it "unprepares" the content, freeing any system resources the stream server might have used. Once you have finished a stream, you won't be able to play any more content unless you call **prepare()** again. It is not necessary to call **finish()** explicitly before calling **prepare()**; an implicit finish occurs when **prepare()** is called.

If you call **finish()** on a piece of content that is currently playing, the content is stopped immediately. If there is content still waiting to be played, it's discarded. You can also use **finish()** with the **finishOne** flag to remove the prepared content or with the **finishLoop** flag to play the stream straight through and end instead of looping. (The stream service also provides a **finishAll** flag to remove multiple outstanding prepared content. However, there is currently no support for preparing multiple content, so the **finishOne** and **finishAll** flags are effectively equal.)

Note that when you call the **mzs::stream::dealloc()** method to deallocate a stream, any content prepared for that stream will automatically be "finished," as if you had called the **finish()** method for the stream.

Example The **finish()** method shuts down the stream. The **finishOne** flag specifies that only the currently prepared content is to be finished.

```
mzs_stream_finish(strm, &env, inst, mzs_stream_finishOne);
```

Deallocate the Stream

When you no longer need a stream, deallocate it by calling **mzs::stream::dealloc()**. This method deallocates any resources allocated by the **stream** service for this stream. It also "unprepares" any prepared content for this stream in the same manner as if you had called **mzs::stream::finish()** for each piece of content.

Example Use the **dealloc()** method to deallocate the stream.

```
mzs_stream_dealloc(strm, &env);
```

Delete the Session

When a session is no longer needed, use the **mzz::ses::Release()** method to remove the session and release any associated circuits. This method also releases any channels associated with these circuits and frees any resources associated with the channel. You must also use **__free()** functions to free the resources used by any complex datatypes.

Example When you no longer need the session, use the **mzz::ses::Release()** method to remove the session. Free the **mzz::session** and **mzc::circuit** structures by invoking **__free()** functions, specifying **yoFree()** to free the contents of the structures. (See *Releasing Complex Datatypes* on page 1-14 for more information.)

```
mzz_ses_Release(ses.or, &env);

mzz_session__free( &ses, yoFree );
mzc_circuit__free( &dcirc, yoFree );
```

CHAPTER

3

Manipulating Video Content

The logical content service, **vscontsrv**, provides you with an abstract view of the content data stored in the Media Data Store (MDS). The **vscontsrv** service does not work with physical content directly, but with **metadata** that describes the physical content.

Physical content is stored in the MDS from beginning to end as content files. The metadata that describes the physical content is stored in MDS as tag files and in a database, if present. The **vscontsrv** service allows you to use the metadata in the database to create abstract units from the MDS content, called **clips**, and organize multiple clips into units of logical content.

For example, a piece of physical content might be a series of 20 cartoons. Five cartoons are needed for a cartoon program, along with a program title and a promo advertising a future cartoon program. With the logical content service, you can create a new logical content object for this program, select the cartoons, title and promotions, and add them to the newly created logical content object. You can then use the logical content object to represent the entire cartoon program, complete with promotions, titles, and cartoons, as a single piece of content.

Oracle Video Server Manager (VSM) contains screens that allow users to manipulate logical content via a standard Java client. Most logical content operations can be performed using VSM, as described in *Getting Started with Oracle Video Server Manager*.

This chapter describes how you might use the logical content interfaces to manipulate the OVS content from an application program. The information provided here is also helpful should you need to build your own content service, as described in Chapter 5, "Writing a Custom Content Service".

The **vscontsrv** operations described in this chapter include:

- Create Content
- List Content
- Search for Named Content
- Locating Clips by Name
- Create New Logical Content
- Add Clips to Logical Content
- Remove Clips from Logical Content
- Destroy Logical Content

For a complete list of logical content methods, see *mza — Logical Content Interfaces* on page A-54.

Logical Content Model

The OVS Logical Content model is supported by four main object types:

- Logical Content (**LgCtnt**)
- Clip (**Clip**)
- Content (**Ctnt**)
- Content Provider (**CtntPvdr**)

Each object type is responsible for tracking unique information. Figure 3-1 shows how the objects that make up the logical content model interact with each other.

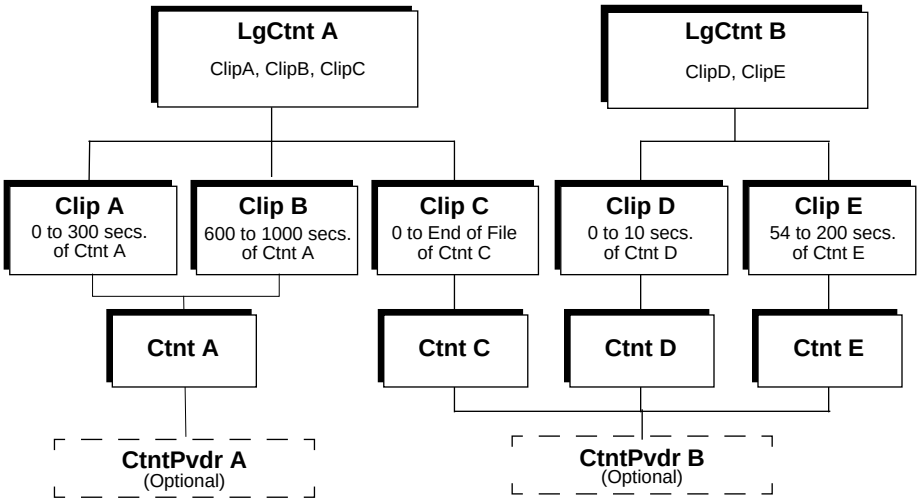


Figure 3-1: OVS Logical Content Relationships

Each object type has three interface types:

- **Base** interfaces are the main interface type for operations implemented by the "base objects," **mza::CtntPvdr**, **mza::Ctnt**, **mza::Clip**, and **mza::LgCtnt.** The methods in this category include operations on object attributes and destruction of objects.
- **Factory** interfaces are used exclusively for creating new objects.
- **Management** methods are used for query operations and returning information about groups of objects.

CtntPvdr Interface

An **mza::CtntPvdr** object is used to assign ownership to a piece of content. The **CtntPvdr** interfaces are **mza::CtntPvdr**, **mza::CtntPvdrFac**, and **mza::CtntPvdrMgmt**. **CtntPvdr** information (if used) is included in the **mza::Ctnt** object.

Returning to the cartoon example, the cartoons have been purchased from **StarPoint**, and royalties are due each time a cartoon is shown. Titles are produced by **Graphiks**, commercials are provided by **Toy City**, and promotional spots are produced in-house. By assigning an **mza::CtntPvdr** to each piece of content, you have the ability to track the royalty payments to be paid to StarPoint and commercial income to be collected from Toy City.

Ctnt Interface

An **mza::Ctnt** object is an abstraction of a piece of MDS content. Each **mza::Ctnt** object contains the metadata for a piece of content, such as encoding rate, encoding format, the MDS tag file associated with the content file, run length of content, and size of the content file. This information is the same information stored in the MDS tag file. **Ctnt** objects are created during the tagging process, as described in *How Content Information is Stored* on page 3-7.

The **Ctnt** interfaces are **mza::Ctnt**, **mza::CtntFac**, and **mza::CtntMgmt**.

Figure 3-2 illustrates the **mza::CtntPvdr** and **mza::Ctnt** objects in our cartoon example. Four **mza::Ctnt** objects are used for the *Titles*, *Commercial*, *Promos*, and *Binky Bopper Cartoons*, and each **mza::Ctnt** object is associated with an **mza::CtntPvdr** object.

Note In this example, we chose to minimize the number of **mza::Ctnt** objects in the system by organizing all of the titles into a single **mza::Ctnt** object, all the commercials into another **mza::Ctnt** object, and so on. This organization is completely arbitrary. We could have just as easily organized each title, commercial, promo, and cartoon into an individual **mza::Ctnt** object.

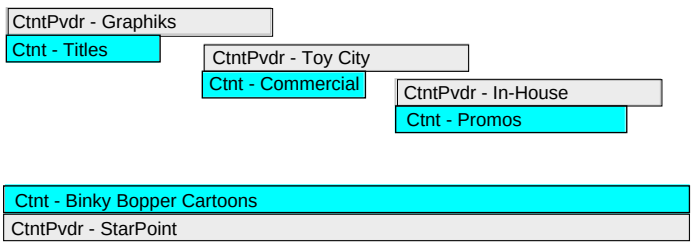


Figure 3-2: Ctnt and CtntPvdr

Clip Interface

An **mza::Clip** object references an **mza::Ctnt** object and adds start and stop positions that allow you to select a portion of the content or the entire content. If, for example, you were selecting 2 portions from a piece of content, you would use 2 clips. Suppose the complete content represented by an **mza::Ctnt** object runs from 0 to 2000 seconds. Clip 1 could represent the portion of content from positions from 25 to 300, and Clip 2 could represent positions 1080 to 2000 seconds.

The **Clip** interfaces are **mza::Clip**, **mza::ClipFac**, and **mza::ClipMgmt**.

As illustrated in Figure 3-3, the cartoon show described in this chapter is made up of the following **mza::Clip** objects:

- 5 clips from the **Ctnt**, *Binky Bopper Cartoons*.
- 1 clip from the **Ctnt**, *Titles*.
- 2 clips from the **Ctnt**, *Commercial*.
- 2 clips from the **Ctnt**, *Promos*.

Note An **mza::Clip** object can be added to an **mza::LgCtnt** object as many times as you wish. This would allow you to repeat the commercial and promo clips multiple times throughout the program.

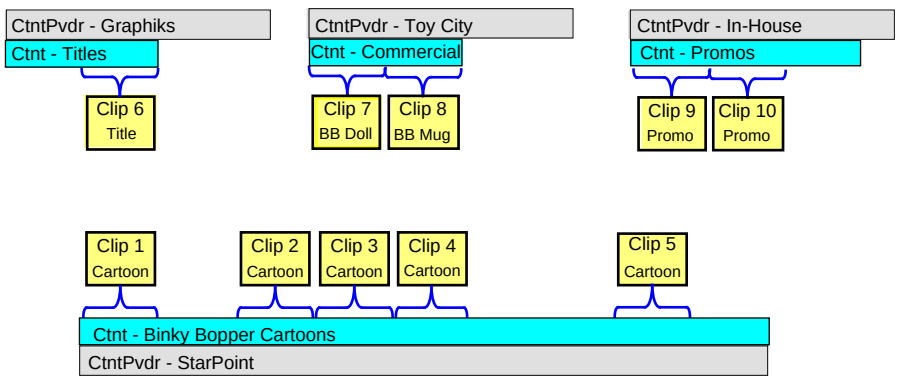


Figure 3-3: Clips

LgCtnt Interface

A **LgCtnt** object combines one or more **mza::Clip** objects into a piece of logical content that can be identified by a unique name and streamed to clients as if it were a single piece of content.

When you first create the **LgCtnt**, it will have zero or one clips, depending on the **mza::LgCtntFac** method you use. Once you've created a **LgCtnt** object, you can add as many clips as you like to the **LgCtnt** and organize them in any sequence.

The **LgCtnt** interfaces are **mza::LgCtnt**, **mza::LgCtntFac** and **mza::LgCtntMgmt**.

Continuing with the cartoon program example, 10 clips are logically assembled in the **LgCtnt** *Binky Bopper Cartoon Show*. When a client queries the **vscontsrv** service for information on the *Binky Bopper Cartoon Show*, **vscontsrv** returns an **asset cookie** that serves as a handle to the **mza::LgCtnt** object. The process of obtaining and using an asset cookie is described in *Content Service* on page 1-6.

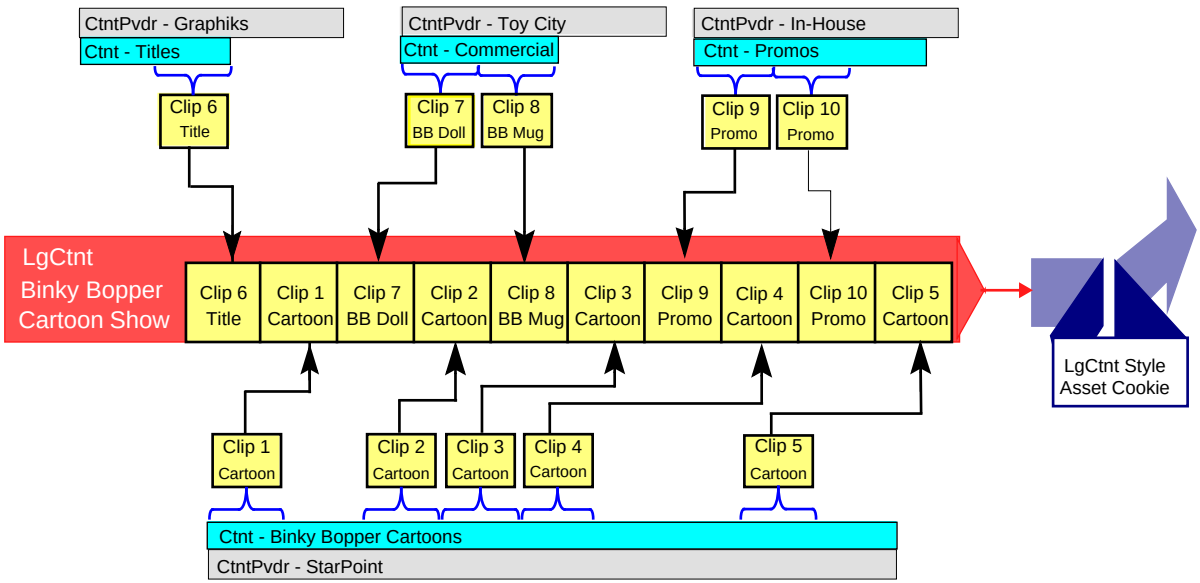


Figure 3-4: Logical Content

How Content Information is Stored

Content **metadata** is the information about a piece of content that is needed by the stream service in order to stream it to a client. The metadata for a piece of content is initially created by a **tagger** utility. This metadata consists of both **header** and **tag** information. Header information describes the compression format of the content, as well as other information, such as the dimensions of the video picture and playback frame rate. Tag information describes specific frames in the content for use in visual fast-forward and other operations that involve locating specific positions within the content.

Figure 3-5 illustrates how a tagger generates and stores the metadata for a piece of content. The tagger sends the header and tag information to a **tag file** in the MDS. The tagger also sends the header information to the logical content service, **vscontsrv**, which stores the header information in the database (if present) and generates new **mza::LgCtnt**, **mza::Clip**, and **mza::Ctnt** objects to represent the content. Note that **vscontsrv** does not store tag information.

For more information on content metadata and the tagging process, see Chapter 7, "Extending Video Encoders for Real-time Feeds". The rest of this section describes how the **vscontsrv** operates with and without a database.

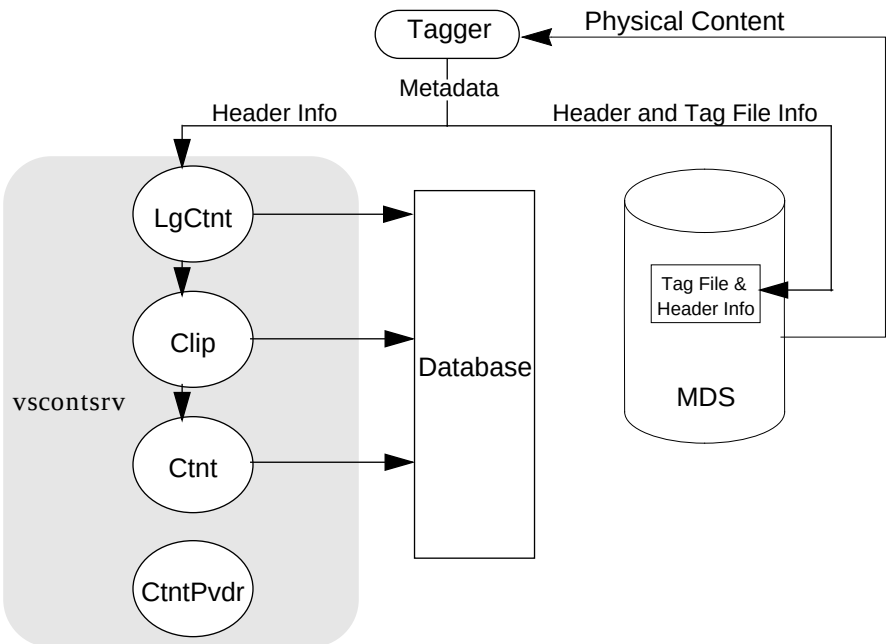


Figure 3-5: Metadata Flow From the Tagger

Logical Content Services with a Database

The logical content data model described in *Logical Content Model* on page 3-2 requires a database to store the metadata for your **mza::Clip** and **mza::LgCtnt** objects. When the tagger first creates an **mza::LgCtnt** object for a piece of content, the **mza::LgCtnt** object contains a single **mza::Clip** object that references the entire **mza::Ctnt** object for the newly tagged content.

Once the **mza::LgCtnt** object is created, you can create more **mza::Clip** objects that reference all or portions of **mza::Ctnt** objects. You can then add these **mza::Clip** objects to the **mza::LgCtnt** in any sequence.

As illustrated in Figure 3-6, clients that query the **mza::LgCtnt** interface for content receive back a logical-content-style **asset cookie** that encapsulates the information needed by the stream service to stream the logical content to the client.

See the discussion on implementing logical content in the *Oracle Video Server Administrator's Guide and Command Reference* for information on how to run OVS with a database.

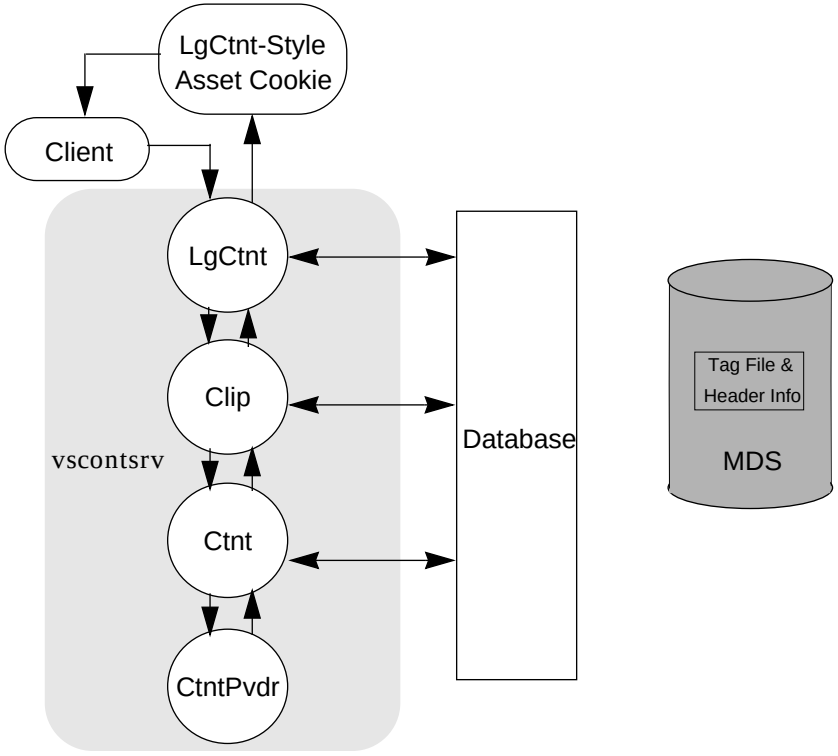


Figure 3-6: Creating Logical Content

Logical Content Services without a Database – Stand-alone Mode

The logical content service can be invoked in **stand-alone mode**, which means it is operating without a database. In stand-alone mode, the information will have the same "look," but the content is not structured using the logical content data model described in this chapter. When invoked without a database, the **mza::LgCtnt** object obtains content data directly from MDS tag file headers. This means there are no clips, no content providers and no real logical content. Figure 3-7 shows logical content in stand-alone mode. Clients that query the **mza::LgCtnt** interface for content receive back an MDS-style **asset cookie** that represents a single tag file.

In stand-alone mode, only the **mza::LgCtntMgmt** and **mza::CtntMgmt** **lstAtr** methods perform any actions. All other methods in the **mza::LgCtnt**, **mza::Clip**, **mza::Ctnt**, and **mza::CtntPvdr** interfaces do not perform any actions.

See the discussion on implementing logical content in the *Oracle Video Server Administrator's Guide and Command Reference* for information on how to run OVS with a database.

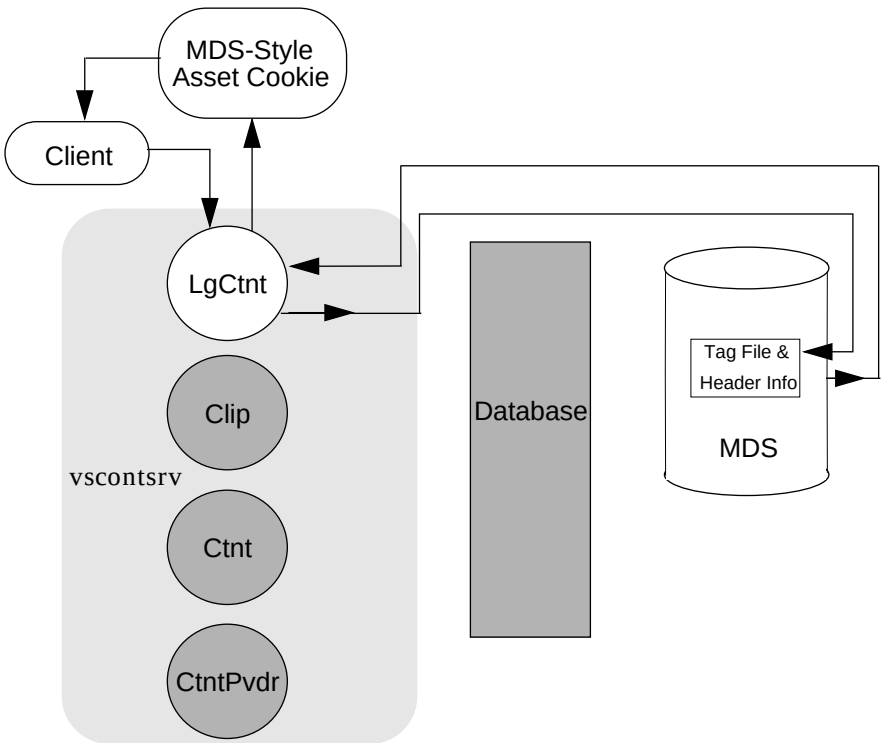


Figure 3-7: Operation in Stand-alone Mode

There are specific differences in the results returned by the **mza::LgCtntMgmt** and **mza::CtntMgmt lstAtr** methods when operating without a database:

- The **mza::LgCtntMgmt::lstAtrByNm()** method must specify the MDS-style filename as the name of the **mza::LgCtnt** object. The **mza::LgCtntMgmt::lstAtr()** method uses a filename pattern of **/mds/*/*.mpi** to find all tag files.
- The object name returned in the **mza::LgCtntAtr** structure and respective **mza::ClipAtr** and **mza::CtntAtr** structures is the name of the MDS tag file from which the objects were created.
- The single **mza::Clip** in the **mza::LgCtnt** references the entire tag file, using **mkd_Beginning** and **mkd_End** for the start and stop position attributes of the clip.
- The asset cookies returned in the **mza::LgCtntAtr** are MDS-style asset cookies that are resolved directly by the stream service.

Using Logical Content Interfaces

This section describes how to make use of the basic methods provided by the interfaces to the **vscontsrv** service. Many of the operations described here can be performed using the Oracle Video Client (OVC). OVC users should extend applications using the Oracle Video Custom Control and Web Plug-in features described in the *Oracle Video Client Developer's Guide*.

Important Do not create logical content consisting of clips encoded at different bit rates or for different codecs. The Oracle Video Client cannot decode logical content consisting of this type. See the *Oracle Video Server Logical Content Administator's Guide* for more information on encoding logical content.

The code examples used in this section are taken from sample applications **ctntdemo.c** and **clipdemo.c**. The **ctntdemo.c** application exercises the **mza::CtntPvdr** and **mza::Ctnt** methods. The **clipdemo.c** application uses the **mza::Clip** and **mza::LgCtnt** interfaces to create logical content from existing clips in the database. The complete applications are listed in Appendix H.

The tasks described in this section are:

Task	Action	Page
Using an Iterator		3-12
Using the Content Provider (CtntPvdr) Interface:		
1.	List Content Providers	3-14
2.	Search for Named Content Provider	3-15
3.	Create Content Provider and Return its Attributes	3-15
4.	Destroy Content Provider	3-16
Using the Content (Ctnt) Interface:		
1.	List Content	3-16
2.	Search for Named Content	3-17
3.	Create Content	3-17
Using the Clip and Logical Content (LgCtnt) Interfaces:		
1.	Locating Clips by Name	3-20
2.	Create New Logical Content	3-21
3.	Add Clips to Logical Content	3-21
4.	Remove Clips from Logical Content	3-22
5.	Destroy Logical Content	3-22

Using an Iterator

When using methods that return lists of items from OVS, it is sometimes more efficient to create the list in parts, using multiple calls to the method, rather than all at once from a single call. This strategy is particularly useful when returning a large number of items.

In order to provide the ability to create lists over multiple calls, methods that return lists of items use an **iterator** to keep track of the positions in the list and the number of items returned before and after each iteration of the method. The iterator is defined by an **mza::Itr** structure, which is passed as an input/output (*inout*) parameter by the list methods.

As shown in Table 3-1, the iterator values have slightly different meanings before and after you invoke the method.

Iterator Value	On Input	On Output
Position in List	Position in list to add next group of items.	New position in list after items have been returned, or -1 if all items have been returned.
Number of Items	Maximum number of items to return.	Number of items actually returned. (The output value may be less than the input value following the last iteration through the list.)

Table 3-1: Iterator Values on Input and Output

Example A server contains 15 *foo* items. A **getfoo()** method allows you to read all of the *foo* items from the server into a list, *fooList*. For various reasons, you can only accommodate a maximum of 4 *foo* items returned from each call to the **getfoo()** method. The **getfoo()** method accepts an **mza::Itr** as an *inout* parameter, which allows you to identify the location in the *fooList* for the next item to be added, along with the maximum number of items to be added, during each call to the **getfoo()** method.

Your first step is to set the iterator, **mza::Itr**, to return up to 4 items, and set the position value to identify the beginning of the list:

```
mza_Itr         itr;

itr.Position = 0;
itr.NumItems = 4;
```

As illustrated in Figure 3-8, the first time you invoke the **getfoo()** method, 4 *foo* items are returned to *fooList* and the *itr.Postion* is set to 4. The next time you invoke the **getfoo()** method, another 4 *foo* items are returned to *fooList* and the *itr.Postion* is set to 8, and so on until the 4th call, when the last *foo* item is returned to *fooList* and the *itr.Postion* is set to -1. Note that *itr.NumItems* now contains a value of 3 to indicate the number of items actually returned in the last call.

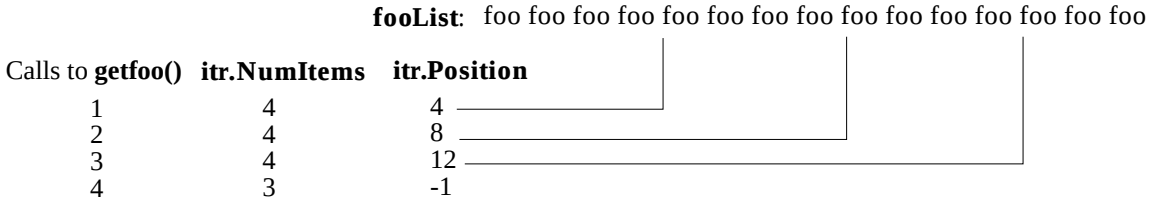


Figure 3-8: Using an iterator to build a list

The code samples in this chapter describe several methods that make use of the **mza::Itr** structure.

Using the Content Provider (CtntPvdr) Interface

An **mza::CtntPvdr** object identifies the source of the content, and can be created when the **mza::Ctnt** is first created, or any time afterwards.

The code examples shown in this section are from the **ctntdemo.c** application listed in Appendix H.

The **mza::CtntPvdrMgmt::lstAtr()** method returns a list that identifies all of the **mza::CtntPvdr** objects in **vscontsrv**. The **mza::CtntPvdrMgmt::lstAtr()** method uses an iterator, **mza::Itr**, to keep track of the position in the list before and after each iteration of the method.

Example In this example, bind to the **mza::CtntPvdrMgmt** interface to use the **mza::CtntPvdrMgmt::lstAtr()** method.

```
mza_CtntPvdrMgmt ctntPvdrMgmtOR;

ctntPvdrMgmtOR = (mza_CtntPvdrMgmt) yoBind(mza_CtntPvdrMgmt__id,
     (char *) 0, (yoRefData *) 0, (char *) 0);
```

Set the iterator, **mza::Itr**, to return up to 10 **mza::CtntPvdr** objects for each call to the **mza::CtntPvdrMgmt::lstAtr()** method. The *itr.Position* value of 0 indicates the start of the list.

```
mza_Itr         itr;

itr.Position = 0;
itr.NumItems = 10;
```

The **mza::CtntPvdrMgmt::lstAtr()** method returns an **mza::CtntPvdrAtrLst** that lists the attributes of each **mza::CtntPvdr** object in the system. The attributes are contained in **mza::CtntPvdrAtr** structures that include the object reference of the **mza::CtntPvdr** object, its name, and any available description.

We don't yet know the number of **mza::CtntPvdr** objects in the server, so create a **while()** loop that returns 10 **mza::CtntPvdrAtr** structures to the **mza::CtntPvdrAtrLst** during each iteration. The *itr.Position* value is incremented to represent the number of **mza::CtntPvdrAtr** structures returned in each loop. When no more **mza::CtntPvdrAtr** structures are found, an *itr.Position* value of -1 is returned, which terminates the **while()** loop.

```
mza_CtntPvdrAtrLst cpAtrLst;

while (itr.Position != -1)
{
  cpAtrLst = mza_CtntPvdrMgmt_lstAtr(ctntPvdrMgmtOR, &env, &itr);

/* print the contents of cpAtrLst */

  mza_CtntPvdrAtrLst__free(&cpAtrLst, yoFree);

}
```

Search for Named Content Provider

Use the **mza::CtntPvdrMgmt::getAtrByNm()** method to search for a specific content provider by name. The **mza::CtntPvdrMgmt::getAtrByNm()** method accepts the name of an **mza::CtntPvdr** object and returns its **mza::CtntPvdrAtr** structure.

Example To return the **mza::CtntPvdrAtr** structure for the content provider, named *StarPoint*, call:

```
mza_CtntPvdrAtr cpAtr;
name = (char *)"StarPoint";

mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env, name, &cpAtr);
```

Create Content Provider and Return its Attributes

Use the **mza::CtntPvdrFac::create()** method to create a new **mza::CtntPvdr** object. The **mza::CtntPvdrFac::create()** method accepts the name of the **mza::CtntPvdr** object, an optional description, and returns an object reference to the newly created **mza::CtntPvdr** object.

Example To create a new content provider, named *In-House*:

```
mza_CtntPvdrFac ctntPvdrFacOR;

newname = (char *)"In-House";
newdesc = (char *)"Promos for upcoming shows";

ctntPvdrFacOR = (mza_CtntPvdrFac) yoBind(mza_CtntPvdrFac__id,
     (char *) 0, (yoRefData *) 0, (char *) 0);

DISCARD mza_CtntPvdrFac_create(ctntPvdrFacOR, &env, newname, newdesc);
```

In this example, the object reference to the new **mza::CtntPvdr** object returned by the **mza::CtntPvdrFac::create()** method is not used. Instead, we use the **mza::CtntPvdrMgmt::getAtrByNm()** method to locate the new **mza::CtntPvdr** object by its name and return its attributes in an **mza::CtntPvdrAtr**.

```
mza_CtntPvdrAtr cpAtr;

mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env, newname, &cpAtr);
```

Destroy Content Provider

Use the **mza::CtntPvdrFac::destroy()** method to destroy an **mza::CtntPvdr** object.

Example The **mza::CtntPvdr::destroy()** method destroys the **mza::CtntPvdr** object on which it is invoked. If you don't have an object reference to the **mza::CtntPvdr** object to be destroyed, you can obtain one by invoking the **mza::CtntPvdrMgmt::getAtrByNm()** method and extracting the object reference from the returned **mza::CtntPvdrAtr** structure.

To destroy the content provider, *StarPoint*:

```
mza_CtntPvdrAtr cpAtr;
name = (char *)"StarPoint";

mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env, name, &cpAtr);

mza_CtntPvdr_destroy(cpAtr.ctntPvdrOR, &env);
```

Using the Content (Ctnt) Interface

This section describes how to use the **mza::Ctnt** object. The examples are taken from the **ctntdemo.c** application in Appendix H.

List Content

The **mza::CtntMgmt::lstAtr()** method returns an **mza::CtntAtrLst** that lists all of the **mza::Ctnt** objects in the system. As with the **mza::CtntPvdrMgmt::lstAtr()** method described in *List Content Providers* on page 3-14 method, the **mza::CtntMgmt::lstAtr()** method uses an **mza::Itr** so the **mza::CtntAtrLst** can be built over multiple invocations.

Example

```
mza_CtntMgmt    ctntMgmtOR;
mza_Itr         itr;
mza_CtntAtrLst  cAtrLst;

ctntMgmtOR = (mza_CtntMgmt) yoBind(mza_CtntMgmt__id, (char *) 0,
     (yoRefData *) 0, (char *) 0);

itr.Position = 0;
itr.NumItems = 10;

while (itr.Position != -1)
{
   cAtrLst = mza_CtntMgmt_lstAtr(ctntMgmtOR, &env, &itr);

/* print the contents of cAtrLst */

mza_CtntAtrLst__free(&cAtrLst, yoFree);
}
```

Search for Named Content

You can use the **mza::CtntMgmt::lstAtrByNm()** method to return the attributes of a named **mza::Ctnt** object. The method returns one or more **mza::CtntAtr** structures in an **mza::CtntAtrLst**. Multiple **mza::CtntAtr** structures can be returned if wildcards are used in the name.

Example To return the **mza::CtntAtr** structure for the content named *Promos*, invoke the **mza::CtntMgmt::lstAtrByNm()** method as follows:

```
mza_Itr         itr;
mza_CtntAtrLst  cAtrLst;

name = (char *)"Promos";

itr.Position = 0;
itr.NumItems = 10;

while (itr.Position != -1)
{
cAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, &env, name, &itr);

/* print the contents of cAtrLst */

mza_CtntAtrLst__free(&cAtrLst, yoFree);
}
```

Create Content

Use the **mza::CtntFac::create()** method to create a new **mza::Ctnt** object.

Example The first step when creating a new **mza::Ctnt** object is to construct an **mza::CtntAtr** structure to describe its attributes. In this example, the MPEG data in the **/mds/video/sample.mpi** file is represented by an **mza::Ctnt** object named *Commercial*; its **mza::CtntPvdr** is represented by the object reference, *ToyCityOR*, which was obtained by calling the **mza::CtntPvdrMgmt::getAtrByNm()** method and passing in "ToyCity." Note that the **ctntOR** field is automatically filled in by the **mza::CtntFac::create()** method.

```
mza_CtntAtr     cAtr;
mza_CtntPvdr    ToyCityOR;

cAtr.ctntOR = (mza_Ctnt)CORBA_OBJECT_NIL;/* Not used in create */
cAtr.ctntPvdrOR = ToyCityOR
cAtr.name = (char *) "Commercial";
cAtr.desc = (char *) "Commercials for the Binky Bopper Cartoon";
```

```
cAtr.createDate.mkd_wallNano = 0;
cAtr.createDate.mkd_wallSec = 0;
cAtr.createDate.mkd_wallMin = 0;
cAtr.createDate.mkd_wallHour = 0;
cAtr.createDate.mkd_wallDay = 1;
cAtr.createDate.mkd_wallMonth = 1;
cAtr.createDate.mkd_wallYear = 1997;

cAtr.filename = (char *)(dvoid*) "/mds/video/sample.mpi";
sysb8set(&cAtr.len,sysb8zero);
cAtr.msecs = 0;
cAtr.rate = 1000;
sysb8set(&cAtr.firstTime,sysb8zero);
sysb8set(&cAtr.lastTime,sysb8zero);

cAtr.format.mkd_contFormatVendor = (char*)0;
cAtr.format.mkd_contFormatFmt = mkd_compFormatMpeg1;
cAtr.format.mkd_contFormatVid._length = 0;
cAtr.format.mkd_contFormatVid._maximum = 0;
cAtr.format.mkd_contFormatVid._buffer = (ub1*)0;
cAtr.format.mkd_contFormatAud._length = 0;
cAtr.format.mkd_contFormatAud._maximum = 0;
cAtr.format.mkd_contFormatAud._buffer = (ub1*)0;
cAtr.format.mkd_contFormatHeightInPixels = 320;
cAtr.format.mkd_contFormatWidthInPixels = 240;
cAtr.format.mkd_contFormatPelAspectRatio = 2;
cAtr.format.mkd_contFormatFrameRate = 30000;

cAtr.prohibFlags = (mkd_prohib) 0;
cAtr.tagsFlag = TRUE;
cAtr.multiRateFlag = FALSE;
cAtr.reliableFlag = FALSE;
cAtr.volLocation = (char *) NULL;
cAtr.contStatus = mkd_contStatusDisk;
cAtr.assigned = FALSE;
cAtr.sugBufSz = mza_BufSzUnknown;
```

Once you have created the **mza::CtntAtr** structure, pass it to the **mza::CtntFac::create()** method to create the new **mza::Ctnt** object.

```
DISCARD mza_CtntFac_create(ctntFacOR, &env, &cAtr);
```

Using the Clip and Logical Content (LgCtnt) Interfaces

The code examples shown in this section are from the **clipdemo.c** application listed in Appendix H. The **clipdemo.c** application uses the **mza::Clip** and **mza::LgCtnt** interfaces to demonstrate how to dynamically build logical content based on certain criteria.

The **clipdemo.c** application creates logical content for two users, John and Mary. Each logical content consists primarily of a movie. To lower the cost of each movie, the application adds a commercial. Since only one commercial is shown, you want to target it as closely as possible to the person requesting the movie.

John orders an action movie. His profile indicates that he works in construction, so the application adds a truck commercial. Mary orders a comedy special. Her profile indicates that she is an executive with a long commute, so the application adds a luxury car commercial. The code that implements the user profiles is not shown in the application and is mentioned here only so you can visualize how a real-life version of this application might work.

Creating New Clips

The **mza::ClipFac::create()** method creates a new **mza::Clip** from a selected **mza::Ctnt** object**.**

Example In this example, the **mza::ClipFac::create()** method creates a new **mza::Clip**, named *JOHNS_MOVIE_CLIP_NAME*, from the *johnMovieCtnt* **mza::Ctnt** object. The *mkdBeginning* and *mkdEnd* parameters indicate that the clip is from the beginning to the end of *johnMovieCtnt* (in other words, it represents the entire **mza::Ctnt** object).

```
mza_Clip johnMovieClip;
mza_Ctnt johnMovieCtnt;

johnMovieClip = mza_ClipFac_create(clipFacOR, &env,
                      johnMovieCtnt,
                      (char*)(dvoid*) JOHNS_MOVIE_CLIP_NAME,
                      (char *)(dvoid*) "Sample Clip for OVS Clip Demo",
                      (mkd_pos*)(dvoid*)&mkdBeginning,
                      (mkd_pos*)(dvoid*)&mkdEnd);
```

The **mza::ClipMgmt::lstAtrByNm()** method returns an **mza::ClipAtrLst** that contains the object reference and other attributes of a named **mza::Clip** object.

Example In this example, the **mza::ClipMgmt::lstAtrByNm()** method returns an **mza::ClipAtrLst** that contains the object reference for the clip named *JOHNS_MOVIE_CLIP_NAME*.

```
const char       *name = "JOHNS_MOVIE_CLIP_NAME";
mza_ClipMgmt     clipMgmtOR;
mza_Clip         clipOR;
mza_Itr          itr;
mza_ClipAtrLst   clAtrLst;

itr.Position = 0;
itr.NumItems = 1;

clipMgmtOR = (mza_ClipMgmt) yoBind(mza_ClipMgmt__id, (char *) 0,
               (yoRefData *) 0, (char *) 0);

clAtrLst = mza_ClipMgmt_lstAtrByNm(clipMgmtOR, &env,
               (char*)(dvoid*)name, &itr);
```

If the clip is located, extract the object reference for the clip from the **mza::ClipAtrLst** and store in *clipOR*.

```
if (clAtrLst._length != 1)
{
  yslPrint("Could not find Clip: <%s>\n",name);
   clipOR = (mza_Clip) CORBA_OBJECT_NIL;
}
else
{
  clipOR = (mza_Clip) yoDuplicate((dvoid *)
                   clAtrLst._buffer[0].clipOR);
}
```

Create New Logical Content

Create a new **LgCtnt** object using the **mza::LgCtntFac::create()** or **mza::LgCtntFac::createCtnt()** method. The **mza::LgCtntFac::create()** method creates an **mza::LgCtnt** object with no clips. The **mza::LgCtntFac::createCtnt()** method creates an **mza::LgCtnt** object with one clip that references an entire **Ctnt** object. Both methods return an object reference to the newly created **mza::LgCtnt** object.

Example In this example, the **mza::LgCtntFac::create()** method creates an empty **mza::LgCtnt** object for John.

```
const char      *johnNm = "John's Logical Content";
mza_LgCtnt      lgCtntOR;

lgCtntOR = mza_LgCtntFac_create(lgCtntFacOR,
                    &env,
                    (char*)johnNm,
                    (char*)"Logical Content created by OVS clip demo");
```

Add Clips to Logical Content

The **mza::LgCtnt::addClip()** and **mza::LgCtnt::addClipByPos()** methods add new clips to an **mza::LgCtnt** object. The **mza::LgCtnt::addClip()** method adds the clip to the end of the **mza::LgCtnt** and returns the position of the newly added clip within the **mza::LgCtnt**. The **mza::LgCtnt::addClipByPos()** method adds a clip to a specific position within the **mza::LgCtnt**.

Example In this example, the movie and commercial clips are added to the **mza::LgCtnt** created for John. We add the clips in reverse order to demonstrate how the two add-clip methods can be used.

Add the movie object reference (*movieClip*) first using the **mza::LgCtnt::addClip()** method. Since this is the first clip added to the **mza::LgCtnt**, the position is not important.

```
sb4   pos;

pos = mza_LgCtnt_addClip(lgCtntOR, &env, movieClip);
```

Use the **mza::LgCtnt::addClipByPos()** method to add the truck commercial (*adClip*) as position 1 (the first clip in the **mza::LgCtnt**.)

```
mza_LgCtnt_addClipByPos(lgCtntOR, &env, adClip, (sb4)1);
```

The *movieClip*, *adClip*, and *lgCtntOR* object references should be released after the logical content has been created.

John's **mza::LgCtnt** object now contains two clips: a truck commercial followed by John's movie selection. In the **clipdemo.c** application, the same process of searching for clips, creating an **mza::LgCtnt** object, and adding the clips to the **mza::LgCtnt** object is repeated for Mary.

Remove Clips from Logical Content

The **mza::LgCtnt::delClip()** and **mza::LgCtnt::delClipByPos()** methods allow you to delete clips from an **mza::LgCtnt** object. (When you "delete clips" from the **mza::LgCtnt** object, you are not removing the **mza::Clip** objects, just the reference to them in the **mza::LgCtnt** object.)

The **mza::LgCtnt::delClip()** method allows you to delete an **mza::Clip** from the **mza::LgCtnt**, based on its object reference. The **mza::LgCtnt::delClipByPos()** method deletes an **mza::Clip** based on its position within the **mza::LgCtnt** object.

Example The **mza::LgCtnt::delClipByPos()** method is used to delete Clip 1 from the **mza::LgCtnt** object.

```
mza_LgCtnt_delClipByPos(lgCtntOR, env, (sb4)1);
```

Destroy Logical Content

The **mza::LgCtnt::destroy()** method destroys an **mza::LgCtnt** object.

Example Call the **mza::LgCtnt::destroy()** method on the object reference to the **mza::LgCtnt** object.

```
mza_LgCtnt_destroy(lgCtntOR, env);
```

After destroying the **mza::LgCtnt**, release the *LgCtntOR*.

```
yoRelease((dvoid *) lgCtntOR);
```

CHAPTER

4

Scheduling Content for Broadcast

The OVS scheduling services provide flexible and generic scheduling features. These services can be used for scheduling OVS logical content and other types of data for playout with OVS, as well as for other types of events that are handled outside of OVS. For example, you can use the OVS scheduling services to schedule events that control equipment other than OVS in a digital broadcast system. The data that makes up the schedule is stored in a database, so a database must be present if you are to make use of the scheduling features.

OVS provides a general "playout" service that allows you to broadcast OVS logical content for pay-per-view (PPV), near video-on-demand (NVOD), as well as regular TV viewing. Most broadcasts of OVS logical content can be scheduled using the Oracle Video Server Manager (VSM) described in *Getting Started with Oracle Video Server Manager*. This chapter describes how you can use the OVS scheduling interfaces to integrate scheduling operations directly into your own applications, as well as how to create your own custom solutions to schedule data other than OVS logical content for broadcast to clients.

Overview of OVS Scheduling Architecture

The OVS scheduling architecture is implemented by three types of services:

- a scheduler service
- a broadcast data service
- one or more exporter services

The scheduler service (**vsschdsrv**) keeps track of the current time and a list of scheduled **broadcast events**. When the time for a broadcast event is reached, the scheduler service forwards the event to one or more implementation-specific **exporter services** that respond to the event.

An example of an exporter service is the near video-on-demand, or NVOD, service (**vsnvodsrv**) that allows you to broadcast OVS logical content on specific channels. The scheduling interfaces described in this chapter allow you to implement your own exporter services that work either along with or independently of the NVOD service.

Note The use of "near video-on-demand" and "NVOD" are misnomers from an earlier architecture of the Oracle Video Server. The **vsnvodsrv** service is better described as a "playout service" that plays selected logical content on a selected channel. While you can use the **vsnvodsrv** service to schedule content for near video-on-demand playout, you can also schedule content for pay-per-view (PPV) and for regular TV broadcasting. For consistency, the term "NVOD service" is used throughout this chapter, but this is not to imply that the **vsnvodsrv** service is exclusive to near video-on-demand operations.

A simplistic view of the relationships between the three types of scheduling services is shown in Figure 4-1. All access to the database is through the broadcast data service (**vsbcastsrv**), which presents the database information as OVS objects to the scheduler service (**vsschdsrv**) and NVOD service (**vsnvodsrv**). These OVS objects are shown in the more detailed illustrations, Figure 4-2 and Figure 4-3.

Other exporters may require their own custom services to manage their data in a database, file system, or other storage mechanism.

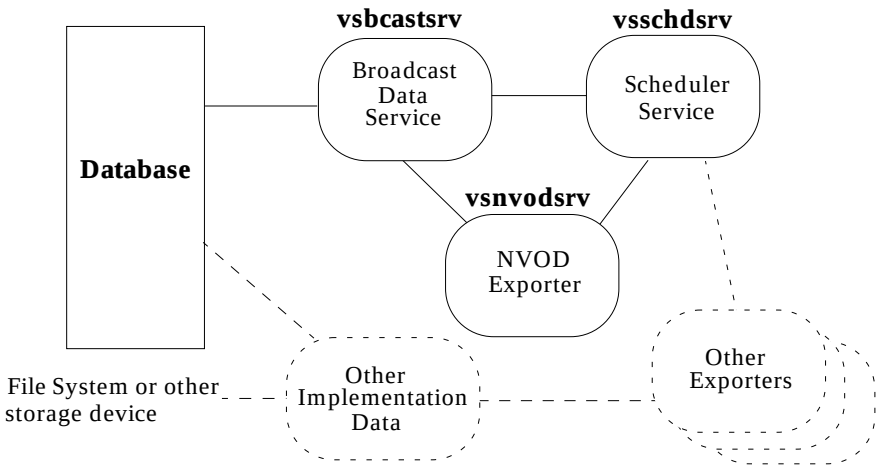


Figure 4-1: OVS Scheduling Architecture

Scheduling Broadcast Events

Figure 4-2 illustrates the scheduling information stored in the database and how it is used by the scheduler service (**vsschdsrv**). Note that all scheduler access to the database is through the **mzabi::Schd**, **mzabi::ExpGrp**, and **mzabi::Exp** objects implemented by the **vsbcastsrv** service.

Each **mzabi::Schd** object represents a separate broadcast event, and allows you to set and get schedule information for that event in the database. The schedule information describes the dates and times a broadcast event is to start and stop, the current status of the event, and a reference to an **exporter group** that identifies which exporter services are to respond to the event.

Each exporter group and exporter registered in the database is represented by an individual **mzabi::ExpGrp** and **mzabi::Exp** object. Figure 4-2 illustrates the objects used to represent one exporter group, **mygroup**, which is associated with two exporter services, **vsnvodsrv** and **myexpsrv**.

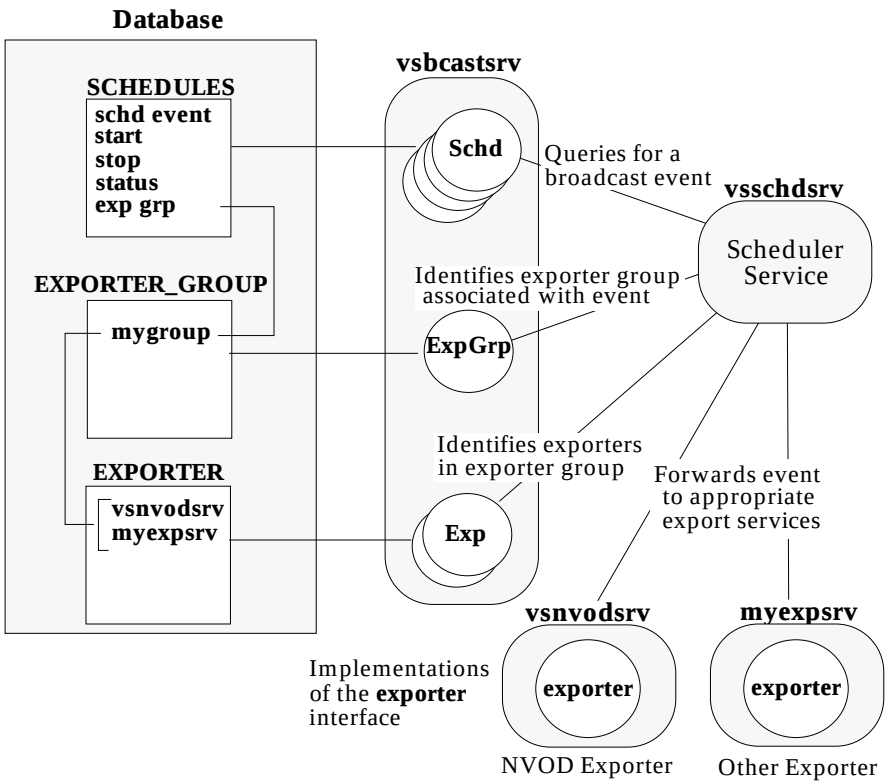


Figure 4-2: Interfaces Implemented by vsbcastsrv

Scheduling OVS Content

The Near Video-on-Demand (NVOD) exporter service allows you to schedule **logical content** for play on a specific **channel** at a specific date and time. An NVOD channel is similar to a broadcasting channel, like a TV channel.

As shown in Figure 4-3, the NVOD exporter service (**vsnvodsrv**) operates in conjunction with the **vsschdsrv**, **vsbcastsrv**, and logical content (**vscontsrv**) services. The **vsnvodsrv** service implements an **mzabix::exporter** object, and accesses the logical content, channel, and other NVOD information in the database through the **mzabin::Nvod** and **mzabin::Chnl** objects implemented by **vsbcastsrv**.

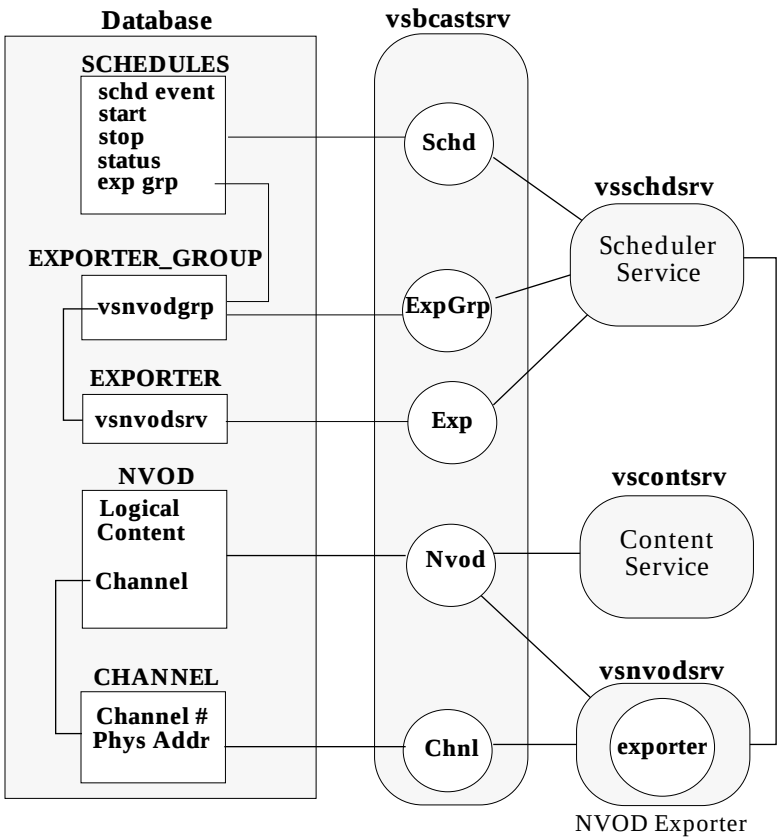


Figure 4-3: NVOD and Scheduler Objects and Database Tables

Each **mzabin::Nvod** object references an **mzabin::Chnl** object and an **mza::LgCtnt** object. The **mzabin::Chnl** object maps a traditional broadcast channel to a physical network address over which OVS streams the scheduled content represented by the **mza::LgCtnt** object.

Using the Scheduling Interfaces

Like the logical content objects discussed in Chapter 3, the scheduling objects have three interface types:

- **Base** interfaces are for operations on base objects. The methods in this category include operations on object attributes and destruction of objects. For example, **mzabi::Schd** is a base object.
- **Factory** interfaces allow you to create new objects. For example, **mzabi::SchdFac** is a factory object.
- **Management** interfaces allow you to query operations and return information about groups of objects. For example, **mzabi::SchdMgmt** is a management object.

Methods that return a list of items use the **mza::Itr** structure. For information about the **mza::Itr** structure, see *Using an Iterator* on page 3-12.

The tasks described in this section are:

Task	Action	Page
1.	**Create a Schedule**	4-8
2.	**Create a Channel**	4-10
3.	**Create an Nvod Object**	4-11
4.	**Create an Exp Object**	4-13
5.	**Create an ExpGrp Object**	4-13

Using the Schedule (Schd) Interface

The OVS scheduler identifies scheduled items as **broadcast events**. However, do not confuse broadcast events with events managed by the Media Net event service, such as the stream events described in Chapter 6.

When creating a new schedule object (**mzabi::Schd**), you define a date and time to start and stop the event, and the status of the event. Your **mzabi::Schd** object must also reference two exporter groups: one to respond to the event, and a **change exporter group** to be notified of any changes to the schedule status. (The exporter group and change exporter group are typically the same.)

Under most circumstances, the exporter services and their respective **vsbcastsrv** objects have already been created during the configuration of your system. For details on creating the **vsbcastsrv** objects needed for your own exporter, see *Using the Exporter Interfaces* on page 4-12.

There are three basic types of broadcast events:

- **Synchronous** – Both start and stop times are specified. There are two types of synchronous events: short and long. When you specify a **synchronous short** event, the scheduler forwards the start and stop times of the event to the exporters in a single call. When you specify a **synchronous long** event, the scheduler forwards the start and stop times of the event to the exporters in separate calls.
- **Asynchronous** – Only the start time is specified; the event completes asynchronously.
- **Change** – Used to notify the “change exporter group” that there has been a change to the schedule.

Create a Schedule

The **mzabi::SchdFac::create()** method creates a new **mzabi::Schd** object. Before you can create a new schedule, you must first determine what exporter group is to be notified of the event, as well as the change exporter group to be notified of any changes to the event schedule.

Example In this example, the broadcast event is directed to the NVOD exporter, **vsnvodsrv**. The exporter group associated with **vsnvodsrv** is named **vsnvodgrp**.

Use the **mzabi::ExpGrpMgmt::lstAtrByNm()** method to obtain an **mzabi::ExpGrpAtr** structure that contains the object reference to the NVOD exporter group, **vsnvodgrp**. This example assumes the NVOD exporter is the only exporter in the system, so the **mzabi::ExpGrpAtrLst** returned contains a single **mzabi::ExpGrpAtr** structure for the **vsnvodgrp**.

```
mzabi_ExpGrpMgmt   ExpGrpMgmtOR;
mzabi_ExpGrp       ExpGrpOR;
mzabi_ExpGrpAtrLst expGrpAtrLst;
mzabi_ExpGrpLst    expGrpLst;
mza_Itr            itr;

itr.Position = 0;
itr.NumItems = 1;

expGrpMgmtOR = (mzabi_ExpGrpMgmt) yoBind(mzabi_ExpGrpMgmt__id,
                    (const char *)NULL,
                    (yoRefData *)0,
                    (const char *)NULL);

expGrpAtrLst = mzabi_ExpGrpMgmt_lstAtrByNm(expGrpMgmtOR, &env,
                          "vsnvodgrp",
                          &itr);
```

Use an **mkd::wall** structure of type **mkd::gmtWall** to hold the start and stop time for the event. In this example, the event is to start on 10 Dec 1997 at 12:00pm Greenwich Mean Time (GMT) , and end the same day at 12:30pm.

```
mkd_gmtWall startTime, stopTime;

startTime.mkd_wallNano = 0;
startTime.mkd_wallSec = 0;
startTime.mkd_wallMin = 0;
startTime.mkd_wallHour= 12;
startTime.mkd_wallDay = 10;
startTime.mkd_wallMonth = 12;
startTime.mkd_wallYear = 1997;
```

```
stopTime.mkd_wallNano = 0;
stopTime.mkd_wallSec = 0;
stopTime.mkd_wallMin = 30;
stopTime.mkd_wallHour= 12;
stopTime.mkd_wallDay = 10;
stopTime.mkd_wallMonth = 12;
stopTime.mkd_wallYear = 1997;
```

Use the **mzabi::SchdFac::create()** method to create a new the **mzabi::Schd** object. The **vsnvodgrp** exporter group (*expGrpOR*) is to be notified when the scheduled start and stop times for the event occur, and when any changes are made to the schedule. The broadcast event is of type *synchShort*, which means that both the start and stop times for the event are to be forwarded to the **vsnvodsrv** in a single call to the **mzabix::exporter::startEvent()** method.

Note that the schedule does not specify an **mzabin::Nvod** object, or provide any other information that indicates the type of operation to be performed in response to the broadcast event. The association between the broadcast event and the operation to be performed on its behalf is made by the exporter service. In the case of NVOD, an **mzabin::Nvod** object contains a reference to this **mzabi::Schd** object (*schdOR*), which is used by the **vsnvodsrv** to associate the **mzabin::Nvod** object with this broadcast event. See *Create an Nvod Object* on page 4-11 for details.

```
mzabi_SchdFac      schdFacOR;
mzabi_Schd         schdOR;
mzabi_ExpGrpAtr    expGrpAtr;

expGrpAtr = expGrpAtrLst._buffer[0];

schdFacOR = (mzabi_SchdFac)yoBind(mzabi_SchdFac__id,
                           (const char *)NULL,
                           (yoRefData *)0,
                           (const char *)NULL);

schdOR = mzabi_SchdFac_create(schdFacOR, &env,
                             startTime,
                             stopTime,
                             expGrpAtr->expGrpOR, /* Exporter Group */
                             expGrpAtr->expGrpOR, /* Change Group */
                             mzabi_synchShort_eventType);
```

Using the NVOD Data Interfaces

You can use the **mzabin::Nvod** and **mzabin::Chnl** interfaces to designate a piece of logical content for broadcast over a specific channel.

Note Like the NVOD service, the name "Nvod" object is a misnomer. The purpose of the **mzabin::Nvod** object is to define a specific piece of logical content to be played on a specific broadcast channel. While the **mzabin::Nvod** object can be used for near video-on-demand operations, it is not exclusive to these operations.

The channel object (**mzabin::Chnl**) simulates a traditional broadcast channel. A channel has a label, a channel number, and a physical network address. The label and channel number are user-defined. You can create and manage NVOD channels using the **mzabin::Chnl, mzabin::ChnlFac**, and **mzabin::ChnlMgmt** interfaces.

Create a Channel

Use the **mzabin::ChnlFac::create()** method to create a new **mzabin::Chnl** object.

Example This example uses the **mzabin::ChnlFac::create()** method to create a new channel and return an object reference (*chnlOR*). This channel is labeled *MVG* and is specified as channel number 4. The network address for channel 4 is UDP:144.25.85.99:9999.

```
mzabin_ChnlFac nvodFacOR;
mzabin_Chnl chnlOR;

chnlFacOR = (mzabi_ChnlFac)yoBind(mzabi_ChnlFac__id,
                                  (const char *)NULL,
                                  (yoRefData *)0,
                                  (const char *)NULL);

chnlOR = mzabi_ChnlFac_create(chnlFacOR, &env,
                          "MVG",   /* Channel name */
                          "UDP:144.25.85.99:9999", /* network address */
                           4);     /* Channel number */
```

Use the **mzabin::NvodFac::create()** method to create a new **mzabin::Nvod** object. You can also use the **mzabin::NvodFac::createSchd()** method to create both an **mzabin::Nvod** and **mzabi::Schd** object in a single operation.

There must be a one-to-one relationship between an **mzabi::Schd** object and an **mzabin::Nvod** object. This relationship is established by identifying the related **mzabi::Schd** object in the **mzabin::Nvod** object. This arrangement may seem counterintuitive at first, but is necessary to keep the scheduler generic and encapsulate all of the implementation-specific information in the exporters.

As shown in Figure 4-4, the **vsschdsrv** calls the **mzabix::exporter::startEvent()** method to forward the broadcast event's start / stop times and an object reference (*SchdOR*) to its respective **mzabi::Schd** object to the **vsnvodsrv** exporter. The **vsnvodsrv** exporter then searches through its list of **mzabin::Nvod** objects and locates the one holding an object reference to the specified **mzabi::Schd** object.

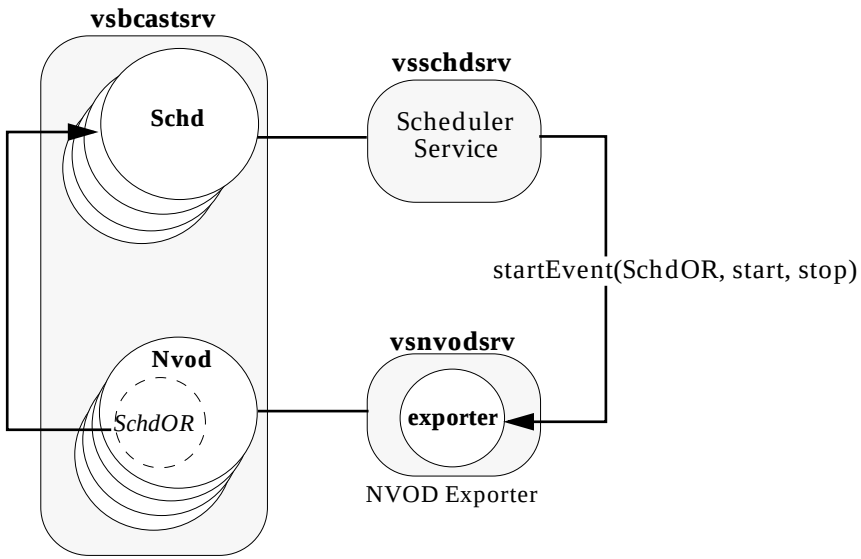


Figure 4-4: Matching a Schd Object with an Nvod Object

Example This example uses the **mzabin::NvodFac::create()** method to create a new **mzabin::Nvod** object and return an object reference (*nvodOR*). This **mzabin::Nvod** object encapsulates a piece of logical content for broadcast on the channel (*chnlOR*) created in the previous example. The **mzabi::Schd** object related to this **mzabin::Nvod** object is identified by the object reference, *schdOR*. This is the **mzabi::Schd** object created in *Create a Schedule* on page 4-8.

```
mzabin_NvodFac  nvodFacOR;
mzabin_Nvod   nvodOR;

nvodFacOR = (mzabi_NvodFac)yoBind(mzabi_NvodFac__id,
                                  (const char *)NULL,
                                  (yoRefData *)0,
                                  (const char *)NULL);

nvodOR = mzabi_NvodFac_create(nvodFacOR, &env,
                              schdOR,/* schedule of this Nvod event */
                              chanOR,/* Channel for Nvod event */
                              lgCtntOR,  /* logical content to be played*/
                             mzabin_loopNone_loopType);  /* play once */
```

Using the Exporter Interfaces

You can create custom exporter services that perform specific operations in response to a scheduled event.

All exporters implement the **mzabix::exporter** interface, which is the interface through which the scheduler service (**vsschdsrv**) communicates with the exporter. The **mzabix::exporter** interface provides methods that are used by the scheduler service to direct an exporter to start and stop events, to notify the exporter of changes to the scheduled event, and to request information from the exporter on the current status of events. See *mzabix::exporter* on page A-131 for details on each **mzabix::exporter** method.

After creating a new exporter service, use the **mzabi::Exp** and **mzabi::ExpGrp** interfaces to register the new exporter service in the database. Each broadcast exporter is associated with an individual **mzabi::Exp** object that maintains the exporter's name, status, implementation ID, and other information in the database. Each **mzabi::Exp** object is associated with an **mzabi::ExpGrp** object, which can reference one or more **mzabi::Exp** objects.

For example, the NVOD exporter service, **vsnvodsrv**, is already configured on your system and can be referenced through the **mzabi::Exp** interface. The exporter group used by the scheduler to reference the NVOD exporter is named *Default NVOD Exporter Group*, and is available through the **mzabi::ExpGrp** interface.

This section describes how to register additional exporter services in the database.

Create an Exp Object

Use the **mzabi::ExpFac::create()** function to create a new **mzabi::Exp** object.

Example This example uses the **mzabi::ExpFac::create()** function to create a new **mzabi::Exp** object for the exporter service, named *myExporter*. The *implId* is the same string used by the *myExporter* service in the **yoSetImpl()** function. (See *Write the Service* on page 5-13 for details on setting the implementation ID for a new service.)

```
mzabi_ExpFac  expFacOR;
mzabi_Exp   expOR;

expFacOR = (mzabi_ExpFac)yoBind(mzabi_ExpFac__id,
                                (const char *)NULL,
                                (yoRefData *)0,
                                (const char *)NULL);

expOR = mzabi_ExpFac_create(expFacOR, &env,
                            “myExporter”,/* name of the exporter */
                            “implId”,   /* as in yoSetImpl call */
                             1000000);  /* setup time, in microseconds */

yoRelease(expFacOR);
```

Create an ExpGrp Object

Use the **mzabi::ExpGrpFac::create()** function to create a new **mzabi::ExpGrp** object.

Example This example uses the **mzabi::ExpGrpFac::create()** function to create a new **mzabi::ExpGrp** object, named *myExpGroup*, to reference the **mzabi::Exp** object you created for the *myExporter* service.

```
mzabi_ExpGrpFac  expFacGrpOR;
mzabi_ExpGrp   expGrpOR;

expFacGrpOR = (mzabi_ExpGrpFac)yoBind(mzabi_ExpGrpFac__id,
                                      (const char *)NULL,
                                      (yoRefData *)0,
                                      (const char *)NULL);

expGrpOR = mzabi_ExpFac_create(expFacOR, &env, “myExpGroup”);
```

Add *myExporter* (*expOR*) to the new exporter group.

```
mzabi_ExpGrp_addExp(expGrpOR, &env, expOR);

yoRelease(expGrpFacOR);
```

CHAPTER

5 Writing a Custom Content Service

This chapter describes the fundamentals of writing your own content service.

As described in *Content Service* on page 1-6 and in Chapter 3, the role of a content service is to provide clients with a particular view of the OVS content. When the client queries the content service for a piece of content, the content service locates the content in storage and returns an **mkd::assetCookie**. When the client is ready to stream the content, it calls the **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** method on the **stream** service and passes the **mkd::assetCookie**. The **stream** service, in turn, calls a **content resolver** to resolve the **mkd::assetCookie** to identify the actual content to be streamed.

Should the OVS logical content model implemented by **vscontsrv** not meet your needs, you can create your own content service. Since there are many ways to write a content service, the information presented in this chapter focuses on the basic framework for creating content query and resolver operations and the OVS interfaces on which this framework is based.

How vscontsrv Works

Before implementing your own content service, it is helpful to understand how the existing OVS content service, **vscontsrv**, operates.

The **vscontsrv** service implements the content query and content resolver interfaces as a single process. The query portion of the **vscontsrv** service implements the **mza::LgCtnt** interface described in *LgCtnt Interface* on page 3-6. The resolver portion of **vscontsrv** implements the **mtcr::resolve** interface.

Figure 5-1 illustrates the role of **vscontsrv** in obtaining the content to be streamed to the client:

1. The client invokes an **mza::LgCtnt::getAtr()** or an **mza::LgCtntMgmt::lstAtrByNm()** method to select the content to be streamed. The **vscontsrv** service returns an **mza::LgCtntAtr** structure that holds the **mkd::assetCookie** for the selected logical content. The OVS architecture allows for more than one content resolver, so each **mkd::assetCookie** begins with an implementation id that identifies the service to be used to resolve the content. (The format of an **mkd::assetCookie** is described in *Define the Asset Cookie Format* on page 5-5.)
2. The client calls the **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** method to stream the logical content. The **mkd::assetCookie** is passed from the client back to OVS via this method.
3. The **stream** service reads the implementation id in the **mkd::assetCookie** that identifies the content resolver object to be used, then invokes an **mtcr::resolve::name()** method on that object to resolve the contents of the **mkd::assetCookie**. (If no database is present to hold the metadata necessary to abstract MDS data into logical content, then the **mkd::assetCookie** is resolved directly by the **stream** service.)
4. The **mtcr::resolve** object reads the contents of the **mkd::assetCookie** and locates the relevant logical content metadata in the database to generate an **mkd::segmentList**, which is a sequence of **mkd::segment** structures that describe the MDS files to play, along with position and rate control information for each segment.
5. The stream service directs the video pump to stream the video data specified in the **mkd::segmentList**.

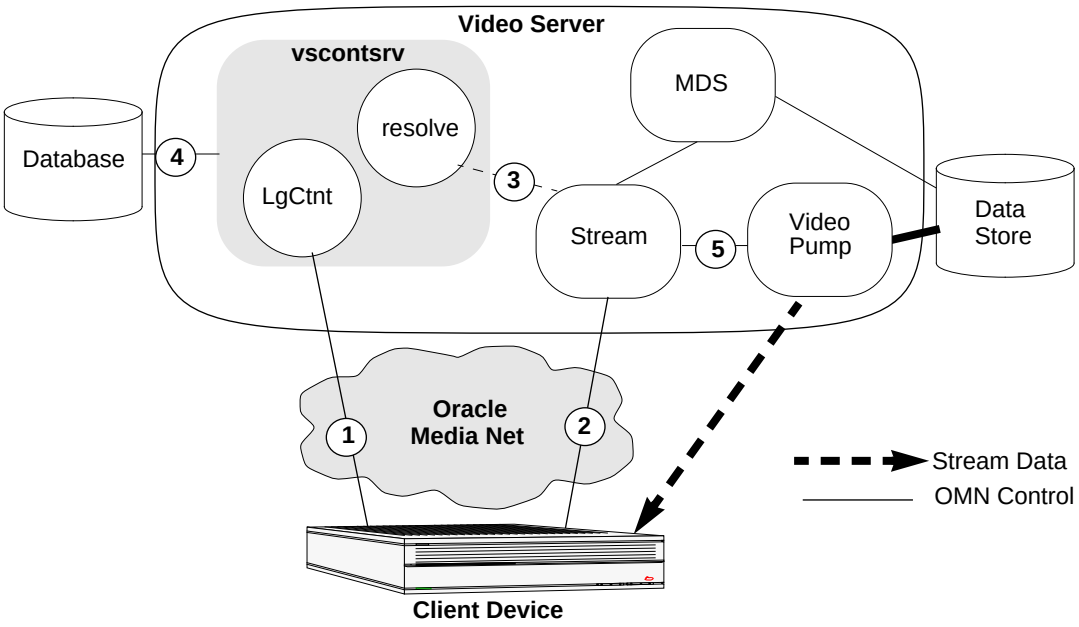


Figure 5-1: Resolving Content

Sample Content Service

The example content service described in this chapter illustrates the basic procedures for writing a custom content service that makes use of a different data model than **vscontsrv**. These procedures include defining and implementing new Media Net interfaces. If you are not familiar with Media Net, read the first few chapters of the *Oracle Media Net Developer's Guide* to obtain the background necessary to follow the discussion in this section.

This code example described in this chapter makes use of the following files, which are listed in Appendix H and described below:

File Name	Description	Listing on Page
cont.idl	Description of content query interface (**cont**)	H-75
contImpl.c	Implementation of content query interface	H-78
rslvImpl.c	Implementation of content resolver interface (**mtcr.idl**)	H-84
dcontsrv.c	Service that implements both **cont** and **mtcr** interfaces as a single process	H-43
contclnt.c	Client that uses the **dcontsrv** service	H-47
metafile1.dat	File containing the content metadata	H-87

The content query portion of the content service uses a custom interface, **cont**, which is described in **cont.idl** and implemented in the **contImpl.c** file. The resolver portion uses the existing **mtcr** interface described in **mtcr.idl** and implements the **mtcr_resolve_name()** method in the **rslvImpl.c** file. The operations implemented by the **contImpl.c** and **rslvImpl.c** files are bound into a single service by the **dcontsrv.c** file.

The client described in the **contclnt.c** file queries the **dcontsrv** service for video content and prepares a stream. The client uses a **demo_cont_queryByNm()** method to retrieve an **mkd::assetCookie** and then passes it to the **mzs::stream::prepare()** method. The content metadata is held in a small "database," which is represented by the **metafile1.dat** file.

The **cont** interface also describes **create()**, **addSegData()**, and **query()** methods to create and manage the content. There is no method for deleting content, but this can be accomplished by editing the **metafile1.dat** file. You can also write your own delete content method as an exercise.

Writing a Content Service

Here's a summary of the tasks involved in writing a content service.

Task	Action	Details on Page
1.	**Define the Asset Cookie Format**	5-5
2.	**Define a Content Query Interface**	5-6
3.	**Implement the Content Query Interface**	5-7
4.	**Implement the Content Resolver Interface**	5-9
5.	**Write the Service**	5-13
6.	**Write a Test Client**	5-14

Define the Asset Cookie Format

An **mkd::assetCookie** uses the format:

```
ImplId:AssetCookieName
```

The *ImplId* is the implementation id used by the **stream** service to identify the specific implementation of the **mtcr::resolve** interface to resolve the *AssetCookieName*. This *ImplId* is established when the content service registers its interfaces with the Media Net Object Request Broker (ORB), as described in *Write the Service* on page 5-13. The *ImplId* in the **mkd::assetCookie** must match the *ImplId* for the **mtcr::resolve** interface that was registered by the service.

The *AssetCookieName* is opaque to everything but the content query object that generated it and the content resolver object used to convert it to an **mkd::segmentList**.

Example In this example, the *ImplId* identifies the **dcontsrv** content service and the *AssetCookieName* is the name of a piece of content in an ASCII file. This ASCII file contains the metadata needed by the **dcontsrv** content resolver operation to create the **mkd::segmentList** for the content. Trade-offs in accuracy have been made in order to keep the example simple. The metadata used to build an **mkd::segmentList** would normally come from a database table or read directly from the MDS, rather than from a file as in this example.

The ASCII file used by the content resolver to make up the **mkd::segmentList** contains one line for each piece of available content. Each line is identified by a content name, followed by one or more groups of fields that define the content segments. Each segment group is delimited by colons (:) and consists of 6 fields separated by spaces. These fields are:

Field	Type	Description
1	string	MDS tag file
2	long	Start Position in seconds
3	long	Stop Position in seconds
4	char	RW Flag (Y | N)
5	char	FF Flag (Y | N)
6	char	Pause Flag (Y | N)

The contents of the **metafile1.dat** file that contains the metadata used in this example is shown below. Note that *Content1* resolves to an **mkd::segmentList** consisting of 3 segments from the **ovs_mpg1_1536k.mpi** file, and *Content3* to 2 segments from the **ovs_mpg1_1536k.mpi** and **ovs_mpg1_2048k.mpi** files.

```
Content1:  /mds/video/ovs_mpg1_1536k.mpi 0 5 N N N:/mds/video/
ovs_mpg1_1536k.mpi 5 10 N N N:/mds/video/ovs_mpg1_1536k.mpi 10 20 N N N

Content2:  /mds/video/ovs_mpg1_2048k.mpi 0 10 N N N

Content3:  /mds/video/ovs_mpg1_1536k.mpi 0 30 N N N:/mds/video/
ovs_mpg1_2048k.mpi 0 30 N N N
```

Define a Content Query Interface

The role of the “content query” interface is to allow a client to query the available video content and return to the client one or more **mkd::assetCookie**s that can be used by the “resolver” to identify the segments to be streamed.

A more advanced query method should provide the option to obtain certain metadata about how the particular piece of content is stored. Such metadata is used by special applications such as “asset managers” that manipulate certain pieces of content to create other content.

Example In this example, the content query methods are described in the **cont.idl** file and implemented in the **contImpl.c** file. The **metafile1.dat** file is used to represent the "database" that stores the metadata needed by the **stream** service to stream the content. In the standard implementation of OVS, the content service, **vscontsrv**, stores this metadata in a database or, if no database is present, in an MDS tag file. How you store the metadata for your content depends on your special needs and is not in the scope of this discussion.

The **cont** client described in *Write a Test Client* on page 5-14 makes use of only one of the methods provided by the **cont** interface. This is the **demo_cont_queryByNm()** method, which is defined in the **cont.idl** file as follows:

```
contAtr queryByNm(in string name, in boolean longFmt)
        raises (NoContent, FileError);
```

The implementation of the **demo_cont_queryByNm()** method in the **contImpl.c** file accepts the *name* of a piece of content, such as *Content1*, *Content2*, or *Content3*, locates the metadata that corresponds to the named content in the **metafile1.dat** file, and uses this metadata to construct and return a **contAtr** structure to the client. The **contAtr** structure is defined as:

```
struct contAtr {
     string contName;
     mkd::assetCookie cookie;
     segDataLst data;
};
```

The *contName* field is the name of the content and **cookie** is the **mkd::assetCookie** to be passed to the **stream** service. The *data* field is a sequence of **segData** structures that hold the metadata about the content. These **segData** structures are only returned if the *longFmt* parameter in the **demo_cont_queryByNm()** method is set to *TRUE*.

Implement the Content Query Interface

Once you've defined your content query interface in the **cont.idl** file, run the file through the IDL compiler to generate a **skeleton file**. This skeleton file provides the framework for your implementation of the **cont** interface. The skeleton file generated will be named **contI.c**. Before adding the implementation code, it is a good practice to rename your skeleton file so that your work is not overwritten the next time you recompile the IDL file. In this example, the skeleton file has been renamed **contImpl.c**. (See the *Oracle Media Net Developer's Guide* for more information on using the IDL compiler.)

Example The **contImpl.c** file implements the content query operations described in the **cont.idl** file. The **externdef** statement indicates that this implementation of the **cont** interface can be located by the ORB through the dispatch vector, *demo_cont__impl*.

```
externdef CONST_W_PTR struct demo_cont__tyimpl demo_cont__impl =
 {
 demo_cont_create_i,
 demo_cont_addSegData_i,
 demo_cont_query_i,
 demo_cont_queryByNm_i
 };
```

This discussion focuses on the **demo_cont_queryByNm_i** operation, which simply reads the metadata associated with some named content from the **metafile1.dat** file and constructs a **contAtr** structure in a manner similar to that shown below. For information on how the other operations in the **cont** interface are implemented, see the **contImpl.c** file on page H-78.

```
demo_contAtr atr;

atr =  demo_cont_queryByNm_i(or, ev, name, FALSE);
```

Open the **metafile1.dat** file and locate the line that matches the content name specified by the client.

```
FILE           *f1;
char           *tok;
demo_contAtr    atr;
char            line[MAX_LINE_LENGTH];
boolean         found = FALSE;

f1 = open_file("r");

while (fgets(line, MAX_LINE_LENGTH, f1) != (char *) NULL)
{
   if (strncmp(line, name, strlen(name)) == 0)
   {
```

Write the name of the content to the *contName* field. Next, construct an **mkd::assetCookie** that consists of the implementation id of the service that implements the content resolver (*myImplId*), a ':' delimitor, and the content name. (Note that the **ysFmtStr()** function is the Media Net equivalent of the **sprintf()** function.)

```
    tok = strtok(line, ":");
    atr.contName = (char *) yoAlloc(strlen(tok) + 1);
    strcpy(atr.contName, tok);
    atr.cookie = (mkd_assetCookie)
                 yoAlloc((mkd_assetCookieMaxlen + 1) * sizeof(char));
    ysFmtStr((char *) atr.cookie, "%s:%s", myImplId, tok);
```

If the *longFmt* parameter is *TRUE*, construct a list of **segData** structures from the remaining fields in the **metafile1.dat** file. Otherwise, return a NULL value.

```
    if (longFmt)
    {
     /*
      * Construct segData structures from the remaining fields
      * in the f1 file.
      */

    }
}

    else
    {
       atr.data._length = 0;
       atr.data._maximum = 0;
       atr.data._buffer = (demo_segData *) 0;
     }

}
```

Close the file and return the **contAtr** structure.

```
fclose(f1);
return atr;
```

Implement the Content Resolver Interface

As described in *How vscontsrv Works* on page 5-2, the **stream** service invokes an **mtcr::resolve::name()** method to resolve the content specified by the **mkd::assetCookie**. There can be more than one content resolver in an OVS system, so the implementation id at the beginning of the **mkd::assetCookie** is used by the **stream** service to identify the specific implementation of the **mtcr::resolve** interface used to resolve the **mkd::assetCookie**.

The custom content resolver created below is simply a new implementation of the **mtcr::resolve::name()** method described in the **mtcr.idl** file. The purpose of the **mtcr::resolve::name()** method is to locate the metadata related to the named content and return it in the form of an **mkd::segmentList**. The **mkd::segmentList** is a sequence of **mkd::segment** structures that describe the file containing the encoded content, the position information within the file necessary to define a **segment**, and rate control information for each segment.

If your resolver is to operate with an authorization service, see *Authorize the Client* on page 5-12 for details on how the resolver might call such an authorization service.

Example The content resolver implementation in this example begins as a skeleton file generated from the **mtcr.idl** file by the IDL compiler. The **mtcrI.c** skeleton file is then copied to the **rslvImpl.c** file, where the implementation code described here is added to implement the **mtcr_resolve_name_i()** operation.

The **externdef** statement indicates that this implementation of the **mtcr::resolve** interface is to be located by the ORB through the dispatch vector, *mtcr_resolve__impl*.

```
externdef CONST_W_PTR struct mtcr_resolve__tyimpl mtcr_resolve__impl =

{
     mtcr_resolve_name_i
};

mkd_segmentList mtcr_resolve_name_i(mtcr_resolve or,
                                    yoenv * ev,
                                    char *name,
                                    CORBA_Object authRef,
                                    yoany *ckt)
```

Bind to the **demo_cont** interface and invoke **demo_cont_queryByNm_i()** on the content name passed in by the **mtcr_resolve_name()** method. Note that **demo_cont_queryByNm_i()** is a direct call to the method implementation. Such direct method invocations are only possible when the method and its caller are implemented in the same service.

The **demo_cont_queryByNm()** method returns a **contAtr** structure that contains the name of the content and an **mkd::assetCookie** for the content. Since the *longFmt* parameter is set to *TRUE*, the **contAtr** structure also returns a **segDataLst** that holds the metadata needed to create the **mkd::segmentList** to be returned to the **stream** service.

Note that **mtcr_resolve_name()** could have obtained the metadata directly from the **metafile1.dat** file. However, since the **segDataLst** returned in the **demo_contAtr** structure contains all of the metadata required, it is easier to simply call **demo_cont_queryByNm_i()**.

```
demo_cont demoOR;
demo_contAtr    atr;

demoOR = (demo_cont) yoBind( demo_cont__id,
                             (char *)0,
                             (yoRefData *)0,
                             (char *)0 );

atr = demo_cont_queryByNm_i(demoOR, ev, name, TRUE);

yoRelease((dvoid*)demoOR);
```

A **segData** structure holds the metadata needed to create an **mkd::segmentList** to return to the stream service. The **demo_cont_queryByNm()** method copies this metadata from the **metafile1.dat** file into the **segData** structures. The **segData** structure is defined as:

```
struct segData {
     string fileName;        // The Tag File Name
     unsigned long startPos; // The Start Position
     unsigned long stopPos;  // The Stop Position
     char rwFlag;            // Rewind Allowed? (Y/N)
     char ffFlag;            // Fast Forward Allowed? (Y/N)
     char pauseFlag;         // Pause Allowed? (Y/N)
};
```

Use **yoAlloc()** to allocate memory for the **mkd::segmentList**, based on the size of the **segDataLst** returned by the **demo_cont_queryByNm()** method. (Note that you must use **yoAlloc()** when allocating data returned to clients. See the *Oracle Media Net Developer's Guide* for details.)

```
mkd_segmentList retVal;
demo_contAtr    atr;
int             maxSegments = 0;
CLRSTRUCT(retVal);

retVal._length = 0;
retVal._maximum = (ub4) atr.data._length;
retVal._buffer = (mkd_segment *) yoAlloc((size_t) (sizeof(mkd_segment) *
                 retVal._maximum));

DISCARD memset((dvoid *) retVal._buffer, 0, (size_t) (retVal._maximum *
                 sizeof(mkd_segment)));
```

Copy the information from the **segDataLst** to the **mkd::segmentList**. Each iteration of the **for()** loop allocates a new **mkd::segment** structure, then copies the name of the tag file for the content from a **segData** structure to the **mkd_segFile** field in the **mkd::segment** structure.

```
for (i = 0; i < atr.data._length; ++i)
{
   retVal._buffer[i].mkd_segFile = yoAlloc(
                              strlen(atr.data._buffer[i].fileName)+1);

   strcpy(retVal._buffer[i].mkd_segFile, atr.data._buffer[i].fileName);
```

Next, write the remaining information extracted from the **segData** structure to their respective fields in the **mkd::segment** structure. (See the complete code example on page H-84 for details.)

After all of the information has been copied from the **segDataLst** to the **mkd::segmentList**, close the file and return the **mkd::segmentList**.

```
  fclose(f1);

  return retVal;
}
```

Authorize the Client

If you wish to authorize the client before returning the **mkd::segmentList** to the **stream** service, your content resolver can either implement an authorization operation, or call another object that authorizes the client.

Examples: The code sample below assumes an object reference (*authRef*) to an **authIntf** interface was passed to the stream service by the client in the **mzs::stream::prepare()** call and that this interface implements an **authorize()** method.

```
yoenv *env;
yoEnvInit(&env);

result = authIntf_authorize(authRef, &env);
if(!result)
     yseThrow(MTCR_EX_AUTHFAILED);
```

An alternative or an additional authorization method is to authorize the client's circuit, which can be passed by the **stream** service to the resolver via the *circ* parameter in the **mtcr_resolve_name()** method.

Below, *circ* is a copy of the **mzc::circuit** used to allocate the stream. The **mzc::circuit** contains the downstream address over which the content is to be streamed. You could define a method, such as **verify_Address(),** to find out if the downstream address of the circuit, *down_Addr*, is authorized to receive the content.

```
if(*ckt->_type == yotkNull)
{
  yseThrow(MTCR_EX_AUTHFAILED);
}
else
  circ = (mzc_circuit *)ckt->_value;
  down_Addr = circ->info.downstream.info.comm.protocol.info
  result = verify_Address(down_Addr)
  if(!result)
    yseThrow(MTCR_EX_AUTHFAILED);
```

Write the Service

After implementing the operations described by the **cont.idl** and **mtcr::resolve** interfaces, define a service to execute these operations as a single process.

Example The sample code described in this section is extracted from the sample content resolver service implemented by the **dcontsrv.c** file.

Establish *dcontsrv* as the implementation id of this service and store in the global variable, *myImplId*. As shown in *Write a Test Client* on page 5-14, *dcontsrv* is the implementation id returned in the asset cookie that directs the stream service to use this implementation of the content resolver.

```
externdef char  *myImplId = "dcontsrv";
```

Initialize the Media Net environment, as described in *Initialize Media Net* on page 1-11.

The **yoSetImpl()** functions below bind the operations dispatched by the content query implementation (*demo_cont__impl*) in the **contImpl.c** file and the content resolver implementation (*mtcr_resolve__impl*) in the **rslvImpl.c** file into a single service implementation called *myImplId*. (Note that *demo_cont__id* and *mtcr_resolve__id* are the **interface ids**, and *demo_cont__stubs* and *mtcr_resolve__stubs* are the **stub routines** assigned by the IDL compiler to the interfaces in their respective ***I.h** header files.)

```
yoSetImpl(demo_cont__id,
          myImplId,
          (yostub *) demo_cont__stubs,
          (dvoid *) & demo_cont__impl,
          (yoload) 0,
          TRUE,
          (dvoid *) 0);

yoSetImpl(mtcr_resolve__id,
          myImplId,
          (yostub *) mtcr_resolve__stubs,
          (dvoid *) & mtcr_resolve__impl,
          (yoload) 0,
          TRUE,
          (dvoid *) 0);
```

The **yoImplReady()** functions register *myImplId* with the Media Net Object Request Broker (ORB) as an implementation of the *demo_cont__id* and *mtcr_resolve__id* interfaces. Once these interfaces are registered with the ORB in this manner, each time a method is invoked on an object reference to the **demo_cont** or **mtcr_resolve** interface, the ORB will forward the request to the **dcontsrv** service.

```
yoImplReady(demo_cont__id, myImplId, (ysque *) 0);
yoImplReady(mtcr_resolve__id, myImplId, (ysque *) 0);
```

Start servicing requests queued to *ysque*:

```
yoService((ysque *) 0);
```

The service runs until shut down using the **mnorbadm** command. This causes the *myImplId* implementation of the interfaces to be "unregistered" with the ORB.

```
yoImplDeactivate(demo_cont__id, (char *) myImplId);
yoImplDeactivate(mtcr_resolve__id, (char *) myImplId);

   return;

}
```

Write a Test Client

Your final task is to write a client application to test your new content service. The client described in this example is a C application.

Example The client that demonstrates how to use the **dcontsrv** content service is named **contclnt.c**. The **contclnt.c** client application is based on the **ovsdemo.c** application described in Chapter 2. Both of these sample applications are listed in Appendix H. Only the differences unique to using the **dcontsrv** content service are described in this section.

The **contclnt.c** client binds to the **demo_cont** interface created in *Define a Content Query Interface* on page 5-6 and invokes its **demo_cont_queryByNm()** method to obtain a **demo_contAtr** structure that contains the **mkd::assetCookie** associated with the content named *Content1*. Since all we really need is the **mkd::assetCookie**, the *longFmt* parameter is set to *FALSE* and no **segDataLst** is returned in the **demo_contAtr** structure.

```
demo_cont   demoOR;
demo_contAtr contAtr;
contentName = (char*) "Content1";

demoOR = (demo_cont) yoBind( demo_cont__id,
                             (char *)0,
                             (yoRefData *)0,
                             (char *)0 );

contAtr = demo_cont_queryByNm(demoOR, env, (char*)contentName, FALSE);

yoRelease((dvoid*)demoOR);
```

Invoke the **mzs::stream::prepare()** method, passing in the **mkd::assetCookie** obtained from the **demo_cont_queryByNm()** method, to prepare the stream to play *Content1* from beginning to end:

```
ub4   bitrate;
mkd_segInfoList status;
mzs_stream_instance inst;

inst = mzs_stream_prepare( strm, env,
                           contAtr.cookie,
                           (mkd_pos *)&mkdBeginning,
                           (mkd_pos *)&mkdEnd,
                           bitrate,
                           mzs_stream_playNow,
                           &status,
                           (dvoid *)0);
```

From this point on, the client application operates in a manner similar to the client described in Chapter 2.

CHAPTER

6 Capturing Stream Events

The events that can be generated by the stream service are listed in Appendix D, "OVS Stream Events". This chapter describes how to use the Media Net **event service** to implement an **event consumer** to capture stream service events.

As illustrated in Figure 6-1, the event service facilitates the transfer of events between objects. The object that produces the event data is called an **event supplier**, and the object that receives and processes the event data is an **event consumer**. An **event channel** is an intermediary object that allows multiple suppliers to communicate with multiple consumers asynchronously. The event channel for the stream service uses the **push model**, which means the supplier initiates the sending of event data to the consumer.

Figure 6-1: Event Service Objects

The OVS stream service implements an event supplier that produces the stream-related events. The event channel associated with stream events is named MZS_EVENT_CHANNEL, and is located by name through the Media Net **naming service**. In order to capture stream events, you must implement an event consumer and *attach* it to the stream service event channel.

Both the event service and naming service are described in detail in the *Oracle Media Net Developer's Guide*. The remainder of this chapter describes how to implement an event consumer for events generated by the stream service, how to *attach* the event consumer to the stream service event channel, and how to process the incoming events.

Implement an Event Consumer

A consumer of stream events must be able to receive events from the stream event channel and then process them in some manner. The event consumer described in this chapter is based on the sample application in the **eclisten.c** file. This application establishes itself as an event consumer object that attaches to a supplier of stream events and implements the **sendEvent()** operation defined by the **mzs::ec** interface.

Note Because OVS does not provide an event consumer, the **sendEvent()** operation is not implemented anywhere in OVS. It is your task to write your own event consumer and implement the **sendEvent()** operation in the process.

When the stream service performs an operation that generates an event, its event supplier invokes the **sendEvent()** method on our event consumer service. The implementation of the **sendEvent()** method in the **eclisten.c** application (**SendEvent_i**) simply accepts any of the stream-related events from the supplier and prints them. You will probably want to implement your event consumer to do more upon receiving incoming events, but the specifics are entirely up to you.

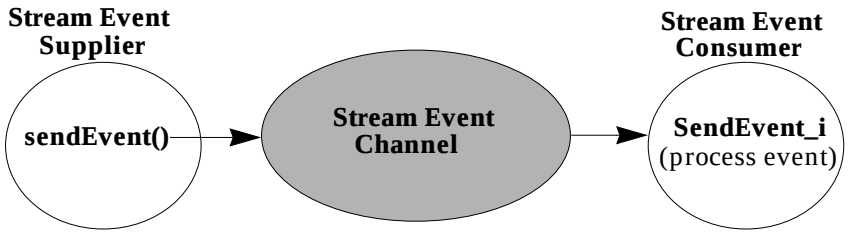


Figure 6-2: Event Supplier Invokes the sendEvent() Method.

The following is a summary of the tasks involved in creating a consumer of stream events:

Task	Action	Details on Page
1.	**Establish an Event Consumer Object**	6-3
2	**Get the Stream Event Channel**	6-5
3.	**Attach Event Consumer to Event Supplier**	6-7
4.	**Implement the sendEvent() Operation**	6-9
5.	**Process Events**	6-12
6.	**Detach Consumer from the Stream Event Channel**	6-12

Table 6-1: Capturing Stream Events

Note The **yo*** functions used in this example are described in the *Oracle Media Net Developer's Guide*; the **ys*** functions are described in **ys.h** and the other **ys*.h** files.

Establish an Event Consumer Object

The **eclisten.c** sample application defines and implements an event consumer object that listens for events generated by the stream service. This application implements the **sendEvent()** operation defined by the **mzs::ec** interface described in the **mzsec.idl** file. (The implementation of the **sendEvent()** operation is described in *Implement the sendEvent() Operation* on page 6-9.)

Most of the functions and procedures described in this section are discussed in more detail in the *Oracle Media Net Developer's Guide*.

Example The **extern** statement indicates that this implementation of the **mzs::ec** interface can be located by the ORB through the dispatch vector, *mzs_ec__impl*.

```
extern const struct mzs_ec__tyimpl mzs_ec__impl =
{
  SendEvent_i
};
```

Use the **mtuxSimpleInit()** function to initialize Media Net for this event consumer process, named "Event Channel Listener."

```
ub1           osdp[SYSX_OSDPTR_SIZE] = {0};
eventChannel  ecV;

if (mtuxSimpleInit(osdp, "Event Channel Listener", (mnLogger)0) !=
     mtuxLayerSuccess)
   return 1;
```

Use the **yoSetImpl()** function to bind the **sendEvent()** operation dispatched by the implementation (*mzs_ec__impl*) into a server called *mzs_ec__implid*.

```
yoSetImpl(mzs_ec__id, mzs_ec_implid, mzs_ec__stubs,
          (dvoid*) &mzs_ec__impl, (yoload)0, FALSE, (dvoid *) 0);
```

Define an **eventChannel** structure to hold the context for the event channel used by this event consumer.

```
typedef struct eventChannel
{
  yeceCa_ProxyPushSupp   orTPS;
  mzs_ec                 orCUE;
  ysque                 *ImplCliQueEC;
  boolean                valid;
} eventChannel;
```

Use the **yoQueCreate()** function to create a queue (**ImplCliQueEC**) in this event channel to hold incoming events for our event consumer. The **ysque** created holds the events sent from the supplier of stream events until our event consumer is ready to receive them. The event channel delivers the events from the queue to the event consumer whenever it's available to receive and process those events. Save a pointer to the queue in the **eventChannel** structure.

```
ysque                *ImplCliQueEC;
eventChannel  *ecP = (eventChannel  *)usrp;

ecP->ImplCliQueEC = yoQueCreate("CliEC");
```

Use the **yoImplReady()** function to notify the Media Net object layer that this implementation (*mzs_ec_implid*) of the event consumer interface (*mzs_ec__id*) is ready to accept requests on its implementation of the **sendEvent()** operation.

```
yoImplReady(mzs_ec__id, mzs_ec_implid, (void*)0);
```

Use the **yoCreate()** function to return an object reference (*orCUE*) to this specific implementation (*mzs_ec_implid*) of the event consumer interface (*mzs_ec__id*).

```
mzs_ec   orCUE = NULL;

orCUE = (mzs_ec) yoCreate(mzs_ec__id,
                          mzs_ec_implid,
                          (yoRefData*)0,
                          (char*)0,
                          NULL);
```

Use the **yoObjListen()** function to tell the ORB to begin listening for events on the *ImplCliQueEC* queue for our implementation of the event consumer (*orCUE*).

```
yoObjListen(orCUE, ecP->ImplCliQueEC);
```

Save a pointer to the event consumer in the **eventChannel** structure.

```
ecP->orCUE = orCUE;
```

Note that you will need to create a connection between the event listener and the event supplier before any events can be received and queued by the event listener. The procedures for creating this connection are described in the following sections.

Get the Stream Event Channel

Before the event listener can receive any stream events, you must locate the event channel for stream service events and attach the event consumer to it.

Stream events are published in a **typed event channel** named MZS_EVENT_CHANNEL. This event channel is known as a “typed” event channel because it only handles events generated from a specific interface, which in this case is the **mzs::stream** interface.

Only one event channel for stream events exists in the OVS system, even if multiple stream services are active. When a stream service starts, it checks to see if another stream service has previously created an event channel. If not, the starting stream service creates a new event channel. Whichever stream service creates the event channel publishes it in the CORBA naming service under the name MZS_EVENT_CHANNEL.

To locate the MZS_EVENT_CHANNEL, you must first obtain an object reference to an **InitialNamingContext** object, which represents the "root" of the naming context. This initial naming context is conceptually similar to the root directory in a file system. From the initial naming context, you can then use the **ydnmNamingContext_resolve()** function to obtain an object reference to any named object published in the naming service.

Example This example describes how to return an object reference to the MZS_EVENT_CHANNEL from the naming service. The objects used for this task are illustrated in Figure 6-3.

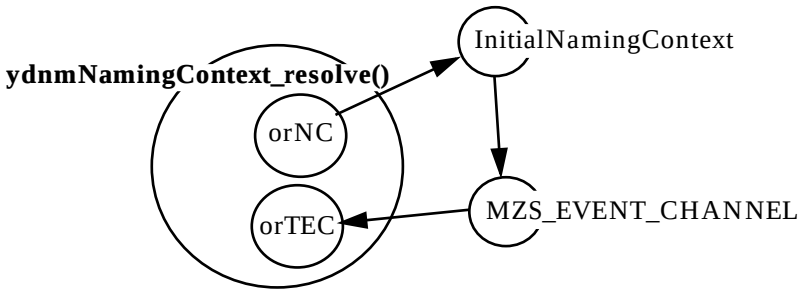


Figure 6-3: Locating the Event Channel in the Naming Service

Find the naming service and return an object reference. The **yoBind()** function returns a loosely-bound object reference (*orINC*) to the **InitialNamingContext** object, which is then used by the **ydnmInitialNamingContext_get()** function to return a tightly-bound reference (*orNC*). A tightly-bound object reference is needed to ensure that all calls are directed to the same naming service

```
ydnmInitialNamingContext    orINC = NULL;
ydnmNamingContext           orNC = NULL;
yoenv                       evNM;

yoEnvInit(&evNM);

  orINC = (ydnmInitialNamingContext)
    yoBind(ydnmInitialNamingContext__id, 0, (yoRefData*)0, (char*)0);

  orNC = ydnmInitialNamingContext_get(orINC, &evNM);
```

Use the **ydnmNamingContext_resolve()** function to return an object reference (*orTEC*) to the object associated with the name MZS_EVENT_CHANNEL.

```
ydnmName                    name;
ydnmNameComponent           name_comp[2];
yeceTeca_TypedEventChannel  orTEC = NULL;

name_comp[0].kind = name_comp[1].kind = (char *)0;
name_comp[0].id   = MZS_EVENT_CHANNEL;

name._length = name._maximum = 1;
name._buffer = name_comp;

yseTry
{
  orTEC = (yeceTeca_TypedEventChannel)
           ydnmNamingContext_resolve(orNC, &evNM, &name);
}

  /* Check for errors */
```

Release the object references that are no longer needed.

```
yoRelease(orNC); orNC = NULL;
yoRelease(orINC);
yoEnvFree(&evNM);
```

Attach Event Consumer to Event Supplier

After obtaining an object reference to the event channel, attach the event consumer to the event supplier. To do this you must first obtain an object reference to the **administration object** for the typed event channel. The administration object provides the functions you need to create the connection between the consumer and supplier in the event channel. This connection is not made directly between the consumer and supplier objects, rather it is made through the **proxy objects** located in the event channel.

Proxy objects are the handles used by the event consumers and event suppliers to communicate with one another through the event channel. As illustrated in Figure 6-4, an event supplier is represented in the event channel by a **proxy supplier**. Likewise, an event consumer is represented by a **proxy consumer**. The “connection” between a supplier and consumer of stream events is created by providing an event supplier with the object reference of a proxy consumer and an event consumer with the object reference of a proxy supplier.

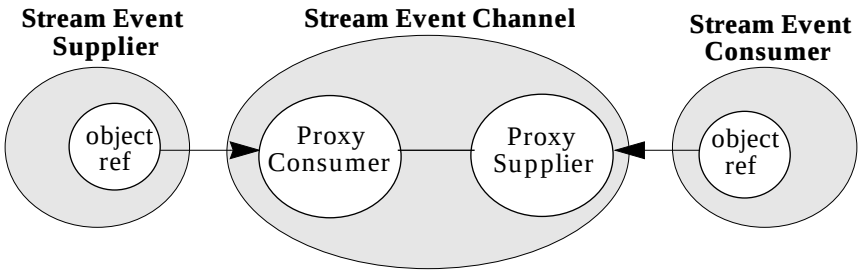


Figure 6-4: Proxy Objects used in the Event Channel

Example This portion of code attaches the event consumer to the proxy supplier in the stream event channel.

Obtain the administration object, **TypedConsAdm**, for the typed event channel (*orTEC*) returned in *Get the Stream Event Channel* on page 6-5. The **TypedConsAdm** object implements the **obtain_typed_push_supp()** function you use to return the proxy object (*orTPS*) that represents the stream event supplier in the event channel.

```
yoenv                          evEC;
yoEnvInit(&evEC);

yeceTeca_TypedConsAdm          orTCA = NULL;
yeceCa_ProxyPushSupp           orTPS = NULL;

orTCA = (yeceTeca_TypedConsAdm)
   yeceTeca_TypedEventChannel_for_consumers(orTEC, &evEC);

orTPS = (yeceCa_ProxyPushSupp)
   yeceTeca_TypedConsAdm_obtain_typed_push_supp(orTCA, &evEC,
    (char *)mzs_ec__id);
```

Save a pointer to the proxy supplier in the **eventChannel** structure.

```
ecP->orTPS = orTPS;
```

Finally, use the **ProxyPushSupp_connect_push_cons()** function to connect our event consumer (*orCUE*) to the proxy supplier in the event channel.

```
yeceCa_ProxyPushSupp_connect_push_cons(orTPS, &evEC,
      (struct YCyecec_PushCons *)orCUE);
```

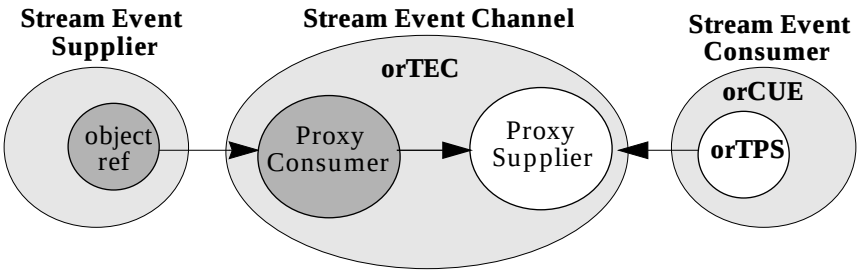


Figure 6-5: Event Consumer Connected to Event Channel

Set the boolean in the **eventChannel** structure to *TRUE* to indicate that the event consumer is attached to an event supplier and ready for use.

```
ecP->valid = TRUE;
```

Clean up the resources used to set up this connection.

```
yoRelease(orTEC);
yoRelease(orTCA);

yoEnvFree(&evEC);     /* Free the environment - we're done with it. */
```

Implement the sendEvent() Operation

Implement the **sendEvent()** operation described in the **mzs::ec** interface.

When the stream service generates an event, it invokes the **sendEvent()** method, passing the name of the event, on the proxy consumer in the event channel. The proxy consumer pushes the **sendEvent()** request to the proxy supplier, which places the request on the *ImplCliQueEC* you created to hold incoming events. The proxy supplier then pushes the request to the event consumer when it is ready to receive the request.

Example This implementation of **sendEvent()** uses a **CircuitToString()** function (defined in the sample code listed on page H-53) to print all the events it receives to **stdout**. You can create more sophisticated implementations that take specific actions depending on the events received.

```
static void SendEvent_i(/* ARGUSED */mzs_ec or, /* ARGUSED */yoenv* ev,
                        mzs_event* ptev)
{
  switch (ptev->_d)
  {
  case mzs_evTypeAlloc:
  {
    char            dsBuf[128];
    mzs_evAlloc    *event_allocP = &(ptev->_u.event_alloc);

    CircuitToString(&event_allocP->cliCircuit, dsBuf);
    (void)printf("Alloc: downstream=%s capMask=%d maxBitrate=%d\n",
                 dsBuf,
                 event_allocP->capabilities,
                 event_allocP->maxBitrate);
  }
    break;

  case mzs_evTypePrepare:
  {
    mzs_evPrepare *event_prepareP = &(ptev->_u.event_prepare);

    (void)printf("Prepare: cookie=%s\n",
                 event_prepareP->cookie);
  }
    break;

  case mzs_evTypeSeqPrepare:
  {
    mzs_evSeqPrepare *event_seqPrepareP = &(ptev->_u.event_seqPrepare);
    uword counter;

    (void)printf("SeqPrepare: cookies=");
    for(counter = 0; counter < event_seqPrepareP->cookies._length;
        ++counter)
    {
      mkd_assetCookie *this =
        (mkd_assetCookie *)(event_seqPrepareP->cookies._buffer[counter]);
      (void)printf("%s%s", counter ? " " : "", this);
    }
    (void)putchar((int)'\n');
  }
    break;

  case mzs_evTypePlay:
  {
    char startBuf[64];
    char endBuf[64];
    size_t ourLen = 64;
    mzs_evPlay *event_playP = &(ptev->_u.event_play);
```

```
    if(! (mkduPos2Str(startBuf, &ourLen, &event_playP->startPos)))
      DISCARD strcpy(startBuf, "???");
    ourLen = 64;
    if(! (mkduPos2Str(endBuf, &ourLen, &event_playP->endPos)))
      DISCARD strcpy(endBuf, "???");

    (void)printf("Play: startPos=%s endPos=%s playRate=%d bitRate=%d\n",
             startBuf, endBuf,
             event_playP->playRate,
             event_playP->bitRate);
  }
    break;
case mzs_evTypeFinish:
  {
    mzs_evFinish *event_finishP = &(ptev->_u.event_finish);
    (void)printf("Finish: loopCancel=%d\n",
             event_finishP->loopCancel);
  }
    break;

  case mzs_evTypeDealloc:
  {
    char dsBuf[128];
    mzs_evDealloc *event_deallocP = &(ptev->_u.event_dealloc);

    CircuitToString(&event_deallocP->cliCircuit, dsBuf);
    (void)printf("Dealloc: downstream=%s\n", dsBuf);
  }
    break;

  case mzs_evTypeDenial:
  {
    mzs_evDenial *event_denialP = &(ptev->_u.event_denial);
    uword counter;

    (void)printf("Denial: cookies=");
    for(counter = 0; counter < event_denialP->cookies._length;
        ++counter)
    {
      mkd_assetCookie *this =
        (mkd_assetCookie *)(event_denialP->cookies._buffer[counter]);
      (void)printf("%s%s", counter ? " " : "", this);
    }
    (void)putchar((int)'\n');
  }
    break;
  }

  /* We flush stuff to stdout just to make sure we don't lose anything */
  /* if we are killed at an inopportune time and stdout was redirected */
  /* to a file. */
  fflush(stdout);
}
```

Process Events

Use the **ysSetIdler()** function to indicate that the **IdleFunc()** function is to be called each time this event consumer is idle. The **IdleFunc()** function uses the **ysSvcPending()** function to determine if any **sendEvent()** requests are on the *ImplCliQueEC* queue in the event channel. If so, it uses the **ysSvcAll()** function to service all of the pending **sendEvent()** requests.

```
ysSetIdler("IdleFunc()", IdleFunc, (void *)ecP);

static void IdleFunc(void *usrp, const ysid *exid, void *arg,
                         size_t argsz)
{
  eventChannel  *ecP = (eventChannel  *)usrp;

  if(ysSvcPending(ecP->ImplCliQueEC))
    ysSvcAll(ecP->ImplCliQueEC);
}
```

Detach Consumer from the Stream Event Channel

When you are finished listening for events, detach the event consumer from the event channel.

Example Remove the event consumer from the idle queue and use the **yoQueDestroy()** function to destroy the *ImplCliQueEC* queue you created to hold the incoming events.

```
ysSetIdler("IdleFunc()", (ysHndlr) 0, (dvoid *) 0);

/* If we created a queue for handling events, clean it up */
if(ecP->ImplCliQueEC != NULL)
{
  yoQueDestroy(ecP->ImplCliQueEC);
  ecP->ImplCliQueEC = NULL;
}
```

Detach the event consumer from the event supplier by invoking the **yeceCa_ProxyPushSupp_disconnect_push_supp()** function on the proxy supplier object. Then call **yoImplDeactivate()** to deactivate the event consumer.

```
yoenv  evEC;

if(ecP->valid == FALSE)
  return;

yoEnvInit(&evEC);

yseTry
{
  yeceCa_ProxyPushSupp_disconnect_push_supp(ecP->orTPS, &evEC);

  yoImplDeactivate(mzs_ec__id, mzs_ec_implid);
```

Use **yoRelease()** to release the object references to the event supplier (*orTPS*) and our implementation of the event consumer (*orCUE*). Then free the resources used by the Media Net environment.

```
  if(ecP->orTPS != NULL)
  {
     yoRelease(ecP->orTPS);
     ecP->orTPS = NULL;
  }
  if(ecP->orCUE != NULL)
  {
    yoRelease(ecP->orCUE);
    ecP->orCUE = NULL;
    }
  }
  yseCatchAll
  {
    yoEnvFree(&evEC);
    yseRethrow;
  }
  yseEnd;

  yoEnvFree(&evEC);
}
```

CHAPTER

7

Extending Video Encoders for Real-time Feeds

The Video Encoding Standard (**VES**) API allows vendors of video encoders and servers to create products that operate together without proliferating unnecessary vendor-specific, non-value-added solutions. With the VES API, video encoding products can store digital video directly into any commercial video server so that consumers can simultaneously view this same content from their client devices in near real-time.

Note The VES API supersedes the MKCFW API described in previous versions of the *Oracle Video Server Developer's Guide*.

To deliver video content to a client, the stream service must have access to certain **metadata** that describes the content; see *Content Metadata* on page 7-4 for more information. This content metadata is stored in a **tag file** in the Media Data Store (MDS), and is used by the stream service to determine how to stream the encoded video content to a client. The metadata provides a map between the byte positions within a piece of content in MDS and the time offsets required by the stream service to locate specific positions within the content when performing visual fast forward, rewind, or seeking operations.

The real-time feed service, **vsfeedsrv**, enables you to load content onto the server in a single step, possibly from a live source such as a video camera or satellite feed. This contrasts with the **vstag** utility, which reads an encoded video stream and generates a tag file for the video data.

This chapter describes how to extend a video encoder to generate content metadata and deliver it to the server along with the encoded video data so that the video is ready for immediate real-time playback to OVS clients. This involves extending your encoder to construct the content metadata as it encodes the video content and to send the encoded content and metadata to the OVS real-time feed service, **vsfeedsrv**, with a frequency that allows for real-time throughput to clients.

About Real-time Encoders

The real-time feed extensions described in this chapter are provided by the **vesw.h** file, which is a C wrapper around the **ves** interfaces defined in the **vesm.idl** file. The purpose of the **vesw** wrapper is to ensure future compatibility should the **ves** interfaces change. The **vesw** functions are used by the program that encodes the video source. If the video is encoded by a hardware device, you generally need to modify the driver for that device.

As illustrated in Figure 7-1, your encoder must supply the **vesw** interface with digitized video content and the metadata described in *Content Metadata* on page 7-4. The **vesw** functions then send the encoded content and metadata to the video server.

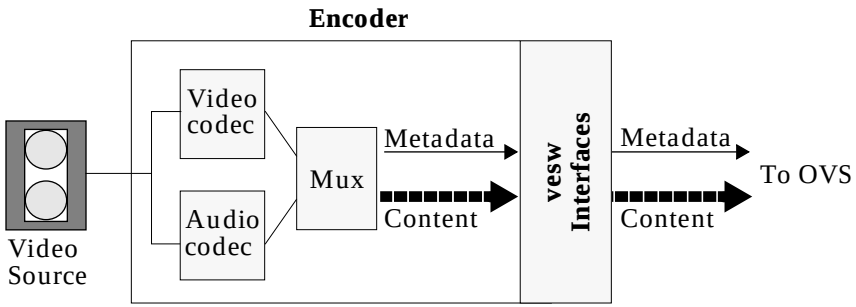


Figure 7-1: Encoder Supplies Content and Metadata to the vesw Interface

As illustrated in Figure 7-2, the output from your real-time encoder is transported to the OVS real-time feed service, **vsfeedsrv**, which creates two types of files in the Media Data Store:

- **Content files** store the digitized video content in a streamable format, such as MPEG or Raw Key format.
- **Tag files** contain the metadata that describes to the stream service how to stream the digitized video content, as well as how to locate specific frames in the content. (If you are using OVS with a database, the logical content service also stores some of the metadata in the database, as described in *How Content Information is Stored* on page 3-7.)

The real-time feed feature uses an **encoder push** model, which means that the encoder calculates when to send data and how much data to send at any given time. If any tags arrive before the corresponding content, the server makes sure that the tags are not made available to the stream service until the content has arrived.

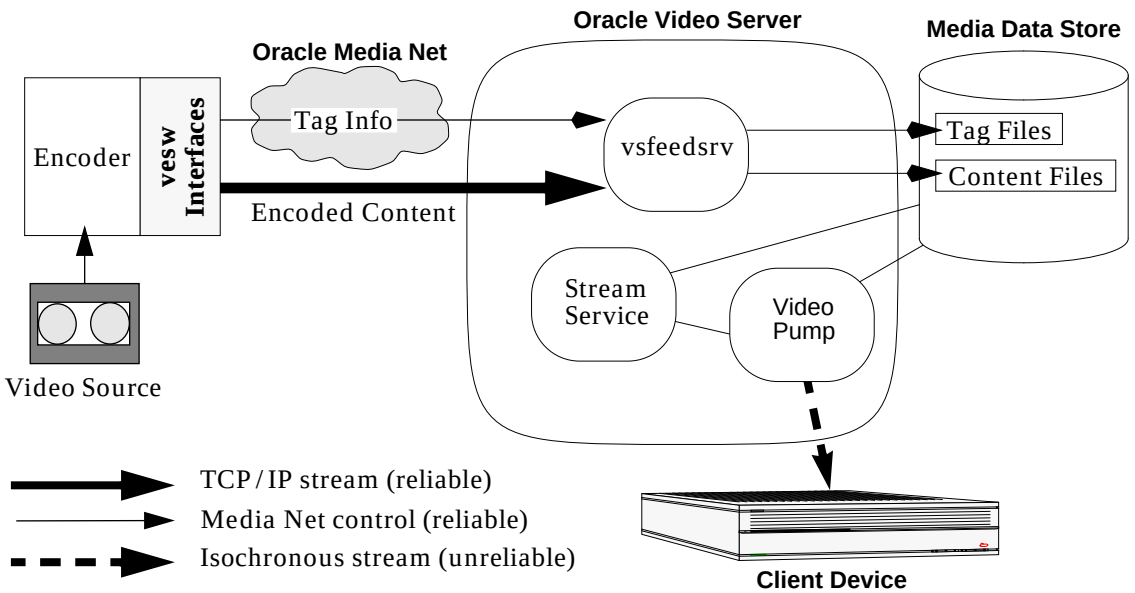


Figure 7-2: Encoding and Delivering Video in Real-time to a Client

Content Metadata

There are two kinds of content metadata required by the Oracle Video Server:

- Header Information — information about the general characteristics of the entire stream
- Tag Information — information about specific tagged frames in the video

Header Information

Header information has both a generic and a compression-specific component. The generic header information describes the compression format of the content, as well as other information such as the dimensions of the video picture and playback frame rate. This generic header information is often used as selection criteria by clients. For instance, a client that can only decode MPEG-1 program streams is not interested in an MPEG-2 transport stream that contains MPEG-2 video. Likewise, the resolution of the video or its bit rate may exclude certain clients. A consequence of this is that it is invalid, for example, to change the pixel resolution of the screen in the middle of the video.

The compression-specific portion of the header describes settings that are specific to the compression format.

Header information is described in detail in *Header Fields* on page E-1.

Tag Information

The server uses tag information to perform seeking and visual fast forward / rewind, which are also called **scanning** operations. For most compression formats, tag information corresponds to selected frames of encoded video. In the sample encoder described in this chapter, each tag is associated with one frame of the video, although not every frame needs to be tagged.

Video content encoded using the MPEG-1, MPEG-2, and Raw Key Compression video formats consists of three types of frames. The **key frames** or **MPEG Intraframes** (I-frames) are the most important frames to tag, as they are used as random access points to the video. The **predictive frames** (P-frames) and **bias frames** (B-frames) may also be tagged to provide smoother scanning operation. You can tag any type of frame and any number of frames, depending on your needs.

Although the concept of frames does not usually exist in audio-only formats, tags in the form of **pseudo-frames** can be used to specify the available access points in the audio data.

Tag information for the various compression formats is described in detail in *Tag Fields* on page E-4.

How the Metadata is Generated

The encoder constructs a **ves::hdr** structure containing the header information and a separate **ves::tag** structure for each tagged frame. The **vesw** functions then send the **ves::hdr** structure, followed by the **ves::tag** structures interleaved with the encoded content, to the video server.

For example, Figure 7-3 illustrates a sample raw key frame encoder. This encoder tags every 10th frame in a sequence consisting of key frame, four bias frames, key frame, and so on. Frame 1 is a key frame, Frames 2 through 5 are bias frames, Frame 6 is a key frame, and so on until the end of the encoded content. The frame rate is 15 frames per second.

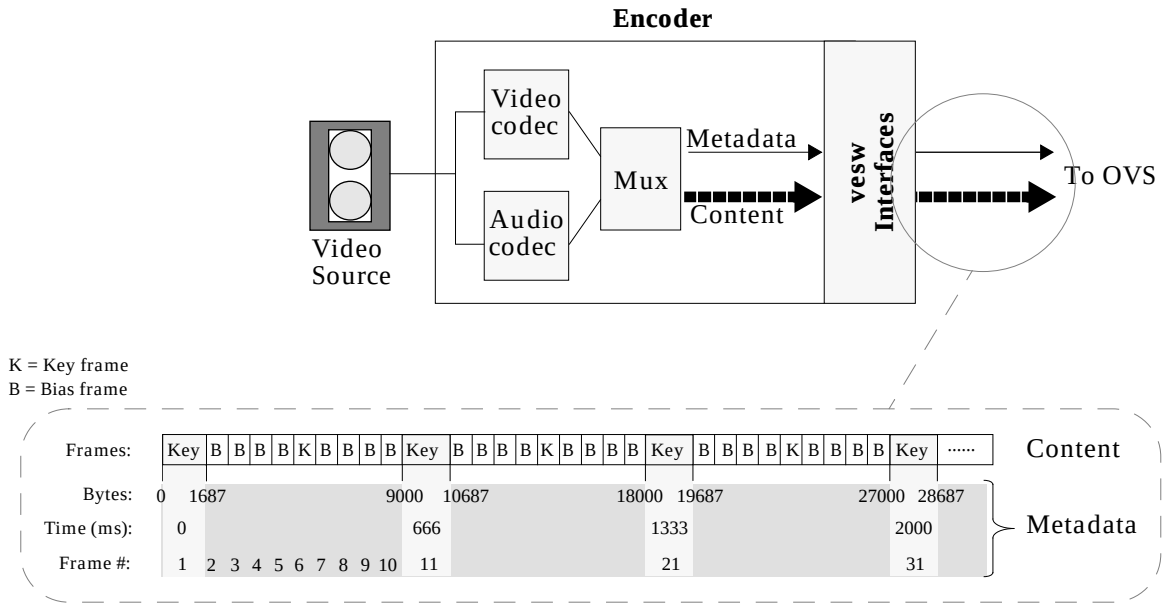


Figure 7-3: Content and Metadata from the Encoder

The metadata generated for this content consists of:

- Frame 1, a 1687-byte key frame at byte offset 0 (0 ms)
- Frame 11, the third 1687-byte key frame at byte offset 9000 (666 ms)
- Frame 21, the fifth 1687-byte key frame at byte offset 18000 (1333ms)
- Frame 31, the seventh 1687-byte key frame at byte offset 27000 (2000ms)
- and so on.

These values are calculated in *Construct Tag Information* on page 7-11.

Note Encoders generate and tag frames as needed. The frequency with which key frames are generated is encoder-specific. In a real-life encoder, it is unlikely that the key frames and bias frames would be generated in any sort of regular pattern, such as the pattern used in the previous example.

Writing a Real-time Encoder

The following is a summary of the tasks involved in extending your encoder to handle real-time feeds.

Note The sample encoder used in the following sections is the same raw key frame encoder described in *How the Metadata is Generated* on page 7-5. The code samples are based on the **vesdemo.c** sample application listed on page H-64, which delivers content metadata and encoded video data to the feed server so that the video is streamed in real-time to OVS clients. These examples are used only to illustrate the general procedures for writing a real-time encoder, and are not intended to be incorporated directly into your applications.

Here is a summary of the tasks described in this chapter:

Task	Action	Details on Page
1.	**Initialize and Set Up**	
	• Initialize Media Net	7-7
	• Establish Real-time Feed Settings	7-8
	• Construct Header Information	7-8
2.	**Open Connection to Video Server**	7-10

Task	Action	Details on Page
3.	**Construct Tag Information**	7-11
4.	**Send Content and Tags to the Server**	7-13
5.	**Shut Down Feed**	7-14
6.	**Terminate Media Net**	7-14

Initialize and Set Up

The **vesw** interface makes most of the details related to using Media Net opaque. This is particularly useful when initializing Media Net, as described in *Initialize Media Net*, and when binding to the real-time feed interface in the server, as described in *Open Connection to Video Server* on page 7-10.

Initialize Media Net

Call **veswInit()** to initialize the Media Net object environment. This function takes care of all the details related to initializing Media Net, as described in *Initialize Media Net* on page 1-11. You do not need to call **veswInit()** if you initialize the **ys**, **mn**, and **yo** layers yourself.

Example

```
ub1        ysCtx_demoCmds[SYSX_OSDPTR_SIZE];
veswLayer  result;

result = veswInit(ysCtx_demoCmds, argv[0]);
if (result != veswLayerSuccess)
{
  yslPrint("veswInit failed\n");
  failed = TRUE;
}
```

See the description for the **veswLayer** type on page B-94 for information on any errors that may occur during initialization.

Establish Real-time Feed Settings

The real-time feed service must know certain information prior to establishing a connection with the video server, such as the compression format of the content, dimensions of the video picture, pixel aspect ratio, bit rate, and frame rate.

Example The following values are based on the sample raw key frame encoder that was described under *How the Metadata is Generated* on page 7-5.

```
format = ves_formatRKF;  /* OSF is one type of raw key format *
heightInPixels = 240;    /* picture size is 352 * 240 pixels */
widthInPixels = 352;
pelAspectRatio = 10950;  /* encoded pixels slightly narrowed */
bitrate = 108000;        /* bitrate of content */
frameRate = 15000;       /* frames per kilosecond; 15 fps */
kpm = 90;                /* # of tagged key frames per minute */
ppm = 0;                 /* # of tagged predictive frames per minute */
bpm = 0;                 /* # of tagged bias frames per minute */
```

Construct Header Information

Create a **ves::hdr** structure that describes the encoded content. The values used to build the **ves::hdr** structure are typically supplied by the encoder. Header information is described in detail in *Header Fields* on page E-1.

Example The content used in the following example is playable raw key frame video (**ves_formatRKF**) with audio. The size of the picture is 352 by 240 pixels, and the frame rate is 15 frames per second.

First, the generic header fields:

```
ves_hdr    vhdr;                /* ves header */

memset((dvoid *)vhdr, 0x0, sizeof(ves_hdr));

vhdr->vend = "Real-time Feed Demo v1.1";

vhdr->fmt = ves_formatRKF;      /* OSF is one type of raw key format */

/* Both vidCmp and audCmp are arbitrarily set since this information is
not directly referenced by the video server */
#define DEMO_VIDCMP     "video codec"
#define DEMO_AUDCMP     "audio codec"

vhdr->vid._maximum = vhdr->vid._length = (ub4)strlen(DEMO_VIDCMP);
vhdr->vid._buffer =
    (ub1 *)ysmGlbAlloc((size_t)vhdr->vid._maximum, "vidCmp");
memcpy((void *)vhdr->vid._buffer, DEMO_VIDCMP,
    (size_t)vhdr->vid._maximum);
vhdr->aud._maximum = vhdr->aud._length = (ub4)strlen(DEMO_AUDCMP);
vhdr->aud._buffer =
    (ub1 *)ysmGlbAlloc((size_t)vhdr->aud._maximum, "audCmp");
memcpy((void *)vhdr->aud._buffer, DEMO_AUDCMP,
    (size_t)vhdr->aud._maximum);

vhdr->heightInPixels = 240;     /* picture size is 352 * 240 pixels */
vhdr->widthInPixels = 352;
vhdr->pelAspectRatio = 10950;   /* encoded pixels slightly narrowed */
vhdr->frameRate = 15000;        /* frames per kilosecond; 15 fps */
vhdr->compData._d = ves_formatRKF;
```

The compression-specific header information is stored in a structure selected by the **compData** field in the **ves::hdr** structure. Every time the stream is played from a position other than **mkdBeginning**, this compression-specific information is prepended to the stream. The decoder recognizes this initialization data and uses it to decode the stream.

In the following example, encode raw key frames and supply the raw key frame initialization data as described in *Raw Key Compression Specific Header Fields* on page E-3. The **ves::rkfHdr** structure indicates that the raw key frame initialization data, **initData**, is prepended to the video stream on each seek operation.

```
ub1      *initData;      /* arbitrary init data */
size_t   initLen;        /* length of init data, in bytes */

vhdr->compData._u.rkf.initData._maximum =
                 vhdr->compData._u.rkf.initData._length = (ub4)initLen;
vhdr->compData._u.rkf.initData._buffer =
                 (ub1 *)ysmGlbAlloc(initLen, "initData");
memcpy((void *)vhdr->compData._u.rkf.initData._buffer,
                 (void *)initData, initLen);
```

Open Connection to Video Server

Use the **veswNewFeed()** function to create a new feed session with the video server and return a real-time feed context. During this creation, the session can be configured as either a “one-step encode” of fixed length content, such as a commercial or a movie, or a continuous real-time feed, where the encode continues indefinitely.

Example In this example, the **veswNewFeed()** function indicates the resource requirements of the feed and passes the **ves::hdr** structure created in *Construct Header Information* on page 7-8, as well as a *duration* value that specifies the length of the incoming feed.

The **veswNewFeed()** function returns a **veswCtx** structure that identifies this encoding session. The **veswCtx** structure is opaque at this level, but it holds the tightly bound object reference to the real-time feed service in the OVS and the transport stream.

```
veswCtx    *veswCtx_demoCtx;   /* our feed context */
char       *lgName;            /* name of the feed */
ub4        seconds = 300;      /* length of content to buffer, 5 min */
ves_time   duration;           /* duration of incoming feed */
boolean    continuousFeed;     /* continuous feed or one-step encode */
```

```
duration._d = ves_timeTypeSMPTE;
duration._u.ves_timeSMPTE.hour = (seconds / 3600);
duration._u.ves_timeSMPTE.minute = (ub1)(seconds / 60);
duration._u.ves_timeSMPTE.second = (ub1)(seconds);
duration._u.ves_timeSMPTE.frame = 0;
continuousFeed = TRUE;
```

```
failed = veswNewFeed(veswCtx_demoCtx, lgName, continuousFeed, bitrate,
                     &duration, kpm, ppm, bpm, &vhdr);
```

The *bitrate* argument is the bit rate of the compressed stream; the bit rate at which the stream should be delivered to viewing clients.

Note The **veswPrepFeed()** function, in combination with the **veswSendHdr()** function, has the same effect as **veswNewFeed()**. The **veswPrepFeed()** function does not send any header information upon opening a connection to the server.

Construct Tag Information

Construct the tag data and send it along with the video content data to the server. For optimum throughput, handle the encode and tag operations in a separate thread from the send operations. Once the content encoding and construction of tag data is complete, the “prepare thread” can signal the “send thread” to send the data to the server.

The **ves::tag** structure you construct for each tagged frame must include a time stamp, the byte offset of the frame from the beginning of the encoded stream, and the byte length of the tagged frame, as illustrated in Figure 7-4. Such values are normally provided by the encoder; however, they are set in this example to simulate the output from a simple encoder. Tag information for the various compression formats is described in detail in *Tag Fields* on page E-4.

Example Figure 7-4 shows the tagged frames and corresponding metadata that is generated by this sample encoder and used to construct the **ves::tag** structures.

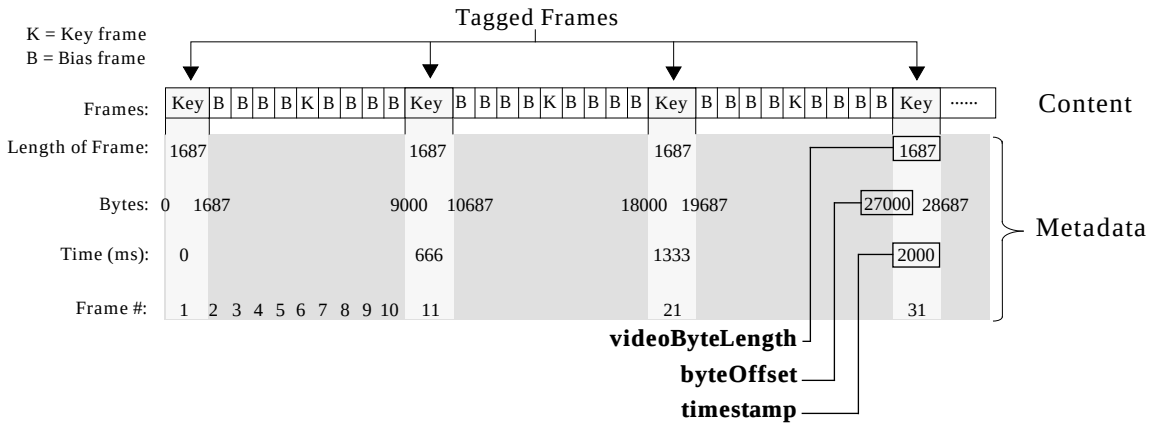


Figure 7-4: Tag Information

When creating a new feed session with the video server, the **veswNewFeed()** function must provide an estimate of the number of tagged frames that will be sent per minute. When in doubt, estimate high. The **kpm**, **ppm**, and **bpm** values are used by the server to reserve the appropriate amount of disk space to hold the key, predictive, and bias tag information, respectively. In this example, three tagged key frames are sent every 2 seconds, which works out to a **kpm** value of 90 key frames each minute.

Note The optimum tagging frequency is between 1 and 3 tags per second. Tagging more frequently than 3 tags every second has minimal benefit, and it uses up bandwidth and disk space. Tagging less frequently than one tag every second makes visual fast forward appear jumpy, although it may be appropriate depending on your application.

Example This example describes how to fill in the fields of a single **ves::tag** structure for the key frame at byte offset 27000, as illustrated in Figure 7-4.

First, define the **frameType** and **videoByteLength** for the tagged frame. In this example, the **videoByteLength** for key frames is 1687. The **videoByteLength** values in this example are arbitrary. The actual key frame sizes you use are determined by your encoder.

```
ves_tag    tag;                              /* ves tag structure */

tag.frameType = ves_frameRawKey;             /* raw key format */
tag.videoByteLength = 1687;                  /* bitrate / 64 */
```

Set the **byteOffset** for this tagged frame to the byte offset from the beginning of the stream.

```
tag.byteOffset = 27000;                      /* bytes */
```

Set the **timestamp** for this tagged frame to the time to display this frame.

```
tag.timestamp._d = ves_timeTypeMillisecs;
tag.timestamp._u.ves_timeMS = 2000;          /* milliseconds */
```

The compression-specific tag information is stored in a structure selected by the **compData** field in the **ves::tag** structure. Specify that this tagged frame uses the raw key frame format (**ves_formatRKF**), which is defined by the **ves::rkfTag** structure. The **rkf.nothing** value indicates that there is no compression-specific initialization data.

```
tag.compData._d = ves_formatRKF;
tag.compData._u.rkf.nothing = NULL;
```

Repeat for each **ves::tag** structure sent to the server.

Send Content and Tags to the Server

Use the **veswSendData()** and **veswSendTags()** functions to send content and tags to the server. The order in which tags and content are sent is not important and could be reversed, or even skewed by any amount.

Example In this example, delay the stream to maintain the average bit rate over the course of the feed. This delay simulates the time required by the "prepare thread" to digitize the next chunk of video and prepare the tag data. The **veswIdle()** function prevents the Media Net connection with the server from timing out during the delay.

```
ub1     ysCtx_demoCmds[SYSX_OSDPTR_SIZE];
sysb8   tmp1, elapseTime, baseTime; /* in micro-seconds */
sysb8   targetTime;

veswIdle(ysCtx_demoCmds);
ysClock(&tmp1);
sysb8sub(&elapseTime, &tmp1, &baseTime);
if (sysb8cmp(&elapseTime, <, &targetTime))
{
  do
  {
    veswIdle(ysCtx_demoCmds);
    ysClock(&tmp1);
    sysb8sub(&elapseTime, &tmp1, &baseTime);
  } while (sysb8cmp(&elapseTime, <, &targetTime));
}
else
  yslPrint("Warning: falling behind, video playback may be glitchy\n");
```

Repeatedly call **veswSendData()** to send the buffer of video data to the server until the number of bytes, **alen**, has been reached or exceeded.

```
size_t    alen;
veswCtx   *veswCtx_demoCtx    /* our feed context */
ub1       *buf_demoCtx;       /* buffer of the content */

if ((alen > 0) && (!failed))
{
  if (veswSendData(veswCtx_demoCtx, (dvoid *)buf_demoCtx, alen))
  {
    yslError("Something bad happened in SendData - %s\n",
             veswLastError(veswCtx_demoCtx));
    failed = TRUE;
  }
}
```

Create an allocated buffer of tag information (**tagBuf**), an array of **ves::tag** structures, to be sent to the server. Call **veswSendTags()** to send the **tagBuf** buffer to the server. The **tagnum** parameter indicates the number of **ves::tag** structures in the buffer.

```
veswCtx  *veswCtx_demoCtx     /* our feed context */
ves_tag  *tagBuf;             /* buffer holding an array of tags */
ub4      tagnum;              /* number of tags in tagBuf */

if ((tagnum > 0) && (!failed))
{
  if (veswSendTags(veswCtx_demoCtx, tagBuf, tagnum))
  {
    yslError("Something bad happened in SendTags - %s\n",
             veswLastError(veswCtx_demoCtx));
    failed = TRUE;
  }
  ysmGlbFree((dvoid *)tagBuf);
}
```

Shut Down Feed

After the content has been sent to the server, call **veswClose()** to free the context and release related resources on the server.

Example

```
char      baseName_demoCtx[SYSFP_MAX_PATHLEN];
veswCtx   *veswCtx_demoCtx    /* our feed context */

yslPrint("Closing feed: %s\n", baseName_demoCtx);
failed = veswClose(veswCtx_demoCtx);
if (failed == TRUE)
  yslPrint("problem during close\n");
```

Terminate Media Net

After the context has been freed, call **veswTerm()** to shut down the Media Net environment and terminate the Media Net session.

Example

```
ub1        ysCtx_demoCmds[SYSX_OSDPTR_SIZE];

result = veswTerm(ysCtx_demoCmds);
if (result != veswLayerSuccess)
{
  failed = TRUE;
  yslPrint("irregularity during termination\n");
}
```

CHAPTER

8 Accessing Files in the Media Data Store

All video server content is stored in the Oracle Media Data Store (MDS). Any OVS service that accesses media data must interact with the MDS.

The MDS stores content, such as video files, audio files, and BLOB (Binary Large OBject) files, on a collection of disks that are organized into groups called **volumes**. The MDS consists of one or more servers that can manage multiple volumes. Volumes are similar to physical file systems but also provide striping, RAID (Redundant Arrays of Inexpensive Disks) protection, hot swapping, and hot sparing. Each volume can support hundreds of disks, and each MDS file is striped across all available disks in the volume.

The MDS has many features that are covered in *Introducing Oracle Video Server*. This chapter focuses on the MDS concepts and operations that are of interest when writing OVS clients and services that open, close, and manipulate files in the MDS.

MDS File Names

A file name passed to the MDS can refer to either a host file or an MDS file. The caller usually doesn't need to know whether the data resides on the host or in MDS, but can restrict the access to one or the other, if necessary.

Native MDS file names are different from host file names; MDS files always begin with the reserved prefix **/mds**. Any file name not beginning with this prefix is assumed to refer to a host file. Native MDS file names are always of the form:

/ mds / *vol* / *file*

where *vol* is the volume name and *file* is the file name.

MDS file names are **case insensitive** but **case retentive**: The case of a file name is maintained by MDS, but never used in file name comparisons.

Interfaces to the Media Data Store

The public interfaces to the Media Data Store are described by these files:

- **mds.h** must be included by any module that wants to access files contained within MDS.
- **mdsnm.h** provides routines to manipulate MDS file names, including shell-like wildcard expansion.
- **mdsblob.h** allows clients to retrieve unstructured and untyped data, known as **BLOBs** (Binary Large OBjects), from MDS.
- **mdsex.h** and **mdsex.idl** define public MDS exceptions.

Applications that need to read BLOB data from the OVS server use the functions in **mdsblob.h**. Applications that need to create, modify, or destroy MDS content, such as BLOBs or video files, use the functions in **mds.h** and **mdsnm.h**.

When using the **mdsblob.h** functions, the content is described using **mkd::assetCookie**s, so you don't need to know anything about how the BLOB data is stored in the MDS. However, when using the functions in **mds.h** and **mdsnm.h**, you need to reference the file by its full **/mds** pathname, as described in *MDS File Names*.

Using the MDS Interface

The MDS interface looks much like any standard file system. You can open and close, read and write, create and delete files in MDS in much the same way as you would when using any other file system. The primary difference between MDS and a standard file system is that once MDS files are created, they cannot grow in length (though they can be truncated).

Caution When using the MDS interface, keep in mind that there are no restrictions on who can read and write MDS files. Data stored elsewhere, such as the metadata in the database used by the logical content service, may point to MDS files. Haphazard or unintentional modification of MDS content can leave the OVS system in an inconsistent state.

The MDS interface lets clients interact with MDS files to perform:

- Operations for Opening and Closing MDS Files (for example, **mdsOpen()** and **mdsClose()**)
- MDS File Operations (for example, **mdsCreate()** and **mdsRemove()**)
- MDS I/O Operations (for example, **mdsRead()** and **mdsWrite()**)
- MDS File Attribute Operations, for example, get information about the length or create time of the file

Clients can also use the MDS interface to transparently access their local (host) file system. Unless you specify restrictions that mandate an MDS file, a file name can map to either an MDS file or a file on the host system. If you attempt an operation that is not supported on host files, the exception MDS_EX_HOST is raised. File operations, such as file deletion, locking, and so on, are generally not available on host files. Additionally, any host I/O request will block the client process on most platforms.

Example The following sample code opens, reads, then closes an MDS file.

The **mtuxInit()** function initializes Media Net, and **mdsInit()** initializes MDS. The **mdsOpen()** function opens an MDS file, named **/mds/vol/video.mpg**, at the default bit rate (*MdsDfltBitRate*) for sequential reading (*MdsFlgSeq*) via the remote file server (*MdsFlgPxy*). The **mdsOpen()** function returns a file pointer (*fp*) that is used by the **mdsRead()** function to read the contents of the file into a buffer (*buf*). The **yslPrint()** function displays the contents of *buf* and the **mdsSeek()** function moves the offset pointer to the beginning of the file. The **mdsClose()** function closes the file, and **mdsTerm()** closes the connection to MDS and frees the associated resources.

```
ub1         osdCtx[SYSX_OSDPTR_SIZE];
mdsFile    *fp;
ub2         buf;
size_t      size;

if (mtuxInit(osdCtx, argv[0], (mnLogger)NULL) != mtuxLayerSuccess)
  exit(1);
mdsInit();

fp = mdsOpen("/mds/vol/video.mpg",
             MdsDfltBitRate,
             MdsFlgSeq | MdsFlgPxy);

while ((size = mdsRead(fp,(dvoid *)&buf, sizeof(buf))) == sizeof(buf))
  yslPrint("%d\n", buf);

if (!mdsSeek(fp, sysb8zero, (sysb8 *)0))
  yslError("Error seeking to beginning of file\n");

mdsClose(fp);

mdsTerm();
mtuxTerm(osdCtx);
```

Synchronous and Asynchronous MDS Functions

Many of the functions in the MDS API are available in synchronous and asynchronous forms.

- a synchronous function completes its operation before returning and may block for some length of time.
- an asynchronous function returns immediately without blocking. When the asynchronous function completes, it returns an event that signals its completion. Asynchronous functions are identified by an **_nw** (no wait) suffix.

When using asynchronous functions, you specify a **ysevt** event to be triggered when the function has completed. You can use the functions in **ysv.h**, such as **ysSemSynch()** or **ysSemAndW()**, to synchronize the completion of the asynchronous function with other operations.

Example There is a synchronous and an asynchronous version of the **mdsRename()** function:

```
void mdsRename (CONST char *oldNm, CONST char *newNm );
ysevt *mdsRename_nw (CONST char *oldNm, CONST char *newNm, ysevt *uevt);
```

The synchronous version, **mdsRename()**, must complete before returning and allowing the application to invoke other functions. The asynchronous version, **mdsRename_nw()**, returns immediately, so the application can invoke other functions to perform operations in parallel while waiting for **mdsRename_nw()** to complete.

The following code sample illustrates how you might invoke two asynchronous functions. The asynchronous function, **mdsRename_nw()**, renames a file. Before the **mdsRename_nw()** function has completed, we invoke another asynchronous function, **mdsLock_nw()**, to lock another file. The **ysSemAndW()** function blocks any further functions until the completion of the **mdsRename_nw()** and **mdsLock_nw()** functions is signaled by the events, *sem[0]* and *sem[1]*.

```
ysevt   *sem[2];

sem[0] = mdsRename_nw("/mds/vol/video.mpg",
                      "/mds/vol/foo.mpg",
                       (ysevt *)0);

sem[1] = mdsLock_nw("/mds/vol/clip.mpg", (ysevt *)0);

if (ysSemAndW((sysb8 *)0, 2, sem) != 2)
    yslError("async operations did not complete\n");

ysSemDestroy (sem[0]);
ysSemDestroy (sem[1]);
```

For more information, see the **ysv.h** header file.

Using the MDS Name Interface

The **mdsnm.h** header file provides an interface for file naming.

It provides functions for:

- checking whether a file name corresponds to a legal MDS file name
- splitting a name into path and file component (and joining two components)
- comparing a file (or volume) name in the same manner as **strcmp()**
- performing a wildcard search and returning a match context
- freeing a match context
- expanding one (or more) file names and returning a list of matches

These functions are described in *mdsnm — MDS Name Functions* on page A-40.

Expanding Wildcards

Wildcard expansion in MDS follows UNIX shell expansion semantics.

The Operating System Dependent (OSD) layer can block a process and wildcard expansion can block a thread, so real-time processes should operate only on fully-specified MDS file names. For more information, see the *Oracle Video Server Administrator's Guide and Command Reference* distributed with your video server.

To preserve shell-like usage semantics, first call **mdsnmExpand()** or **mdsnmExpandOne()** (see *mdsnmExpand mdsnmExpandOne* on page A-51) for any file name to expand wildcard characters. Since wildcard expansion is potentially expensive, it should be performed explicitly by the client and should only be performed once.

Example This example expands a wildcard file name and opens all the matching file names. The **mdsnmExpandOne()** function expands the name, **/mds/movies/***.

```
yslst *matches = NULLP(yslst);
char *filename = "/mds/movies/*";

if (!mdsnmExpandOne (filename, 0, 0))
    yslError ("No matches found\n");
else
    for (i=0; i<ysLstCount(matches); i++)
    {
       fp = mdsOpen(...).....
```

Using the MDS BLOB Interface

The **mdsblob.h** header file defines an interface that can be used by clients to transfer Binary Large OBjects, known as BLOBs, from MDS. As described in *Streams and BLOBs* on page 1-8, BLOBs are typically bitmaps, image stills, or any other data stored on the server that client applications might want to download for local access. BLOBs are transferred over a **Media Net stream**. Do not confuse a Media Net stream with an OVS video stream, which does not use Media Net.

Note The preferred method for storing BLOB data is in the Oracle Web Application Server, if present. See the Oracle Web Application Server documentation for details.

Maximizing BLOB Storage in MDS

MDS files always start on a RAID stripe boundary and extend for a multiple of the block size. Even if an MDS file contains only a few bytes of data, it will occupy a minimum of one RAID stripe (typically 32 to 256 kilobytes) of disk storage.

To make better use of MDS for smaller BLOBs, store several BLOBs in one MDS file, as shown in Figure 8-1. Then write your own content resolver to access the individual BLOBs in the MDS file. The BLOBs should be tightly packed so that there is no wasted disk space due to MDS stripe alignment restrictions. The procedures for writing a content resolver are described in *Implement the Content Resolver Interface* on page 5-9.

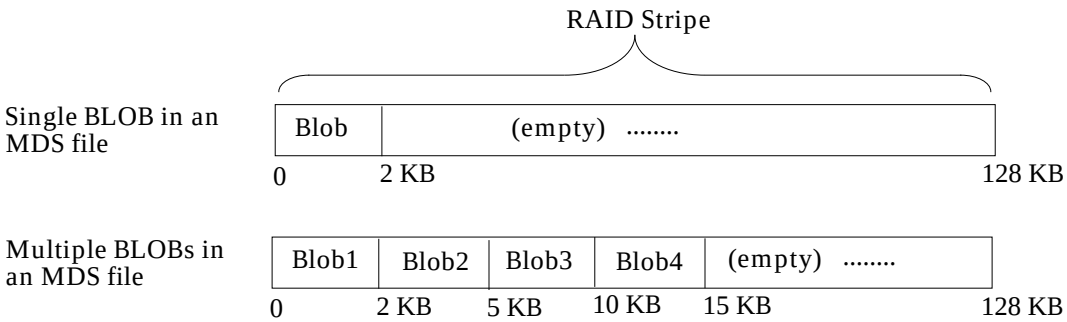


Figure 8-1: Combining Multiple BLOBs into a Single MDS File

Streaming a BLOB to a Client

Before using the MDS BLOB interface, call the **mdsBlobInit()** function to initialize the MDS BLOB library. When you have finished using the MDS BLOB library, call **mdsBlobTerm()** to free the associated resources.

When the client queries a content service, a number of **mkd::assetCookie**s are returned to represent the various pieces of content available to the client, as described in *Query the Content Service* on page 2-11. Each **mkd::assetCookie** returned from the content service may correspond to a complete MDS file, a segment of an MDS file, or multiple MDS files and / or file segments; however, this makeup is not exposed to the client. In fact, the MDS interface treats the BLOB corresponding to an **mkd::assetCookie** as one contiguous piece of content. The client can retrieve the entire BLOB by calling **mdsBlobPrepare()** or retrieve a segment of the BLOB by calling **mdsBlobPrepareSeg()**.

The client allocates memory for the BLOB as it arrives by supplying a non-blocking callback function. This scheme allows the client to control the layout of the data. When the BLOB has been retrieved, the semaphore supplied by the client for that BLOB is triggered. Any errors that have occurred during the asset name resolution or data transmission are reported to the client at this time. Unless the client waits for the semaphore to be triggered, the client should take care to periodically yield to the underlying library by calling **ysYield()**.

Each **mdsBlobTransfer()** call sets up its own Media Net port and a **Media Net stream** to the BLOB server. (BLOB data is transported over the control circuit, so no additional circuit is required.) To perform multiple BLOB transfers, the client calls **mdsBlobPrepare()** multiple times to obtain an **mdsBlob** structure for each BLOB. To initiate the BLOB transfer, call **mdsBlobTransfer()**, which takes as input one or more **mdsBlob** structures returned by **mdsBlobPrepare()** and downloads the data from each BLOB sequentially.

The client can initiate multiple simultaneous transfers by using multiple **mdsBlobTransfer()** calls, each with one or more BLOBs. However, the client may not perceive any increase in the rate of data transfer, depending on the type of network connection used and the nature of any video being simultaneously streamed to the client.

Example This code example is extracted from the **ovsdemo.c** sample application described on page H-2. The purpose of this example is to illustrate how a client application might use the MDS BLOB interface to prepare and transfer a BLOB from OVS.

Define a structure, **blobcx**, to hold the transfer context (such as the BLOB data, size, portion transferred, and so on) for BLOBs.

```
struct blobcx
{
  size_t segSz;     /* BLOB segment size */
  ub4    total;     /* total BLOB length */
  int    numSegs;   /* number of BLOB segments */
  ub1    **segs;    /* array of BLOB segments */
};
typedef struct blobcx blobcx;
```

Call the **mdsBlobInit()** function to initialize the BLOB library.

```
mdsBlobInit();
```

Use the **ysSemCreate()** function to create a semaphore that will be triggered when BLOB transfer has completed.

```
ysevt   *sem;

sem = ysSemCreate((dvoid *)0);
```

Clear any transfer state from the **blobcx** structure.

```
blobcx   bcx;

bcx.segSz = (size_t)0;
bcx.total = (ub4)0;
bcx.numSegs = 0;
bcx.segs = (ub1 **)0;
```

Use the **mdsBlobPrepare()** function to prepare the BLOB for transfer to the client. The *cookie* parameter is an **mkd::assetCookie** returned by the content service as described in *Query the Content Service* on page 2-11. The **mdsBlobPrepare()** function also returns an **mdsBlob** structure that describes the BLOB context.

Allocate memory for the returned content using the **ovsdemoBlobAlloc()** callback function described in **ovsdemo.c**. The *sem* parameter is a semaphore set by the **ysSemCreate()** function to be triggered when the transfer has completed.

```
mdsBlob  *blob;
mkd_assetCookie cookie;

/* initialize asset cookie here*/

blob = mdsBlobPrepare(cookie,
                      (CORBA_Object)0,
                      (ub4)0,
                      ovsdemoBlobAlloc,
                      (dvoid *)&bcx,
                      sem);
```

Call the **mdsBlobTransfer()** function, passing the **mdsBlob** structure returned by **mdsBlobPrepare()**, to stream the BLOB from the MDS.

```
mdsBlobTransfer(&blob, 1, (ub4)0);
```

Use the **ysSemSynch()** function to wait for the BLOB transfer operation to complete and set the semaphore, *sem* (exceptions will be raised synchronously).

```
ysSemSynch(sem, (dvoid *)0);
```

Use the **mdsBlobTerm()** function to terminate the BLOB library.

```
mdsBlobTerm();
```

Free the BLOB.

```
int      i;

for (i=0 ; i < bcx.numSegs ; i++)
    ysmGlbFree((dvoid *)bcx.segs[i]);
```

APPENDIX

A OVS Interface Reference

This appendix describes the OVS interfaces that are of interest to both client and server applications. The interfaces listed in this chapter consist of IDL-defined object interfaces and standard C header files.

Note Every IDL method invoked through the Common Object Adapter (COA) must include a reference to a remote object that implements the method's interface and a pointer to the ORB environment:

method (*object_reference*, *environment*, *other_parameters*);

The *object_reference* is the reference to the implementation of the method's interface returned by **yoBind()**, and *environment* is the environment variable initialized by **yoEnvInit()**. Any remaining parameters are specific to the method. For more information, see *Using Oracle Media Net* on page 1-10.

The IDL methods discussed in this chapter do *not* explicitly list the *object_reference* and *environment* variables. If an IDL method does not include other parameters, the syntax for the method looks like:

method ();

The following interfaces are discussed in this chapter:

Interface	Type	Description	Page
mtux	C	Media Net initialization functions for initializing and terminating Media Net.	A-4
mds	C	Media Data Store functions. Enables clients to access files contained within the Media Data Store (MDS).	A-9
mdsBLOB	C	MDS BLOB functions. Enables clients to retrieve unstructured data, or BLOBs, from MDS.	A-34
mdsnm	C	MDS Name functions. Enables clients to manipulate MDS file names, including shell-like wildcard expansion.	A-40
mtcr	IDL	Content Resolver methods. Resolves logical asset cookies to physical content segments. Defines the **mtcr::resolve** interface.	A-52
mza	IDL	Logical Content methods. Creates and manipulates logical content. Defines the interfaces: **mza::LgCtnt**, **mza::LgCtntFac**, **mza::LgCtntMgmt**, **mza::Ctnt**, **mza::CtntFac**, **mza::CtntMgmt**, **mza::CtntPvdr**, **mza::CtntPvdrFac**, **mza::CtntPvdrMgmt**, **mza::Clip**, **mza::ClipFac**, **mza::ClipMgmt**, and **mza::BlobMgmt**.	A-54
mzabi	IDL	Schedule methods. Creates and manipulates scheduled events and exporter objects. Defines the interfaces: **mzabi::Schd**, **mzabi::SchdFac**, **mzabi::SchdMgmt**, **mzabi::Exp**, **mzabi::ExpFac**, **mzabi::ExpMgmt**, **mzabi::ExpGrp**, **mzabi::ExpGrpFac**, and **mzabi::ExpGrpMgmt**.	A-92

Interface	Type	Description	Page
mzabin	IDL	Content Broadcast methods. Creates and manipulates NVOD and channel objects. Defines the interfaces: **mzabin::Chnl**, **mzabin::ChnlFac**, **mzabin::ChnlMgmt**, **mzabin::Nvod**, **mzabin::NvodFac**, and **mzabin::NvodMgmt**.	A-116
mzabix	IDL	Schedule Exporter methods. Implements the functionality of the generic exporter object. Defines the **mzabix::exporter** interface.	A-130
mzz	IDL	Session methods. Provides methods for establishing and managing client sessions in the server. Defines the **mzz::factory** and **mzz::ses** interfaces.	A-135
mzc	IDL	Virtual Circuit Manager (VCM) methods. Manages virtual circuits and their related communications channels. (Not usually used by clients.) Defines the **mzc::factory**, **mzc::ckt**, and **mzc::chnl** interfaces.	A-146
mzs	IDL	Stream methods. Enables clients to create an **mzs::stream** object. Enables clients to initiate and control streams from the server. Defines the **mzs::factory**, **mzs::stream**, and **mzs::ec** interfaces.	A-156
mzscl	C	Stream Client functions. Enables clients to establish callback functions for notification of server events.	A-180
vesw	C	Real-Time Feed functions. Enables encoders to load content onto OVS in real time.	A-183

mtux — Media Net Initialization Functions

The functions described in the **mtux.h** file enable you to initialize and terminate Media Net.

Function	Description	Page
mtuxSimpleInit	Initializes Media Net.	A-5
mtuxInit	Initializes Media Net and parses an argument map.	A-6
mtuxTerm	Terminates Media Net and frees associated resources.	A-7
mtuxVersion	Returns the version, release, and build information of the OVS library being used.	A-8

Initializes Media Net.

The **mtuxSimpleInit()** function should only be called once during the lifetime of a program. After initializing Media Net, you must call other Media Net functions, such as **ysYield()**, at least once every thirty seconds to maintain a heartbeat with **mnaddrsrv**, or Media Net will disconnect your program.

Note The **mtuxSimpleInit()** function imports the resources from the resource file designated by your YSRESFILE environment variable. (See the *Oracle Media Net Administrator's Guide* for details.)

```
mtuxLayer mtuxSimpleInit(dvoid *osdCtx,
                         CONST char *progName,
                         mnLogger logger);
```

Parameters:

osdCtx	A pointer to memory reserved for Media Net resources. This memory needs to be at least SYSX_OSDPTR_SIZE bytes long and have the strictest byte alignment required on your platform (typically 4 bytes). You might generate a bus error if the bytes are not well enough aligned. This memory is not automatically allocated for you—you need to pass in the block. This memory remains allocated until you call the **mtuxTerm()** function.
progName	The name of your program (used for logging purposes).
logger	A pointer to the Media Net logger function. Use this function to log Media Net messages. (In contrast to the standard server messages logged using the **ysRecord()** function, a "Media Net message" uses the Media Net protocol.) Specify NULL to use the standard server logging functions.

Returns:

An **mtuxLayer** value that indicates either Media Net was successfully initialized or the layer at which the initialization failed.

Raises:

Nothing

See Also:

Initialize Media Net on page 1-11

Initializes Media Net and parses an **argument map** that maps command-line parameters to resource names and values. The **mtuxInit()** function associates the command-line parameters with their respective resources in the **resource database** of your program's process. This parameter-to-resource mapping is useful when using **ysr.h** functions, such as **ysResGetLast()**, **ysResGetBool()**, and so on to retrieve resource values from the resource database. See the **ysr.h** file for more information on Media Net resource management.

Note This call loads the resources from the resource file designated by your YSRESFILE environment variable. (See the *Oracle Media Net Administrator's Guide* for details.)

The **mtuxInit()** function should only be called once during the lifetime of a program. After initializing Media Net, you must call other Media Net functions, such as **ysYield()**, at least once every thirty seconds to maintain a heartbeat with **mnaddrsrv**, or Media Net will disconnect your program.

```
mtuxLayer mtuxInit(dvoid *osdCtx,
                   CONST char *progName,
                   mnLogger logger,
                   sword argCnt,
                   char **argLst,
                   CONST ysargmap *argMap);
```

Parameters:

osdCtx	A pointer to memory reserved for Media Net resources. This memory needs to be at least SYSX_OSDPTR_SIZE bytes long and have the strictest byte alignment required on your platform (typically 4 bytes). You might generate a bus error if the bytes are not well enough aligned. This memory is not automatically allocated for you—you've got to pass in the block. This memory remains allocated until you call the **mtuxTerm()** function.
progName	The name of your program (used for logging purposes).
logger	A pointer to the Media Net logger function. This is used if you wish to redirect the Media Net messages somewhere else. Specify NULL to use the standard server logging functions.
argCnt	The number of items in *argLst*. The program name is not included, so this value is typically **argc - 1**.
argLst	Command-line arguments, not including the program name, so this value is typically **argv + 1**.

argMap The map between command-line parameters and internal resource names. See the **ysr.h** file for details.

Returns:

An **mtuxLayer** value that indicates either Media Net was successfully initialized or the layer at which the initialization failed.

Raises:

Nothing

See Also:

Initialize Media Net on page 1-11

mtuxTerm

Terminates Media Net and frees associated resources. You must call this function before terminating your program.

This function should only be called if the **mtuxInit()** or **mtuxSimpleInit()** call was successful (meaning that it returned a status of **mtuxLayerSuccess**).

```
mtuxLayer mtuxTerm(dvoid *osdCtx);
```

Parameters:

osdCtx The pointer to memory reserved for Media Net resources by the **mtuxInit()** or **mtuxSimpleInit()** call.

Returns:

An **mtuxLayer** value that indicates either successful termination of Media Net or the layer at which the termination failed.

Raises:

Nothing

See Also:

Freeing Media Net Resources on page 1-13

Returns the version, release, and build information of the OVS library being used.

```
char *mtuxVersion(char *buf,
                  size_t buf_len);
```

Parameters:

buf A buffer into which the product name, version, release and build information is copied. For example:

Oracle Video Server Release 3.0.2.0.0 - Beta (built: Sep 15 1997)

buf_len The length of the specified buffer.

Returns:

The buffer (*buf*) that you passed in. This is provided as a convenience.

Raises:

Nothing

mds — Media Data Store Functions

Oracle Media Data Store (MDS) is a disk system, organized in volumes, for storing video files, audio files, and BLOB (Binary Large OBjects) files.

Applications running on client systems do not usually need to call any functions in the MDS interface; in fact, they are strongly discouraged from doing so because they can inadvertently destroy MDS contents completely. The stream service takes care of retrieving files from MDS when the client calls the **prepare()** or **prepareSequence()** method. The interface is included for developers who are writing server applications.

File access to the Media Data Store (MDS) is provided by the MDS interface described in the **mds.h** file. The functions described in this section are discussed in the following subsections:

- **Operations for Opening and Closing MDS Files**
- **MDS File Operations**
- **MDS I/O Operations**
- **MDS File Attribute Operations**

Note The MDS interface is a C interface, not an IDL interface.

MDS Data Types

MDS uses the following data types:

Data Type	Description
fileExReason	Specifies possible reasons for an exception.
mdsBlob	Defines the context for a Binary Large OBject (BLOB).
mdsBw	MDS bandwidth specification.
mdsFile	Contains information for MDS-native and host files.
mdsMch	Opaque match context for MDS name wildcard calls.

MDS also makes use of a number of flags, which are documented in *MDS flags* on page B-10.

Operations for Opening and Closing MDS Files

MDS provides the following initialization and termination functions:

Function	Description	Page
mdsInit	Initializes an MDS client for access to the MDS file system.	A-11
mdsTerm	Closes MDS server connections and frees associated resources.	A-12

Initializes an MDS client for access to the MDS file system. The **mdsInit()** function is the default initialization function that is used by most MDS clients and must be called before any other MDS functions are called.

Note You must call **mtuxInit()** to initialize Media Net before calling the **mdsInit()** function.

A client may call both **mdsInit()** and **mdsBlobInit()**, but must call **mdsInit()** before **mdsBlobInit()**.

```
void mdsInit();
```

Returns:

Nothing

Raises:

mds::io

See Also:

mdsTerm(), **mdsBlobInit()**

Using the MDS Interface on page 8-3

Closes MDS server connections and frees associated resources. Any process should call the **mdsInit()** / **mdsTerm()** pair only once.

A client may call both **mdsTerm()** and **mdsBlobTerm()**, but must call **mdsTerm()** after **mdsBlobTerm()**.

```
void mdsTerm(void);
```

Returns:

Nothing

Raises:

mds::io

See Also:

mdsInit(), **mdsBlobTerm()**

Using the MDS Interface on page 8-3

MDS File Operations

MDS provides the following file operation functions in asynchronous and synchronous modes, unless specifically noted.

Note When calling the asynchronous functions, an *uevt* value of NULL causes a semaphore to be created and returned. You must explicitly dispose of this semaphore via **ysSemDestroy()**.

Function	Description	Page
mdsCreate	Creates an MDS file.	A-14
mdsOpen	Opens an existing MDS file and obtains a file descriptor.	A-15
mdsClose	Closes an MDS file and frees associated data structures.	A-16
mdsTruncClose	Closes an MDS file, frees associated data structures, and truncates the file.	A-17
mdsLock	Marks an MDS file as read-only.	A-18
mdsUnlock	Unlocks an MDS file.	A-19
mdsRemove	Deletes an MDS file and reclaims its space immediately.	A-20
mdsUnremove	Restores an MDS file that was marked deleted if it has not yet been reclaimed.	A-21
mdsRename	Renames an MDS file.	A-22

Creates an MDS file, called *name*, with a length of *len*, and returns a file descriptor to use for all subsequent accesses to the file. Newly created files are always opened in read-write mode.

When you call **mdsCreate()**, create a file of length *X*, where *X* is the largest size the file could be. When the file is written out in *Y* bytes, call **mdsTruncClose()** to truncate the file to *Y* bytes in length.

```
mdsFile *mdsCreate(CONST char *name,
                   CONST sysb8 *len,
                   mdsBw *bwtkn,
                   ub4 flags);

ysevt *mdsCreate_nw(CONST char *name,
                    CONST sysb8 *len,
                    mdsBw *bwtkn,
                    ub4 flags,
                    ysevt *uevt);
```

Parameters:

name	The name of the file to create.
len	The maximum number of bytes to allocate for the file. Files cannot grow beyond this size. Files may be truncated with **mdsTruncClose()**.
bwtkn	Specify *MdsDfltBitRate* for volume default (see **mdsBw**).
flags	Zero or more MDS flags. The *MdsFlgWrt* flag is set automatically (see **MDS flags**).
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsCreate() returns the **mdsFile** file descriptor for the specified file.

mdsCreate_nw() returns the event that is triggered upon completion. The file descriptor is returned when the event is triggered.

Raises:

mds::fileEx

See Also:

mdsRemove(), **mdsOpen()**

Opens an existing MDS file and obtains a file descriptor to use for all subsequent accesses to the file. Files opened with **mdsOpen()** can be closed with **mdsClose()**. If the file has been opened in read / write mode, it is possible to truncate it.

```
mdsFile *mdsOpen(CONST char *name,
                 mdsBw *bwktn,
                 ub4 flags);

ysevt *mdsOpen_nw(CONST char *name,
                  mdsBw *bwtkn,
                  ub4 flags,
                  ysevt *uevt);
```

Parameters:

name	The name of the file to open.
bwtkn	Specify *MdsDfltBitRate* for volume default (see **mdsBw**).
flags	Zero or more MDS flags (see **MDS flags**).
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsOpen() returns the **mdsFile** file descriptor for the specified file.

mdsOpen_nw() returns the event that is triggered upon completion. The file descriptor is returned when the event is triggered.

Raises:

mds::fileEx

See Also:

mdsClose()

Using the MDS Interface on page 8-3

Closes an MDS file and frees associated data structures.

```
void mdsClose(mdsFile *fp);

ysevt *mdsClose_nw(mdsFile *fp,
                   ysevt *uevt);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate()**.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsClose() returns nothing.

mdsClose_nw() returns the event that is triggered upon completion.

Raises:

mds::io

See Also:

mdsOpen(), **mdsTruncClose()**

Using the MDS Interface on page 8-3

Closes an MDS file, frees associated data structures, and truncates the file to *newLen* bytes. If *newLen* is set to *MdsEof*, the file is truncated to the "high water mark," which is the greatest offset ever written to. This is the only function for changing the size of an MDS file. Truncation applies only to MDS files.

Most clients that create files intentionally overestimate their size at creation time, write to the file for some time, and then use **mdsTruncClose()** to truncate the file at the last written position.

```
void mdsTruncClose(mdsFile *fp,
                   CONST sysb8 *newLen);

ysevt *mdsTruncClose_nw(mdsFile *fp,
                        CONST sysb8 *newLen,
                        ysevt *uevt);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate()**.
newLen	The new length for the file.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsTruncClose() returns nothing.

mdsTruncClose_nw() returns the event that is triggered upon completion.

Raises:

mds::io

See Also:

mdsOpen(), **mdsTruncClose()**, **mdsHighWaterMark()**

Marks an MDS file as read-only. Subsequent attempts to open the file for writing result in an MDS exception.

The **mdsLock()** operation is **idempotent**, which means multiple successful invocations are equivalent to a single successful invocation.

Use locking to prevent inadvertent modification of read-only data.

Locking is provided as a convenience to the developer. There is no ownership of files, and no permissions are associated with a lock. The operation is not available on host files.

```
void mdsLock(CONST char *name);

ysevt *mdsLock_nw(CONST char *name,
                  ysevt *uevt);
```

Parameters:

name	The file name.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsLock() returns nothing.

mdsLock_nw() returns the event that is triggered upon completion.

Raises:

mds::fileEx

See Also:

mdsUnlock()

Synchronous and Asynchronous MDS Functions on page 8-4

Unlocks an MDS file. This operation is **idempotent** (see **mdsLock()** for the definition).

```
void mdsUnlock(CONST char *name);

ysevt *mdsUnlock_nw(CONST char *name,
                    ysevt *uevt);
```

Parameters:

name	The file name.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsUnlock() returns nothing.

mdsUnlock_nw() returns the event that is triggered upon completion.

Raises:

mds::fileEx

See Also:

mdsLock()

Deletes an MDS file and reclaims its space immediately if the *forever* parameter is set to true. If *forever* is set to false, **mdsRemove()** marks the file deleted. Subsequent attempts to open the file for reading fail unless the **MdsFlgDel** flag is specified.

Files deleted with *forever* set to false may be restored using **mdsUnremove()** if they have not yet been reclaimed by the system.

The system periodically reclaims the space associated with deleted files. When this happens, a deleted file is removed from the system entirely and can no longer be reclaimed with **mdsUnremove()**. The system is free to remove all traces of a deleted file at its discretion.

Deleting a file with *forever* set to false is the normal behavior. Typically the file is merely marked as deleted and only overwritten when the system needs the space to store new files. This gives the administrator the option of using the **mdsUnremove()** function to restore the file. Permanent removal makes sense for temporary files. It also makes it possible to remove a secure file and all records of it from the system.

```
*void mdsRemove(CONST char *name,
                boolean forever);

ysevt *mdsRemove_nw(CONST char *name,
                    boolean forever,
                    ysevt *uevt);
```

Parameters:

name The file name.

forever True if the file is to be completely deleted and its space reclaimed. False if the file may later be restored by **mdsUnremove()**.

uevt An event to trigger upon completion of the function. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsRemove() returns nothing.

mdsRemove_nw() returns the event that is triggered upon completion.

Raises:

mds::fileEx

mdsUnremove

Restores an MDS file that has been removed by the **mdsRemove()** function with the *forever* flag set to false.

```
void mdsUnremove(CONST char *name);

ysevt *mdsUnremove_nw(CONST char *name,
                      ysevt *uevt);
```

Parameters:

name	The name of the file to restore.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsUnremove() returns nothing.

mdsUnremove_nw() returns the event that is triggered upon completion.

Raises:

mds::fileEx

Renames an MDS file. The specified file must be read-write, and not open for writing by any other client.

```
void mdsRename(CONST char *oldNm,
               CONST char *newNm);

ysevt *mdsRename_nw(CONST char *oldNm,
                    CONST char *newNm,
                    ysevt *uevt);
```

Parameters:

oldNm	The name of the file to be renamed.
newNm	The new name for the file.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsRename() returns nothing.

mdsRename_nw() returns the event that is triggered upon completion.

Raises:

mds::fileEx

See Also:

Synchronous and Asynchronous MDS Functions on page 8-4

MDS I/O Operations

MDS provides the following I/O operation functions in asynchronous and synchronous modes, unless specifically noted:

Function	Description	Page
mdsRead	Reads from the current file to the specified buffer.	A-24
mdsWrite	Writes from the specified buffer to the current file offset.	A-25
mdsFlush	Flushes all dirty buffers corresponding to the specified file pointer.	A-26
mdsSeek	Moves the current file offset pointer to a new offset.	A-27
mdsCopySeg	Efficiently copies a segment of a file.	A-28

Reads *len* bytes of data from the current file offset, *fp*, and copies it into *buf*.

```
size_t mdsRead(mdsFile *fp,
               dvoid *buf,
               size_t len);

ysevt *mdsRead_nw(mdsFile *fp,
                  dvoid *buf,
                  CONST sysb8 *off,
                  size_t len,
                  ysevt *uevt);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate()**.
buf	A buffer into which the data is copied.
len	The number of bytes of data to read.
off	Starting file offset of data to read.
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsRead() returns the number of bytes of data successfully read. The return value is less than *len* when reading past EOF.

mdsRead_nw() returns the event that is triggered upon completion. The number of bytes of data successfully read is returned when the event is triggered.

Raises:

mds::io

See Also:

mdsWrite(), **mdsSeek()**

Using the MDS Interface on page 8-3

Writes *len* bytes from *buf* to the current file offset, *fp*. **mdsWrite_nw()** is the asynchronous version of **mdsWrite()**, which requires that *buf* remain valid and unaltered until the operation completes and that a file offset, *off*, is specified.

```
size_t mdsWrite(mdsFile *fp,
                dvoid *buf,
                size_t len);

ysevt *mdsWrite_nw(mdsFile *fp,
                   dvoid *buf,
                   CONST sysb8 *off,
                   size_t len,
                   ysevt *uevt);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**
buf	The buffer from which to load data.
len	The number of bytes of data to write.
off	Starting offset of data to write. (Asynchronous only.)
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsWrite() returns the number of bytes of data successfully written. The return value is less than *len* when writing past EOF.

mdsWrite_nw() returns the event that is triggered upon completion. The number of bytes of data successfully written is returned when the event is triggered.

Raises:

mds::fileEx, **mds::io**

See Also:

mdsRead(), **mdsSeek()**

Flushes all dirty buffers corresponding to the specified file pointer, *fp*. If *fp* is NULL, then **mdsFlush()** flushes buffers corresponding to all open files.

```
void mdsFlush(mdsFile *fp);

ysevt *mdsFlush_nw(mdsFile *fp,
                   ysevt *uevt);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**
uevt	An event to trigger upon completion. If NULL, this function creates and returns a semaphore. (Asynchronous only.)

Returns:

mdsFlush() returns nothing.

mdsFlush_nw() returns the event that is triggered upon completion.

Raises:

mds::io

See Also:

mdsRead(), **mdsWrite()**

Moves the current file offset pointer to a new offset. If the new offset is greater than the size of the file, **mdsSeek()** sets the file pointer to the end of the file.

```
boolean mdsSeek(mdsFile *fp,
                CONST sysb8 *off,
                sysb8 *newOff);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**
off	Offset relative to the beginning of the file to which the current file offset pointer is set.
newOff	The new file offset. Normally, this is equal to *off*. However, if *off* is greater than the size of the file, it sets it to end of file (EOF). If *newOff* is NULL, it is not set.

Returns:

True if the new offset is equivalent to the offset requested, false otherwise.

See Also:

mdsOpen(), **mdsCreate()**

Using the MDS Interface on page 8-3

mdsCopySeg

Efficiently copies a segment of file *srcFp* to file *dstFp*.

```
void mdsCopySeg(mdsFile *srcFp,
                mdsFile *dstFp,
                CONST sysb8 *srcOff,
                CONST sysb8 *srcLim,
                CONST sysb8 *dstOff);
```

Parameters:

srcFp	A pointer to the **mdsFile** to copy from.
dstFp	A pointer to the **mdsFile** to copy to. Must be opened with **mdsCreate()** or **mdsOpen()** with **MdsFlgWrt** in the *flags* parameter.
srcOff	Offset relative to the start of the *srcFp* file at which to start copying.
srcLim	Offset relative to the start of the *srcFp* file at which to stop copying.
dstOff	Offset relative to the start of the *dstFp* file at which to start copying.

Returns:

Nothing

Raises:

mds::io

See Also:

mdsOpen(), **mdsCreate()**

MDS File Attribute Operations

MDS provides the following file attribute operation functions:

Function	Description	Page
mdsFileTypeMds	Checks whether a file pointer corresponds to an MDS file.	A-29
mdsName	Returns the full file name.	A-30
mdsLen	Returns the length of the specified file.	A-30
mdsCreateLen	Returns the number of bytes allocated on the disk for the specified file.	A-31
mdsCreateTime	Returns the creation time for the file, in seconds.	A-31
mdsCreateWall	Returns the creation time for the file, in calendar units.	A-32
mdsHighWaterMark	Returns the high-water mark.	A-32
mdsPos	Returns the current file offset.	A-33
mdsEof	Checks whether the file offset is at EOF.	A-33

mdsFileTypeMds

Checks whether a file pointer corresponds to an MDS file.

```
boolean mdsFileTypeMds(mdsFile *fp);
```

Parameters:

fp A pointer to the **mdsFile** to check.

Returns:

True if *fp* corresponds to an MDS file.
False if *fp* corresponds to a host file.

mdsName

Returns the full file name. This function is valid on both host and MDS files.

Note The name of the file (*fp*) is cached when the file is opened. If you invoke the **mdsRename()** function to rename the file while it is opened, **mdsName()** returns the *original* name.

```
char *mdsName(mdsFile *fp);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

Returns:

Full file name corresponding to the specified file pointer.

mdsLen

Returns the length of the specified file. This function is valid on both host and MDS files.

```
void mdsLen(mdsFile *fp,
            sysb8 *len);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**
len	Length of the file.

Returns:

Nothing. (Stores the length in *len*.)

mdsCreateLen

Returns the number of bytes allocated on the disk for the specified file.

```
void   mdsCreateLen(mdsFile *fp,
                    sysb8 *len);
```

Parameters:

fp A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

len The number of bytes allocated for the file.

Returns:

Nothing. (Stores the length in *len*.)

mdsCreateTime

Returns the creation time for the file, in seconds. This function is valid only on MDS files.

The returned value is a measurement of the time elapsed since the Unix epoch (Jan 1, 1970). See **ystm.h** for more information on time and how the creation time is specified.

```
sb4 mdsCreateTime(mdsFile *fp);
```

Parameters:

fp A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

Returns:

File creation time, in seconds. (Number of seconds since Jan 1, 1970.)

mdsCreateWall

Returns the creation time for the file, in calendar units. This function is valid only on MDS files.

```
mkd_localWall *mdsCreateWall(mkd_localWall *crtTm,
                             mdsFile *fp );
```

Parameters:

crtTm	An **mkd::localWall** value that specifies time in "real world" fashion.
fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

Returns:

An **mkd::localWall** value that specifies the file creation time, in calendar units.

mdsHighWaterMark

Returns the "high-water mark" (the greatest offset written to) of the file.

```
void mdsHighWaterMark(mdsFile *fp,
                      sysb8 *off);
```

Parameters:

fp	A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**
off	High water mark offset.

Returns:

Nothing. (Stores the current high water mark in *off*.)

mdsPos

Returns the current file offset.

```
void mdsPos(mdsFile *fp,
            sysb8 *pos);
```

Parameters:

fp A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

pos Stores returned file offset.

Returns:

Nothing

See Also:

mdsEof()

mdsEof

Checks whether the file offset is at end of file (EOF).

```
boolean mdsEof(mdsFile *fp);
```

Parameters:

fp A pointer to an **mdsFile** returned by **mdsOpen()** or **mdsCreate().**

Returns:

True if the file offset is at EOF, false otherwise.

See Also:

mdsPos()

mdsBlob — MDS BLOB Functions

OVS clients can retrieve unstructured and untyped data, or BLOBs (Binary Large OBjects), from MDS using the MDS BLOB interface. For an overview of this service, see *Using the MDS BLOB Interface* on page 8-7.

Functions

The MDS BLOB interface provides the following functions:

Function	Description	Page
mdsBlobInit	Initializes the client for future BLOB interface calls.	A-35
mdsBlobTerm	Cleans up any outstanding states created by prior calls to the BLOB interface and terminates the BLOB interface.	A-35
mdsAllocFunc	Prototype for a user-defined callback function; used by **mdsBlobPrepare()** and **mdsBlobPrepareSeg()**.	A-36
mdsBlobPrepare	Prepares for the retrieval of MDS content.	A-36
mdsBlobPrepareSeg	Identical to **mdsBlobPrepare()** but enables the client to specify the starting offset and size within the BLOB to transfer.	A-38
mdsBlobTransfer	Initiates transfer of the content prepared by previous calls to **mdsBlobPrepare()** and / or **mdsBlobPrepareSeg()**.	A-39

mdsBlobInit

Initializes the client for future BLOB interface calls. The **mdsBlobInit()** function should only be called once. A client must call **mdsBlobInit()** before it can use the MDS BLOB interface. This function blocks until completed.

A client may call both **mdsInit()** and **mdsBlobInit()**, but must call **mdsInit()** before **mdsBlobInit()**.

```
void mdsBlobInit(void);
```

Returns:

Nothing

Raises:

MDS_EX_NOTINIT Initialization failed.

See Also:

mdsBlobTerm(), **mdsInit()**

Streaming a BLOB to a Client on page 8-8

mdsBlobTerm

Cleans up any outstanding states created by prior calls to the BLOB interface and terminates the BLOB interface. This function should only be called once.

A client may call both **mdsTerm()** and **mdsBlobTerm()**, but must call **mdsTerm()** after **mdsBlobTerm()**.

```
void mdsBlobTerm(void);
```

See Also:

mdsBlobInit(), **mdsTerm()**

Streaming a BLOB to a Client on page 8-8

mdsAllocFunc

Prototype for a user-defined callback function to allocate memory for BLOBs.

```
typedef ub1 *(*mdsAllocFunc)(dvoid *usrp,
                             size_t segSz,
                             ub4 off,
                             ub4 total,
                             size_t *actSz);
```

Parameters:

usrp	User pointer that was passed into **mdsBlobPrepare()** or **mdsBlobPrepareSeg()**.
segSz	The number of bytes of memory to allocate.
off	Offset within the requested BLOB or BLOB segment.
total	The total number of bytes in the BLOB or BLOB segment.
actSz	The number of bytes of memory allocated for BLOB offset, *off*.

Returns:

A pointer to allocated memory. Returns NULL if no more memory is available.

mdsBlobPrepare

Prepares for the retrieval of one BLOB from MDS. The BLOB is described by the **mkd::assetCookie**, which is a tag that is completely opaque to the client and is resolved by the server. The actual data transfer is initiated by a call to **mdsBlobTransfer()** using the context returned by this call.

The **mdsBlobTransfer()** function uses the specified allocation function to allocate space for the content as it arrives. The input event is triggered with a return code indicating the transfer status when the transfer is complete. The client can cancel the transfer at any time by destroying the event.

Note that **mdsBlobPrepare()** only initializes state on the client side and that calling it does not result in any communication with the server.

```
mdsBlob *mdsBlobPrepare(mkd_assetCookie asset,
                        CORBA_Object auth,
                        ub4 flgs,
                        mdsAllocFunc allocf,
                        dvoid *usrp,
                        ysevt *evt);
```

Parameters:

asset	The **mkd::assetCookie** used to identify the content. This value must remain valid until the transfer is complete.
auth	An object reference to the authentication interface (if any) for the asset.
flgs	Flags. No flags are currently supported.
allocf	The **mdsAllocFunc()** callback routine to be invoked when the caller needs to allocate memory for the content.
usrp	User pointer that is passed to the allocation function.
evt	An event to be triggered when transfer is complete.

Returns:

An **mdsBlob** structure containing the context for the BLOB transfer. The **mdsBlob** structure should be passed to **mdsBlobTransfer()** to initiate the transfer. This context is automatically destroyed by **mdsBlobTransfer()**, or when the event is destroyed.

Raises:

If the BLOB transfer fails, the event is triggered with **mds::io** or **mds::fileEx**.

The event may also be triggered with YS_EX_TIMEOUT if a timeout occurs during BLOB transfer, indicating network transmission problems or that the user should yield to Media Net more frequently by adding **ysYield()** calls (see *Using the MDS BLOB Interface* on page 8-7).

See Also:

Streaming a BLOB to a Client on page 8-8

Identical to **mdsBlobPrepare()** but enables the client to specify the starting offset and size within the BLOB to transfer.

```
mdsBlob *mdsBlobPrepareSeg(mkd_assetCookie asset,
                           CORBA_Object auth,
                           ub4 off,
                           ub4 len,
                           ub4 flgs,
                           mdsAllocFunc allocf,
                           dvoid *usrp,
                           ysevt *evt);
```

Parameters:

asset	The **mkd::assetCookie** used to identify the content. This value must remain valid until the transfer is complete.
auth	An object reference to the authentication interface (if any) for the asset.
off	Starting offset of BLOB to transfer.
len	The number of bytes of BLOB to transfer.
flgs	Flags. No flags are currently supported.
allocf	The **mdsAllocFunc()** callback routine to be invoked when the caller needs to allocate memory for the content.
userp	User pointer that is passed to the allocation function.
evt	An event to be triggered when transfer is complete.

Returns:

An **mdsBlob** structure containing the context for the BLOB transfer. The **mdsBlob** structure should be passed to **mdsBlobTransfer()** to initiate the transfer. This context is automatically destroyed by **mdsBlobTransfer()**, or when the event is destroyed.

Raises:

If the BLOB transfer fails, the event is triggered with **mds::io** or **mds::fileEx**.

The event may also be triggered with YS_EX_TIMEOUT if a timeout occurs during BLOB transfer, indicating network transmission problems or that the user should yield to Media Net more frequently by adding **ysYield()** calls (see *Using the MDS BLOB Interface* on page 8-7).

mdsBlobTransfer

Initiates transfer of the **mdsBlob** content prepared by previous calls to **mdsBlobPrepare()** and / or **mdsBlobPrepareSeg()**. When this function has completed the transfer of a particular BLOB or BLOB segment, the events set by the **prepare** functions are triggered.

```
void mdsBlobTransfer(mdsBlob **blobs,
                     sword numBlobs,
                     ub4 flgs);
```

Parameters:

blobs	An array of **mdsBlob** contexts to be transferred. These BLOBs are returned by previous calls to **mdsBlobPrepare()** or **mdsBlobPrepareSeg()**.
numBlobs	The number of BLOBs to transfer.
flgs	Flags. No flags are currently supported.

Returns:

Nothing

See Also:

Streaming a BLOB to a Client on page 8-8

mdsnm — MDS Name Functions

Provides an interface for file name expansion and manipulation.

Note In the **mdsnm** interfaces, the term “native” refers to MDS files, rather than the files located on your “host” file system.

mdsnm Data Types

The MDS Name interface uses the following data types:

Data Type	Description
MdsnmMaxLen	Maximum length of an MDS or host file name.
MdsnmNtvMaxLen	Maximum length of a native MDS file name.

Functions

The MDS Name interface provides the following functions:

Function	Description	Page
mdsnmTypeNtv	Checks whether the specified file corresponds to an MDS file.	A-41
mdsnmIsLegal	Checks whether the specified file name is a legal MDS file name.	A-42
mdsnmSplit	Splits the specified file name into its component parts.	A-43
mdsnmNtvSplit	Identical to **mdsnmSplit()** but assumes a native name.	A-44
mdsnmJoin	Joins a directory path and file name into a complete path name.	A-45
mdsnmNtvJoin	Identical to **mdsnmJoin()** but assumes a native name**.**	A-46
mdsnmVolCmp **mdsnmFileCmp**	Compares MDS volumes (files) in the same manner as **strcmp()** (no wildcard expansion).	A-47

mdsnmFormat	Enables code, dependent on the operating system, a first pass at interpreting a file name.	A-48
mdsnmMatchCreate	Begins a wildcard search according to the specified expression and returns a match context.	A-49
mdsnmMatchNext	Returns the next matching file, or false if there are no more matching files.	A-50
mdsnmMatchDestroy	Frees the match context.	A-50
mdsnmExpand **mdsnmExpandOne**	Expands a list of user-specified file names (or a single user-specified file name) and returns an ordered list of matches.	A-51

mdsnmTypeNtv

Checks whether the specified file corresponds to an MDS file. The specified file must be a formatted file name. Fully formatted file names can be constructed with **mdsnmFormat()**, the wildcard expansion functions, or from an open file pointer.

```
boolean mdsnmTypeNtv(CONST char *name);
```

Parameters:

name The name of the file to check.

Returns:

True if *name* corresponds to an MDS file, false otherwise.

See Also:

mdsnmIsLegal()

Checks whether the specified file name corresponds to a legal MDS file name. A simple name, such as **myfile**, or a full path name, such as **/mds/vol1/myfile**, may be specified. Host file name checking is not currently supported.

Legal file names must begin with an alphanumeric character, followed by any sequence of alphanumeric, "*" and "_" characters. The maximum length is defined by **MdsnmNtvMaxLen()**. The **mdsnmIsLegal()** function enables clients to test if a file name from a host file system can be used within an MDS file system. This is necessary because the host system may have different file naming conventions than the MDS file system.

```
boolean mdsnmIsLegal(CONST char *name);
```

Parameters:

name The name of the file to check.

Returns:

True if *name* is a legal **mdsFile** name, false otherwise.

See Also:

mdsnmTypeNtv()

Splits the specified file name into its component parts.

Both the host file system and MDS guarantee only that a full file name (one with path or volume information) is unique. Most host systems support nested containers (directories) for file names; MDS supports only a single container (the volume). This function operates on both host and MDS names.

If a component part is not available, then an empty string is returned. Passing NULL for either *dirNm* or *fileNm* indicates that the component is to be ignored. The **mdsnmSplit()** function must be able to determine if the file name is a host name or an MDS name by examining the file prefix.

```
void mdsnmSplit(CONST char *name,
                char *dirNm,
                char *fileNm);
```

Parameters:

name	The full name of the file.
dirNm	The space to hold either the returned host directory or an MDS container of the form /mds/*volume* (at least **MdsnmMaxLen** +1 bytes).
fileNm	The space to hold the returned file name (at least **MdsnmMaxLen** +1 bytes).

Returns:

Nothing

Raises:

mds::fileEx

See Also:

mdsnmJoin(), **mdsnmNtvSplit()**

Identical to **mdsnmSplit()** but assumes a native name. As a result, the container is returned as *volNm* as opposed to /mds/*volNm*.

```
void mdsnmNtvSplit(CONST char *name,
                   char *volNm,
                   char *fileNm);
```

Parameters:

name	The full name of the file, in the form /mds/*volume*/*file*.
volNm	The space to hold the returned volume name (at least **MdsnmMaxLen**+1 bytes, or NULL).
fileNm	The space to hold the returned file name (at least **MdsnmMaxLen**+1 bytes, or NULL).

Returns:

Nothing

Raises:

mds::fileEx

Example

```
mdsnmNtvSplit("/mds/vol1/file1", vnm, (char *)0);
```

Copies the string “vol1” into *vnm*.

See Also:

mdsnmSplit(), **mdsnmNtvJoin()**

Joins a directory path and file name into a complete path name. Can be used on either host or MDS files. The **mdsnmJoin()** function determines if the resulting path name is to be an MDS path name or a host path name by examining the *dirNm* prefix. (The process of forming a host file name is platform specific.)

```
void mdsnmJoin(char *name,
               CONST char *dirNm,
               CONST char *fileNm);
```

Parameters:

name	The space to hold the returned path name (at least **MdsnmMaxLen**+1 bytes).
dirNm	The host directory or MDS container of the form /mds/*volume*.
fileNm	The name of the file.

Returns:

Nothing

See Also:

mdsnmSplit(), **mdsnmNtvJoin()**

Identical to **mdsnmJoin()** but assumes a native name. The caller can specify a volume name as /mds/*volume* or as *volume*. Additionally, if *fileNm* is NULL, then the resulting name is of the form /mds/*volume* (without a trailing slash).

```
void mdsnmNtvJoin(char *name,
                  CONST char *volNm,
                  CONST char *fileNm);
```

Parameters:

name	The space to hold the returned path name of the form /mds/*vol1*/*file1* (at least **MdsnmMaxLen**+1 bytes).
volNm	The volume name of the form /mds/*volume* or *volume*.
fileNm	The name of the file, or NULL.

Returns:

Nothing

See Also:

mdsnmSplit(), **mdsnmNtvSplit()**

mdsnmVolCmp
mdsnmFileCmp

Compares complete MDS names in the same fashion as **strcmp()**. This comparison does *not* perform wildcard expansion.

You can specify MDS volume names with strings of the form /mds/*volume*, or simply *volume*.

Use **mdsnmFileCmp()** to compare file names of the form /mds/*volume/file*.

```
word mdsnmVolCmp(CONST char *volNm1,
                 CONST char *volNm2);

sword mdsnmFileCmp(CONST char *fileNm1,
                   CONST char *fileNm2);
```

Parameters:

Nm1, *Nm2* The files (or volumes) to compare.

Returns:

An integer greater than, equal to, or less than 0. If the string pointed to by *nm1* is greater than, equal to, or less than, the string pointed to by *nm2* respectively, that is:

< 0 if *nm1* < *nm2*,
0 if equal,
> 0 if *nm1* > *nm2*.

Enables code, dependent on the operating system, a first pass at interpreting a file name. Usually, the operating system dependent code resolves relative file names and performs any other necessary formatting. The resulting name must be identifiable as a native MDS or host file name with **mdsnmTypeNtv()**.

The formatted name, *inName*, is copied into *outName*, which must be at least **MdsnmMaxLen** +1 bytes long.

```
void mdsnmFormat(CONST char *inName,
                 char *outName);
```

Parameters:

inName	The file name to be formatted.
outName	The space to hold the *inName* file name after formatting (at least **MdsnmMaxLen** +1 bytes).

Returns:

Nothing

mdsnmMatchCreate

Begins a wildcard search according to the specified expression and returns a match context to be used in subsequent **mdsnmMatchNext()** calls.

The caller can pass in flags (of type **MdsFlg**) to further constrain matching files. Typically the caller passes in the same flags that are used in subsequent **mdsOpen()** calls to restrict the set of matching files to those of interest. The **MdsFlgWrt** and **MdsFlgDel** flags are supported (see **MDS flags**).

Wildcard characters can only be used for native MDS files but can be used to return both volume names and file names. To retrieve an MDS volume name, use a wildcard character for the volume name, where *expr* is of the form /mds/*volume*. Matching volume names are returned.

This call may block the calling thread or process.

```
mdsMch *mdsnmMatchCreate(CONST char *expr,
                         ub4 flags);
```

Parameters:

expr	Wildcard expression (example: /mds/v*1/?aa?)
flags	Match deleted, invalid files, and so on (see **MDS flags**).

Returns:

An **mdsMch** structure representing a match context. Use the **mdsnmMatchDestroy()** function to delete the **mdsMch** structure when you are finished.

mdsnmMatchNext

Returns the next matching file, or false if there are no more matching files. The caller is expected to allocate sufficient space for a matching file name, at least **MdsnmMaxLen** +1 bytes.

Wildcard characters can only be used for native MDS files. Host file names are copied into *fileNm* by **mdsnmMatchNext()**, but otherwise ignored.

This call may block the calling thread or process.

```
boolean mdsnmMatchNext(mdsMch *mcx,
                       char *fileNm);
```

Parameters:

mcx The **mdsMch** structure representing a match context.

fileNm The file name that matches.

Returns:

True if returning the next matching file.
False if there are no more matches.

mdsnmMatchDestroy

Frees the match context.

```
void mdsnmMatchDestroy(mdsMch *mcx);
```

Parameters:

mcx The **mdsMch** structure representing the match context to free.

Returns:

Nothing

The **mdsnmExpand()** function expands a list of user-specified file names and returns an ordered list of matches. The **mdsnmExpandOne()** function is a convenience function that can be used if only a single file name is to be expanded. Both functions format a user-specified file name according to **mdsnmFormat()**, expand any wildcard characters for MDS file names, and return a list of matches. Host file names are simply copied to the output list.

These functions support **mdsnmMatchCreate()** and related functions. They provide UNIX-style shell behavior by expanding all wildcard characters up front, copying over elements without wildcard characters, and raising MDS_EX_NMRESOLVE with reason *mds_nomatch* for false matches.

It is the responsibility of the caller to free the output list and list elements at a later time. This can be accomplished by calling **ysLstDestroy(lst, ysmFGlbFree)**.

An empty list is returned if no matches are found. If *maxMatches* is zero, then the list is unbounded; otherwise it will contain at most *maxMatches* elements.

These calls can block the calling thread or process.

```
yslst *mdsnmExpand(yslst *inNms,
                   ub4 flags,
                   sb4 maxMatches);

yslst *mdsnmExpandOne(CONST char *inNm,
                      ub4 flags,
                      sb4 maxMatches);
```

Parameters:

inNm(s)	A user-specified file name (or list of user-specified file names). See **yslst.h**.
flags	Match deleted files (see **MDS flags**).
maxMatches	The maximum number of files to match (pass zero for unbounded).

Returns:

A **yslst** containing a list of expanded, formatted MDS file names.

See Also:

Expanding Wildcards on page 8-6

mtcr — Content Resolver Interface

The content resolver is used to resolve logical content into physical content segments. This service is typically used by the stream service or by the MDS BLOB library, and not directly by client applications.

mtcr::resolve

The **resolve** interface is described in the **mtcr.idl** file. The resolver is implemented by a separate server to resolve logical **mkd::assetCookies** to physical content segments. When a client requests the stream service to play a piece of content represented by an **mkd::assetCookie**, the stream service processes the cookie to determine which resolution implementation is required to properly translate the content name. The stream service then calls a content resolver of the appropriate type, which translates the name and returns a list of segments that describe the physical instantiation of the name.

Methods

The **resolve** interface provides a single method, **name()**.

Translates a name into physical segments. The format of the name depends on the implementation of the resolver.

The *destination* is the network circuit authorized to deliver the content. The content resolver can use this for further authorization. It is possible that a client may ask for content translation without providing this circuit (it may not be available to all content distributors). In this case, it is up to the discretion of the content manager whether to actually do the resolution or to return an **mtcr::authFailed** exception.

```
mkd::segmentList name(in string resName,
                      in Object authRef,
                      in any destination);
```

Parameters:

resName	A string whose format is determined by the implementation. This is what is translated.
authRef	An object reference to an object used to authenticate the client. The source of the **mkd::assetCookie** dictates what object reference, if any, to pass here. For example, when the **mkd::assetCookie** is passed from **vscontsrv**, no authorization object is required.
destination	The network circuit that is used to deliver the content if it is authorized. Note that this circuit is defined as an **any** rather than an **mzc::circuit** to prevent dependencies for clients that need to resolve content but will not be passing a circuit. The object passed should always be either an **mzc::circuit** or a NULL object.

Returns:

An **mkd::segmentList** sequence of **mkd::segment** structures that describe the physical instantiation of a piece of logical content.

Raises:

mtcr::authFailed, **mtcr::badName**

See Also:

Implement the Content Resolver Interface on page 5-9

mza — Logical Content Interfaces

The logical content service enables you to manipulate the metadata for the physical content to build abstractions of the content in the form of **clips** and **logical content**, as described in *Logical Content Model* on page 3-2.

The logical content service, **vscontsrv**, consists of the **mtcr::resolve** interface described on page A-52 and the following interfaces:

- **mza::LgCtnt**, **mza::LgCtntFac**, **mza::LgCtntMgmt**
- **mza::Ctnt**, **mza::CtntFac**, **mza::CtntMgmt**
- **mza::CtntPvdr**, **mza::CtntPvdrFac**, **mza::CtntPvdrMgmt**
- **mza::Clip**, **mza::ClipFac**, **mza::ClipMgmt**
- **mza::BlobMgmt**

mza::LgCtnt

The **LgCtnt** interface represents a unit of logical content, and is the interface most clients will use to find available OVS content. In the simplest terms, logical content consists of zero or more clips, each describing a portion of a content file in MDS.

LgCtnt Data Types

The **LgCtnt** interface uses the following data types:

Data Type	Description
mza::ClipAtr	Attribute information of an **mza::Clip** object.
mza::ClipAtrLst	Sequence of **mza::ClipAtr** structures.
mza::LgCtntAtr	Attribute information of an **mza::LgCtnt** object.
mza::LgCtntAtrLst	Sequence of **mza::LgCtntAtr** structures.
mza::CtntAtr	Attribute information of an **mza::Ctnt** object.
mza::CtntAtrLst	Sequence of **mza::CtntAtr** structures.
mza::Itr	Structure for iterating through a list.

Attributes

The logical content service contains attributes that are stored as persistent data in a database. These attributes are written to the database using **set** methods and read using **get** methods. If the attribute is read-only, there is no **set** method. The *msecs* and *numClips* attributes are read-only because these are calculated depending on the number and type of clips.

Attribute	Description
name	An unique name identifying the logical content.
desc	An optional description about the logical content.
msecs	The total run time of the entire logical content, in milliseconds. This is calculated based on the clips contained in the logical content.
numClips	The number of clips assigned to the logical content.

Methods

The **LgCtnt** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **LgCtnt** object.	A-56
destroy	Destroys the **LgCtnt** object.	A-56
getAtrClipByPos	Returns the attributes of the **mza::Clip** object located at the specified position.	A-57
lstAtrClips	Returns the attributes of all **mza::Clip** objects in the **LgCtnt** object.	A-57
addClip	Adds an **mza::Clip** object to the **LgCtnt** object.	A-58
addClipByPos	Inserts an **mza::Clip** object at the specified position in the **LgCtnt** object.	A-58
delClip	Removes the specified **mza::Clip** object from the **LgCtnt** object.	A-59

Method	Description	Page
delClipByPos	Removes an **mza::Clip** object from the specified position in the **LgCtnt** object.	A-60

getAtr

Returns the attributes of the **LgCtnt** object.

```
void getAtr(in boolean longFmt,
            out LgCtntAtr lcAtr);
```

Parameters:

longFmt If true, clip and content information is added to *lcAtr*.

lcAtr The **mza::LgCtntAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::BadPosition, **mza::BadProhib**, **mza::PersistenceError**

destroy

Destroys the **LgCtnt** object. This method first removes all of the **mza::Clip** objects in the **LgCtnt** object and then destroys the **LgCtnt** object.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

See Also:

Destroy Logical Content on page 3-22

getAtrClipByPos

Returns the attributes of the **mza::Clip** object located at the specified position in the **LgCtnt** object.

```
void getAtrClipByPos(in long position,
                     out ClipAtr clipAtr);
```

Parameters:

position Absolute position of the **mza::Clip** object within the **LgCtnt** object. The first clip in a **LgCtnt** is located at *position 1*.

clipAtr The **mza::ClipAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::BadPosition, **mza::PersistenceError**

lstAtrClips

Returns the attributes of all **mza::Clip** objects in the **LgCtnt** object.

```
ClipAtrLst lstAtrClips(inout Itr iterator);
```

Parameters:

iterator On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::ClipAtrLst** sequence of **mza::ClipAtr** structures.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::PersistenceError**

addClip

Adds an **mza::Clip** object to the **LgCtnt** object. The new clip is added *after* any existing clips in the **LgCtnt** object.

```
long addClip(in Clip clipOR);
```

Parameters:

clipOR An object reference of the **mza::Clip** object to add.

Returns:

The position of the newly added **mza::Clip** object within the **LgCtnt** object.

Raises:

mza::PersistenceError

See Also:

Add Clips to Logical Content on page 3-21

addClipByPos

Inserts an **mza::Clip** object at the specified position in the **LgCtnt** object. The remaining clips are reordered.

```
void addClipByPos(in Clip clipOR,
                  in long position);
```

Parameters:

clipOR An object reference of the **mza::Clip** object to add.

position Absolute position within the **LgCtnt** object at which the **mza::Clip** object is to be inserted.

Returns:

Nothing

Raises:

mza::PersistenceError

See Also:

Add Clips to Logical Content on page 3-21

delClip

Removes the specified **mza::Clip** object from the **LgCtnt** object. The remaining clips are reordered.

```
void delClip(in Clip clipOR);
```

Parameters:

clipOR An object reference of the **mza::Clip** object to delete.

Returns:

Nothing

Raises:

mza::PersistenceError

See Also:

Remove Clips from Logical Content on page 3-22

delClipByPos

Removes an **mza::Clip** object from the specified position in the **LgCtnt** object. The clip is deleted and the remaining clips are reordered.

```
void delClipByPos(in long position);
```

Parameters:

position Absolute position within the **LgCtnt** object of the **mza::Clip** object being removed.

Returns:

Nothing

Raises:

mza::PersistenceError

See Also:

Remove Clips from Logical Content on page 3-22

mza::LgCtntFac

The **LgCtntFac** interface enables you to create new logical content objects.

A logical content object contains zero or more clips. When a new **mza::LgCtnt** object is created using the **mza::LgCtntFac::create()** method, it will have no clips. Clips must be added to the **LgCtnt** object using the **mza::LgCtnt** object interface methods before the **LgCtnt** object is used.

If the method **createCtnt()** is used, an **mza::LgCtnt** object is created with one **mza::Clip** object and one **mza::Ctnt** object. The clip created will reference the entire **mza::Ctnt** object. Using this method, a new logical content object may be created that can be played immediately.

Each clip contained by a logical content object will have a position that denotes its location in the clip list. A clip may be inserted or deleted by position, inserted at the end of the list, or deleted by reference. The first clip in the logical content is located at *position 1*.

LgCtntFac Data Types

The **LgCtntFac** interface uses the **LgCtnt Data Types**.

Methods

The **LgCtntFac** interface provides the following methods:

Method	Description	Page
create	Creates a new **mza::LgCtnt** object.	A-61
createCtnt	Creates a new **mza::LgCtnt** object with one **mza::Clip** object and one **mza::Ctnt** object.	A-62

create

Creates a new **mza::LgCtnt** object. Clips may be added using the **mza::LgCtnt** interface methods.

```
LgCtnt create(in string name,
              in string desc);
```

Parameters:

name The name of the logical content.

desc The description of the logical content (optional).

Returns:

An object reference to the newly created **mza::LgCtnt** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create New Logical Content on page 3-21

Creates a new **mza::LgCtnt** object with one **mza::Clip** object that references an **mza::Ctnt** object from beginning to end.

The name and description for all three object references is extracted from the *name* and *description* values in the **mza::CtntAtr** structure. Note that the **mza::Ctnt** object is created / updated first, then the **mza::Clip** object, and finally the **mza::LgCtnt** object.

- If an **mza::Ctnt** object reference with the same name already exists, its attributes are updated to match those in the new **mza::CtntAtr** structure. If there is no **mza::Ctnt** with this name, a new one is created with the new name and attributes.
- If an **mza::Clip** object reference with the same name exists, it is updated and pointed to the **mza::Ctnt** object that has just been created / updated. If an **mza::Clip** object with the same name does not exist, a new **mza::Clip** object is created with the new name.
- If an **mza::LgCtnt** object with the same name already exists, only the *description* value in **mza::LgCtntAtr** is updated. If an **mza::LgCtnt** object with the new name does not exist, a new **mza::LgCtnt** object reference is created and the new **mza::Clip** object is added to it.

```
LgCtnt createCtnt(in CtntAtr ctntAtr,
                  out Clip clipOR,
                  out Ctnt ctntOR);
```

Parameters:

CtntAtr	The **mza::CtntAtr** structure used to create the new **mza::LgCtnt**, **mza::Clip**, and **mza::Ctnt** objects.
clipOR	An object reference to the new **mza::Clip** object.
ctntOR	An object reference to the new **mza::Ctnt** object.

Returns:

An object reference to the new **mza::LgCtnt** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create New Logical Content on page 3-21

mza::LgCtntMgmt

The **LgCtntMgmt** interface manages groups of **mza::LgCtnt** objects.

LgCtntMgmt Data Types

The **LgCtntMgmt** interface uses **LgCtnt Data Types**.

Methods

The **LgCtntMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mza::LgCtnt** objects in the system.	A-64
lstAtrByNm	Returns the attributes for the specified **mza::LgCtnt** object.	A-65
lstAtrByClipNm	Returns the attributes of all **mza::LgCtnt** objects in the system that are associated with the specified **mza::Clip** object.	A-66
lstAtrByCtntNm	Returns the attributes of all **mza::LgCtnt** objects in the system that are associated with the specified physical content name.	A-67
usingDB	Determines whether the logical content server is currently using a database.	A-68

Returns the attributes of all **mza::LgCtnt** objects in the system.

```
LgCtntAtrLst lstAtr(in boolean longFmt,
                    inout Itr itr);
```

Parameters:

longFmt If true, the clip and content information associated with the **mza::LgCtnt** object is filled in and returned as part of the **mza::LgCtntAtr** structures.

itr On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::LgCtntAtrLst** sequence of **mza::LgCtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::PersistenceError**

See Also:

Logical Content Services without a Database – Stand-alone Mode on page 3-9

Returns the attributes for the specified **mza::LgCtnt** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::LgCtntAtr** structure can be returned for the specified **mza::LgCtnt** object.

```
LgCtntAtrLst lstAtrByNm(in string lgCtntName,
                        in boolean longFmt,
                        inout Itr itr);
```

Parameters:

lgCtntName	The name of the **mza::LgCtnt** object to search for.
longFmt	If true, the clip and content information associated with each **mza::LgCtnt** object is filled in and returned as part of the **mza::LgCtntAtr** structures.
itr	On input, specifies the maximum number of items to return and the number of the item to start with. On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::LgCtntAtrLst** sequence of **mza::LgCtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::BadProhib**, **mza::DataConversion**, **mza::PersistenceError**

See Also:

Query the Content Service on page 2-11

Logical Content Services without a Database – Stand-alone Mode on page 3-9

Returns the attributes of all **mza::LgCtnt** objects in the system that are associated with the specified **mza::Clip** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::LgCtntAtr** structure can be returned for the **mza::LgCtnt** object.

```
LgCtntAtrLst lstAtrByClipNm(in string clipName,
                            in boolean longFmt,
                            inout Itr itr);
```

Parameters:

clipName The name of the **mza::Clip** object to use to search for **mza::LgCtnt** objects.

longFmt If true, the clip and content information associated with each **mza::LgCtnt** object is filled in and returned as part of the **mza::LgCtntAtr** structures.

itr On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::LgCtntAtrLst** sequence of **mza::LgCtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::BadProhib**, **mza::DataConversion**, **mza::PersistenceError**

Returns the attributes of all **mza::LgCtnt** objects in the system that are associated with the specified physical content name. You can use regular expression select-type wildcards: “*” to match any sequence of characters or ”.” to match any single character. If no pattern matching characters are given in the name, only one **mza::LgCtntAtr** structure can be returned for the **mza::LgCtnt** object.

```
LgCtntAtrLst lstAtrByCtntNm(in string ctntName,
                            in boolean longFmt,
                            inout Itr itr);
```

Parameters:

ctntName The name of the physical content to search for **mza::LgCtnt** objects.

longFmt If true, the clip and content information associated with each **mza::LgCtnt** object is filled in and returned as part of the **mza::LgCtntAtr** structures.

itr On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::LgCtntAtrLst** sequence of **mza::LgCtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::BadProhib**, **mza::DataConversion**, **mza::PersistenceError**

usingDB

Determines whether the logical content server is currently using a database. If the server is not connected to a database, only the **lstAtr()** and **lstAtrByNm()** methods in the **mza::LgCtnt** and **mza::Ctnt** interfaces are operational. All other **mza::LgCtnt** and **mza::Ctnt** methods do nothing; they just return without performing any action.

When the content server is running without a database, it will query MDS to find available tag files. It turns these tag files into "pseudo logical content" that consists of one clip that includes the entire tag file. Queries by name must use MDS filename syntax /mds/*volume*/*tagfilename*. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character.

```
boolean usingDB();
```

Returns:

True if connected to a database, false otherwise.

Raises:

Nothing

mza::Ctnt

The **Ctnt** object stores all of the metadata for a piece of content. **Ctnt** objects will typically be created by **vstag** or a real-time encoder as described in Chapter 7.

Ctnt objects by themselves are not played by the stream service. An **mza::Clip** object that references the **Ctnt** object must first be created and added to an **mza::LgCtnt** object. The **mza::LgCtntFac::createCtnt()** method can be used to create all three objects at the same time, resulting in playable content.

Ctnt Data Types

Content data types are used to encapsulate all of the content attributes into one package. The *contAssigned* value in the **mza::CtntAtr** structure indicates if the content is assigned to at least one clip.

The **Ctnt** object and all related objects use the following data types:

Data Type	Description
mza::CtntAtr	Attribute information of an **mza::Ctnt** object.
mza::CtntAtrLst	Sequence of **mza::CtntAtr** structures.
mza::Itr	Structure for iterating through a list.

Methods

The **Ctnt** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Ctnt** object.	A-70
setAtr	Sets the attributes of the **Ctnt** object.	A-70
updateStats	Updates the length, size, and status attributes for the **Ctnt** object.	A-71
updateSugBufSz	Updates the suggested buffer size for the **Ctnt** object.	A-71
updateTimes	Updates the first time and last time attributes.	A-72
destroy	Destroys the **Ctnt** object.	A-72

getAtr

Returns the attributes of the **Ctnt** object.

```
void getAtr(out CtntAtr ctntAtr);
```

Parameters:

ctntAtr The **mza::CtntAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError

setAtr

Sets the attributes of the **Ctnt** object.

```
void setAtr(in CtntAtr ctntAtr);
```

Parameters:

ctntAtr The **mza::CtntAtr** structure containing the attributes of the **Ctnt** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

updateStats

Updates the length, size, and status attributes for the **Ctnt** object.

```
void updateStats(in long len,
                 in long msecs,
                 in long sugBufSz,
                 in string status);
```

Parameters:

len	The total length of the file, in bytes.
msecs	The total duration of the content, in milliseconds.
sugBufSz	The suggested buffer size for the content.
status	The status of the content. Must be either: DISK, TAPE, FEED, ROLLING, TERMINATED, or UNAVAILABLE.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

updateSugBufSz

Updates the suggested buffer size for the **Ctnt** object.

```
void updateSugBufSz(in long sugBufSz);
```

Parameters:

sugBufSz	The suggested buffer size for the content.

Returns:

Nothing

Raises:

mza::PersistenceError

updateTimes

Updates the first time and last time attributes.

```
void updateTimes(in long long firstTime,
                 in long long lastTime);
```

Parameters:

firstTime The time offset of the first tag in the file.

lastTime The time offset of the last tag in the file.

Returns:

Nothing

Raises:

mza::PersistenceError

destroy

Destroys the **Ctnt** object. If the **Ctnt** object is currently used in any **mza::Clip** object, it is not destroyed and an exception is raised.

```
void destroy(in boolean killTagFile,
             in boolean killContent);
```

Parameters:

killTagFile If true, destroys the tag file.

killContent If true, destroys the logical content that the tag file references and destroys the tag file regardless of the *killTagFile* value.

Returns:

Nothing

Raises:

mza::PersistenceError

mza::CtntFac

The **CtntFac** interface enables you to create new content objects.

CtntFac Data Types

The **CtntFac** interface uses **Ctnt Data Types**.

Methods

The **CtntFac** interface provides a single method, **create()**.

create

Creates a new **mza::Ctnt** object.

```
Ctnt create(in CtntAtr ctntAtr);
```

Parameters:

CtntAtr The **mza::CtntAtr** structure containing the attributes for the **mza::Ctnt** object.

Returns:

An object reference to the newly created **mza::Ctnt** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create Content on page 3-17

mza::CtntMgmt

The **CtntMgmt** interface manages groups of **mza::Ctnt** objects.

CtntMgmt Data Types

The **CtntMgmt** interface uses **Ctnt Data Types**.

Methods

The **CtntMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mza::Ctnt** objects in the system.	A-75
lstAtrByNm	Returns the attributes for the specified **mza::Ctnt** object.	A-76
lstAtrByFileNm	Returns attributes of the **mza::Ctnt** object associated with the specified MDS tag file.	A-77

Returns the attributes of all **mza::Ctnt** objects in the system.

```
CtntAtrLst lstAtr(inout Itr itr);
```

Parameters:

itr On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::CtntAtrLst** sequence of **mza::CtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadProhib**, **mza::PersistenceError**

See Also:

List Content on page 3-16

Returns the attributes for the specified **mza::Ctnt** object, which may or may not have the same name as the MDS filename. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::CtntAtr** structure can be returned for the specified **mza::Ctnt** object.

```
CtntAtrLst lstAtrByNm(in string name,
                      inout Itr itr);
```

Parameters:

name	The name of the **mza::Ctnt** object to search for.
itr	On input, specifies the maximum number of items to return and the number of the item to start with. On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::CtntAtrLst** sequence of **mza::CtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadProhib**, **mza::DataConversion**, **mza::PersistenceError**

See Also:

Search for Named Content on page 3-17

Queries the **mza::Ctnt** object's tag file name, which is the same as in the MDS volume. Returns attributes of the **mza::Ctnt** object associated with the specified MDS tag file. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::CtntAtr** structure can be returned for the specified **mza::Ctnt** object.

```
CtntAtrLst lstAtrByFileNm(in string filename,
                          inout Itr itr);
```

Parameters:

filename The name of the MDS tag file to search for.

itr On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::CtntAtrLst** sequence of **mza::CtntAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadProhib**, **mza::DataConversion**, **mza::PersistenceError**

mza::CtntPvdr

The **CtntPvdr** methods assign ownership to an **mza::Ctnt** object. A content provider is an optional parameter for content.

Content providers are usually organizations that develop or own content. For example, a large database of videos may contain content from CNN, United Artists, MCA, CBS, and so on. Each of these companies might be identified as a separate content provider.

CtntPvdr Data Types

The **CtntPvdr** interface uses the following data types:

Data Type	Description
mza::CtntPvdrAtr	Attribute information of an **mza::CtntPvdr** object.
mza::CtntPvdrAtrLst	Sequence of **mza::CtntPvdrAtr** structures.
mza::Itr	Structure for iterating through a list.

Attributes

Attribute	Description
name	The name given to the **CtntPvdr** object.
desc	The description of the **CtntPvdr** object (optional).

Methods

The **CtntPvdr** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **CtntPvdr** object.	A-79
destroy	Destroys the **CtntPvdr** object.	A-79

getAtr

Returns the attributes of the **CtntPvdr** object.

```
void getAtr(out CtntPvdrAtr ctntpvdrAtr);
```

Parameters:

ctntpvdrAtr The **mza::CtntPvdrAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError

destroy

Destroys the **CtntPvdr** object. This method cannot be used to destroy a **CtntPvdr** object referenced by an **mza::Ctnt** object.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

See Also:

Destroy Content Provider on page 3-16

mza::CtntPvdrFac

The **CtntPvdrFac** interface enables you to create new content provider objects.

CtntPvdrFac Data Types

The **CtntPvdrFac** interface uses **CtntPvdr Data Types**.

Methods

The **CtntPvdrFac** interface provides a single method, **create()**.

create

Creates a new **mza::CtntPvdr** object.

```
CtntPvdr create(in string name,
                in string desc);
```

Parameters:

name The name of the **mza::CtntPvdr** object.

desc The description of the **mza::CtntPvdr** object (optional).

Returns:

An object reference to the newly created **mza::CtntPvdr** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create Content Provider and Return its Attributes on page 3-15

mza::CtntPvdrMgmt

The **CtntPvdrMgmt** interface manages groups of **mza::CtntPvdr** objects.

CtntPvdrMgmt Data Types

The **CtntPvdrMgmt** interface uses **CtntPvdr Data Types**.

Methods

The **CtntPvdrMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mza::CtntPvdr** objects in the system.	A-87
getAtrByNm	Returns the attributes for the specified **mza::CtntPvdr** object.	A-82

lstAtr

Returns the attributes of all **mza::CtntPvdr** objects in the system.

```
CtntPvdrAtrLst lstAtr(inout Itr itr);
```

Parameters:

itr — On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::CtntPvdrAtrLst** sequence of **mza::CtntPvdrAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::PersistenceError**

See Also:

List Content Providers on page 3-14

Returns the attributes for the specified **mza::CtntPvdr** object.

```
void getAtrByNm(in string name,
                out CtntPvdrAtr ctntPvdrAtr);
```

Parameters:

name The name of the **mza::CtntPvdr** object.

ctntPvdrAtr The **mza::CtntPvdrAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**

See Also:

Search for Named Content Provider on page 3-15

mza::Clip

Clips are used to create segments of content. A **clip** references a specific **mza::Ctnt** object and adds a start and end position. This way portions of a video can be selected and sequenced together to form a single **mza::LgCtnt** object. Clips may also reference the beginning and end of an **mza::Ctnt** object, indicating that the entire **mza::Ctnt** object be played.

Clip Data Types

The clip data types are used to encapsulate all of the attributes of a clip. A boolean flag indicates if the **Clip** object is assigned to one or more **mza::LgCtnt** objects.

The **Clip** interface uses the following data types:

Data Type	Description
mza::ClipAtr	Attribute information of an **mza::Clip** object.
mza::ClipAtrLst	Sequence of **mza::ClipAtr** structures.
mza::Itr	Structure for iterating through a list.

Attributes

The following attributes can be set and returned for a clip:

Attribute	Description
ctntOR	An object reference to an **mza::Ctnt** object.
name	The name of the clip.
desc	The description of the clip (optional).
startPos	Starting position of the clip within an **mza::Ctnt** object.
stopPos	Ending position for the clip within an **mza::Ctnt** object.

Methods

The **Clip** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Clip** object.	A-84
destroy	Destroys the **Clip** object.	A-84

getAtr

Returns the attributes of the **Clip** object.

```
void getAtr(out ClipAtr clipAtr);
```

Parameters:

clipAtr The **mza::ClipAtr** structure to hold the returned attributes. To allocate memory for the **mza::ClipAtr** structure, call **yoAlloc()**.

Returns:

Nothing

Raises:

mza::PersistenceError

destroy

Destroys the **Clip** object. If the **Clip** object is used in any **mza::LgCtnt** object, it is not destroyed and will raise an exception.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

mza::ClipFac

The **ClipFac** interface enables you to create new clip objects.

ClipFac Data Types

The **ClipFac** interface uses **Clip Data Types**.

Methods

The **ClipFac** interface provides a single method, **create()**.

create

Creates a new **mza::Clip** object.

```
Clip create(in Ctnt ctnt,
            in string name,
            in string desc,
            in mkd::pos startPos,
            in mkd::pos stopPos);
```

Parameters:

ctnt	An object reference to the **mza::Ctnt** object for the **mza::Clip** object.
name	The name of the new **mza::Clip** object.
desc	The description of the **mza::Clip** object (optional).
startPos	An **mkd::pos** value indicating the start time of the clip from the beginning of the **mza::Ctnt** object.
stopPos	An **mkd::pos** value indicating the stop time of the clip from the beginning of the **mza::Ctnt** object.

Returns:

An object reference to the newly created **mza::Clip** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

mza::ClipMgmt

The **ClipMgmt** interface manages groups of **mza::Clip** objects.

ClipMgmt Data Types

The **ClipMgmt** interface uses **Clip Data Types**.

Methods

The **ClipMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mza::Clip** objects managed by the **ClipMgmt** object.	A-87
lstAtrByCtnt	Returns the attributes of all **mza::Clip** objects in the system that are associated with the specified **mza::Ctnt** object.	A-88
lstAtrByNm	Returns the attributes for the specified **mza::Clip** object.	A-89
lstAtrByCtntNm	Returns the attributes of all **mza::Clip** objects in the system that are associated with the specified physical content name.	A-90

Returns the attributes of all **mza::Clip** objects managed by the **ClipMgmt** object.

```
ClipAtrLst lstAtr(inout Itr iterator);
```

Parameters:

iterator On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::ClipAtrLst** sequence of **mza::ClipAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::PersistenceError**

See Also:

Locating Clips by Name on page 3-20

Returns the attributes of all **mza::Clip** objects in the system that are associated with the specified **mza::Ctnt** object.

```
ClipAtrLst lstAtrByCtnt(in Ctnt ctnt,
                        inout Itr iterator);
```

Parameters:

ctnt An object reference to an **mza::Ctnt** object. Only the attributes of **mza::Clip** objects using the **mza::Ctnt** object are returned.

iterator On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::ClipAtrLst** sequence of **mza::ClipAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::PersistenceError**

Returns the attributes for the specified **mza::Clip** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::ClipAtr** structure can be returned for the specified **mza::Clip** object.

```
ClipAtrLst lstAtrByNm(in string name,
                      inout Itr iterator);
```

Parameters:

name The name of the **mza::Clip** object to search for.

iterator On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::ClipAtrLst** sequence of **mza::ClipAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises

mza::BadIterator, **mza::BadPosition**, **mza::DataConversion**, **mza::PersistenceError**

lstAtrByCtntNm

Returns the attributes of all **mza::Clip** objects in the system that are associated with the specified physical content name. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mza::ClipAtrLst** structure can be returned for the **mza::Clip** object.

```
ClipAtrLst lstAtrByCtntNm(in string ctntName,
                          inout Itr iterator);
```

Parameters:

ctntName	The name of the physical content to search for **mza::Clip** objects.
itr	On input, specifies the maximum number of items to return and the number of the item to start with. On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::ClipAtrLst** sequence of **mza::ClipAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::BadPosition**, **mza::DataConversion**, **mza::PersistenceError**

mza::BlobMgmt

The **BlobMgmt** interface manages groups of BLOBs. This is the primary interface for retrieving MDS-style asset cookies from the content service.

Methods

The **BlobMgmt** interface provides a single method, **lstAtrByNm()**.

Returns the attributes for the specified BLOB. The BLOB name must be an MDS-style file name, such as **/mds/video/*.mpg** or **/mds/*/***. The BLOB name can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters given in the name, this method returns only an exact match. Be careful when using wildcards, as a large number of files may be returned and it may take a long time to query MDS.

Since the data returned in the **mza::LgCtntAtr** structures are for BLOBs, they contain none of the long format information. This means only the name and asset cookie data is filled in. The asset cookies returned are MDS style, the *longFmt* will always be false, and *numClips* is meaningless.

```
LgCtntAtrLst lstAtrByNm(in string blobName,
                        inout Itr itr);
```

Parameters:

blobName	The name of the BLOB.
itr	On input, specifies the maximum number of items to return and the number of the item to start with. On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass. See **mza::Itr**.

Returns:

An **mza::LgCtntAtrLst** sequence of **mza::LgCtntAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises

mza::DataConversion, **mza::PersistenceError**

mzabi — Schedule Interfaces

The following Schedule interfaces are defined in the **mzabi.idl** file:

- **mzabi::Schd**, **mzabi::SchdFac**, **mzabi::SchdMgmt**
- **mzabi::Exp**, **mzabi::ExpFac**, **mzabi::ExpMgmt**
- **mzabi::ExpGrp**, **mzabi::ExpGrpFac**, **mzabi::ExpGrpMgmt**

The **Schd** interface allows you to get, set, and destroy schedule data. It also allows you to create new scheduled events and specify which implementation-specific services (such as the NVOD exporter service) are to handle the scheduled events when they occur.

Each scheduled event is exported to an implementation of the **exporter** interface on A-130, which invokes an operation on behalf of the scheduled event. The exporter group (**ExpGrp**) and exporter (**Exp**) objects export the scheduled event to its target implementation.

mzabi::Schd

The **Schd** object provides the interface to schedule events.

Schd Data Types

The **Schd** interface uses the following data types:

Data Type	Description
mzabi::SchdAtr	Attribute information of an **mzabi::Schd** object.
mzabi::SchdAtrLst	Sequence of **mzabi::SchdAtr** structures.
mzabi::schdStatus	Type describing the status of a scheduled event.
mza::Itr	Structure for iterating through a list.

Methods

The **Schd** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Schd** object.	A-93
setAtr	Sets the attributes of the **Schd** object.	A-94
setStatus	Sets the status of the **Schd** object.	A-94
destroy	Destroys the **Schd** object.	A-95

getAtr

Returns the attributes of the **Schd** object.

```
void getAtr(out SchdAtr schdAtr);
```

Parameters:

schdAtr The **mzabi::SchdAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError, **mzabi::badEventType**, **mzabi::badStatus**

setAtr

Sets the attributes of the **Schd** object.

```
void setAtr(in SchdAtr schdAtr);
```

Parameters:

schdAtr The **mzabi::SchdAtr** structure containing the attributes of the **Schd** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**, **mzabi::badEventType**, **mzabi::badStatus**

setStatus

Sets the status of the **Schd** object.

```
void setStatus(in schdStatus sts);
```

Parameters:

status An **mzabi::schdStatus** value that identifies the current status of the **Schd** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**, **mzabi::badStatus**

destroy

Destroys the **Schd** object. A scheduled event may not be destroyed if the status is **exported_schdStatus**, **exporting_schdStatus**, **finishing_schdStatus**, or **started_schdStatus**.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

mzabi::SchdFac

The **SchdFac** interface enables you to create new scheduled event objects.

SchdFac Data Types

The **SchdFac** interface uses **Schd Data Types**.

Methods

The **SchdFac** interface provides a single method, **create()**.

Creates a new **mzabi::Schd** object. New scheduled events will get created with a status of **scheduled_schdStatus**.

```
Schd create(in mkd::gmtWall startDate,
            in mkd::gmtWall stopDate,
            in ExpGrp expGrpOR,
            in ExpGrp chgGrpOR,
            in eventType type);
```

Parameters:

startDate An **mkd::gmtWall** value indicating the start time and date of the scheduled event, in Greenwich Mean Time (GMT).

stopDate An **mkd::gmtWall** value indicating the stop time and date of the scheduled event, in Greenwich Mean Time (GMT).

expGrpOR An object reference to the **mzabi::ExpGrp** object that should be informed of the scheduled event.

chgGrpOR An object reference to the **mzabi::ExpGrp** object that should be informed if the scheduled event changes.

type An **mzabi::eventType** value that indicates the type of event scheduled.

Returns:

An object reference to the newly created **mzabi::Schd** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create a Schedule on page 4-8

mzabi::SchdMgmt

The **SchdMgmt** interface manages groups of **mzabi::Schd** objects.

SchdMgmt Data Types

The **SchdMgmt** interface uses **Schd Data Types**.

Methods

The **SchdMgmt** interface provides the following methods:

Method	Description	Page
lstAtrByDate	Returns the attributes of all scheduled events to be started during the specified time period.	A-98
lstAtrChangedEvents	Returns the attributes of all **mzabi::Schd** objects that have changed.	A-99
lstAtrActiveEvents	Returns the attributes of all scheduled events that should be active.	A-100
lstAtrActiveEventsByExpGrp	Returns the attributes of all scheduled events that should be active for the specified **mzabi::ExpGrp** object.	A-101

Returns the attributes of all scheduled events to be started during the specified time period. This method returns all of the information about the scheduled events that need to be started in the time period that starts at *startDate* for the time duration specified in *secs*.

```
SchdAtrLst lstAtrByDate(in mkd::gmtWall startDate,
                        in long secs,
                        in schdStatus sts,
                        in boolean exporting,
                        inout mza::Itr itr);
```

Parameters:

startDate — An **mkd::gmtWall** value indicating the start time and date for which information is being requested, in Greenwich Mean Time (GMT).

secs — The duration in seconds from the time specified in *startDate* for which data is being requested.

sts — An **mzabi::schdStatus** value that specifies which events are to be returned, based on their current status. A value of **unknown_schdStatus** returns all events regardless of status. Any other **mzabi::schdStatus** value returns only those scheduled events associated with that status.

exporting — If this boolean is true, then the server will mark those scheduled events returned as having a status of **exporting_schdStatus**. This should only be set true by the generic Scheduler when reading scheduled events for the purpose of exporting them.

itr — The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::SchdAtrLst** sequence of **mzabi::SchdAtr** structures for all scheduled events in the time period *startDate* + *secs*.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badEventType**, **mzabi::badStatus**

lstAtrChangedEvents

Returns the attributes of all **mzabi::Schd** objects that have changed. A "changed" **mzabi::Schd** object is one that has been updated while active (currently used by an exporter, but not yet finished).

```
SchdAtrLst    lstAtrChangedEvents(inout mza::Itr itr);
```

Parameters

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::SchdAtrLst** sequence of **mzabi::SchdAtr** structures for all scheduled events that have changed.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badEventType**, **mzabi::badStatus**

lstAtrActiveEvents

Returns the attributes of all scheduled events that should be active.

```
SchdAtrLst    lstAtrActiveEvents(inout mza::Itr itr);
```

Parameters:

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::SchdAtrLst** sequence of **mzabi::SchdAtr** structures for all scheduled events that should be active.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badEventType**, **mzabi::badStatus**

lstAtrActiveEventsByExpGrp

Returns the attributes of all scheduled events that should be active for the specified **mzabi::ExpGrp** object.

```
SchdAtrLst    lstAtrActiveEventsByExpGrp(in ExpGrp expGrpOR,
                                         inout mza::Itr itr);
```

Parameters:

expGrpOR The **mzabi::ExpGrp** object for which information is being requested.

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::SchdAtrLst** sequence of **mzabi::SchdAtr** structures for all scheduled events that should be active for the specified **mzabi::ExpGrp** object.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badEventType**, **mzabi::badStatus**

mzabi::Exp

The **Exp** object provides a common interface for passing scheduled events from the **mzabi::Schd** object to the specific **mzabix::exporter** object designated to respond to the event.

Exp Data Types

The **Exp** interface uses the following data types:

Data Type	Description
mzabi::ExpAtr	Attribute information of an **mzabi::Exp** object.
mzabi::ExpAtrLst	Sequence of **mzabi::ExpAtr** structures.
mzabi::expStatus	Status of an **mzabi::Exp** object.
mza::Itr	Structure for iterating through a list.

Methods

The **Exp** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Exp** object.	A-103
setAtr	Sets the attributes of the **Exp** object.	A-103
setStatus	Sets the status of the **Exp** object.	A-104
destroy	Destroys the **Exp** object.	A-104

getAtr

Returns the attributes of the **Exp** object.

```
void getAtr(out ExpAtr exporterAtr);
```

Parameters:

exporterAtr The **mzabi::ExpAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError, **mzabi::badStatus**

setAtr

Sets the attributes of the **Exp** object.

```
void setAtr(in ExpAtr exporterAtr);
```

Parameters:

exporterAtr The **mzabi::ExpAtr** structure containing the attributes of the **Exp** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**, **mzabi::badStatus**

setStatus

Sets the status of the **Exp** object.

```
void setStatus(in expStatus sts);
```

Parameters:

sts	An **mzabi::expStatus** value that identifies the current status of the **Exp** object.

Returns:

Nothing

Raises:

mza::PersistenceError, **mzabi::badStatus**

destroy

Destroys the **Exp** object. **Exp** objects currently referenced by an **mzabi::ExpGrp** object cannot be destroyed. First use the **mzabi::ExpGrp::remExp()** or **mzabi::ExpGrp::remExpByNm()** method to remove the **Exp** object from the group, or the **mzabi::ExpGrp::destroy()** method to destroy the **mzabi::ExpGrp** object.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

mzabi::ExpFac

The **ExpFac** interface enables you to create new exporter objects.

ExpFac Data Types

The **ExpFac** interface uses **Exp Data Types**.

Methods

The **ExpFac** interface provides a single method, **create()**.

create

Creates a new **mzabi::Exp** object.

```
Exp create(in string name,
           in string implId,
           in long setupTime);
```

Parameters:

name	An unique name identifying the new exporter.
implID	The *implId* of the **exporter** object to bind to when making calls to the **mzabix::exporter** interface methods.
setupTime	The time in microseconds the exporter needs to set up prior to calling the **startEvent** method.

Returns:

An object reference to the newly created **mzabi::Exp** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create an Exp Object on page 4-13

mzabi::ExpMgmt

The **ExpMgmt** interface manages groups of **mzabi::Exp** objects.

ExpMgmt Data Types

The **ExpMgmt** interface uses **Exp Data Types**.

Methods

The **ExpMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mzabi::Exp** objects in the system.	A-106
lstAtrByNm	Returns the attributes for the specified **mzabi::Exp** object.	A-107

lstAtr

Returns the attributes of all **mzabi::Exp** objects in the system.

```
ExpAtrLst lstAtr(inout mza::Itr itr);
```

Parameters:

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::ExpAtrLst** sequence of **mzabi::ExpAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::PersistenceError**, **mzabi::badStatus**

Returns the attributes for the specified **mzabi::Exp** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mzabi::ExpAtrLst** structure can be returned for the specified **mzabi::Exp** object.

```
ExpAtrLst lstAtrByNm(in string name,
                     inout mza::Itr itr);
```

Parameters:

name The name of the **mzabi::Exp** object to search for.

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::ExpAtrLst** sequence of **mzabi::ExpAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badStatus**

mzabi::ExpGrp

The **ExpGrp** object provides the interface to exporter groups.

ExpGrp Data Types

The **ExpGrp** interface uses the following data types:

Data Type	Description
mzabi::ExpAtr	Attribute information of an **mzabi::Exp** object.
mzabi::ExpAtrLst	Sequence of **mzabi::ExpAtr** structures.
mzabi::ExpGrpAtr	Attribute information of an **mzabi::ExpGrp** object.
mzabi::ExpGrpAtrLst	Sequence of **mzabi::ExpGrpAtr** structures.
mza::Itr	Structure for iterating through a list.

Methods

The **ExpGrp** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **ExpGrp** object.	A-109
setAtr	Sets the attributes of the **ExpGrp** object.	A-109
destroy	Destroys the **ExpGrp** object.	A-110
addExp	Adds an **mzabi::Exp** object to the **ExpGrp** object.	A-110
addExpByNm	Given the name of an **mzabi::Exp** object, adds the **mzabi::Exp** object to the **ExpGrp** object.	A-111
remExp	Removes an **mzabi::Exp** object from the **ExpGrp** object.	A-111
remExpByNm	Removes the specified **mzabi::Exp** object from the **ExpGrp** object.	A-112
lstAtrExps	Returns the attributes of all **mzabi::Exp** objects in the **ExpGrp** object.	A-112

getAtr

Returns the attributes of the **ExpGrp** object.

```
void getAtr(out ExpGrpAtr exporterAtr);
```

Parameters:

exporterAtr The **mzabi::ExpGrpAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError

setAtr

Sets the attributes of the **ExpGrp** object.

```
void setAtr(in ExpGrpAtr exporterAtr);
```

Parameters:

exporterAtr The **mzabi::ExpGrpAtr** structure containing the attributes of the **ExpGrp** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

destroy

Destroys the **ExpGrp** object.

The **ExpGrp** object cannot be destroyed if referencing any **mzabi::Exp** objects, or if referenced by an **mzabi::Schd** object. If the **ExpGrp** object to be destroyed is referencing one or more **mzabi::Exp** objects, then use the **mzabi::ExpGrp::remExp()** or **mzabi::ExpGrp::remExpByNm()** method to remove the **Exp** objects from the group. If the **ExpGrp** object is referenced by an **mzabi::Schd** object, use **mzabi::Schd::destroy()** to destroy the **mzabi::Schd** object.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

addExp

Adds an **mzabi::Exp** object to the **ExpGrp** object.

```
void addExp(in Exp exporterOR);
```

Parameters:

exporterOR The **mzabi::Exp** object to add.

Returns:

Nothing

Raises:

mza::PersistenceError

addExpByNm

Given the name of an **mzabi::Exp** object, adds the **mzabi::Exp** object to the **ExpGrp** object.

```
void addExpByNm(in string name);
```

Parameters:

schdAtr The name of the **mzabi::Exp** object to add.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

remExp

Removes an **mzabi::Exp** object from the **ExpGrp** object.

```
void remExp(in Exp exporterOR);
```

Parameters:

exporterOR The **mzabi::Exp** object to remove.

Returns:

Nothing

Raises:

mza::PersistenceError

remExpByNm

Removes the specified **mzabi::Exp** object from the **ExpGrp** object.

```
void remExpByNm(in string name);
```

Parameters:

name The name of the **mzabi::Exp** object to remove.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

lstAtrExps

Returns the attributes of all **mzabi::Exp** objects in the **ExpGrp** object.

```
ExpAtrLst lstAtrExps(inout mza::Itr iterator);
```

Parameters:

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::ExpAtrLst** sequence of **mzabi::ExpAtr** structures in the **ExpGrp** object.

Raises:

mza::BadIterator, **mza::PersistenceError**

mzabi::ExpGrpFac

The **ExpGrpFac** interface enables you to create new exporter group objects.

ExpGrpFac Data Types

The **ExpGrpFac** interface uses **ExpGrp Data Types**.

Methods

The **ExpGrpFac** interface provides a single method, **create()**.

create

Creates a new **mzabi::ExpGrp** object.

```
ExpGrp create(in string name);
```

Parameters:

name An unique name identifying the new exporter group.

Returns:

An object reference to the newly created **mzabi::ExpGrp** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create an ExpGrp Object on page 4-13

mzabi::ExpGrpMgmt

The **ExpGrpMgmt** interface manages groups of **mzabi::ExpGrp** objects.

ExpGrpMgmt Data Types

The **ExpGrpMgmt** interface uses **ExpGrp Data Types**.

Methods

The **ExpGrpMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mzabi::ExpGrp** objects in the system.	A-114
lstAtrByNm	Returns the attributes for the specified **mzabi::ExpGrp** object.	A-114

lstAtr

Returns the attributes of all **mzabi::ExpGrp** objects in the system.

```
ExpGrpAtrLst lstAtr(inout mza::Itr itr);
```

Parameters:

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::ExpGrpAtrLst** sequence of **mzabi::ExpGrpAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::PersistenceError**, **mzabi::badStatus**

Returns the attributes for the specified **mzabi::ExpGrp** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mzabi::ExpGrpAtrLst** structure can be returned for the specified **mzabi::ExpGrp** object.

```
ExpGrpAtrLst lstAtrByNm(in string name,
                        inout mza::Itr itr);
```

Parameters:

name The name of the **mzabi::ExpGrp** object to search for.

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabi::ExpGrpAtrLst** sequence of **mzabi::ExpGrpAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badStatus**

mzabin — Content Broadcast Interfaces

The following Content Broadcast interfaces are defined in the **mzabin.idl** file:

- **mzabin::Chnl**, **mzabin::ChnlFac**, **mzabin::ChnlMgmt**
- **mzabin::Nvod**, **mzabin::NvodFac**, **mzabin::NvodMgmt**

The near video-on-demand (NVOD) interface is an example of an implementation-specific service that handles scheduled events when they occur. The NVOD exporter service (**vsnvodsrv**) allows you to broadcast OVS logical content on defined channels. The NVOD interface enables the NVOD service to access the channel and logical content from the database. You can implement your own implementation-specific services that work either along with or independently of the NVOD service.

The NVOD service must implement the **exporter** interface on A-130 to be initiated by the scheduler service (**vsschdsrv**).

Note The use of "near video-on-demand" and "NVOD" are misnomers perpetuated from an earlier architecture of the Oracle Video Server. The **vsnvodsrv** service is better described as a "playout service" that plays selected logical content on a selected channel. While you can use the **vsnvodsrv** service to schedule content for near video-on-demand playout, you can also schedule content for pay-per-view (PPV) and for regular TV broadcasting. For consistency, the term "NVOD service" is used throughout this chapter, but this is not to imply that the **vsnvodsrv** service is exclusive to near video-on-demand operations.

mzabin::Chnl

Implements the functionality of the **Chnl** object.

Chnl Data Types

The **Chnl** interface uses the following data types:

Data Type	Description
mzabin::ChnlAtr	Attribute information of an **mzabin::Chnl** object.
mzabin::ChnlAtrLst	Sequence of **mzabin::ChnlAtr** structures.
mza::Itr	Structure for iterating through a list.

Methods

The **Chnl** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Chnl** object.	A-117
setAtr	Sets the attributes of the **Chnl** object.	A-118
destroy	Destroys the **Chnl** object.	A-118

getAtr

Returns the attributes of the **Chnl** object.

```
void getAtr(out ChnlAtr chnlAtr);
```

Parameters:

chnlAtr The **mzabin::ChnlAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError

setAtr

Sets the attributes of the **Chnl** object.

```
void setAtr(in ChnlAtr chnlAtr);
```

Parameters:

chnlAtr The **mzabin::ChnlAtr** structure containing the attributes of the **Chnl** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

destroy

Destroys the **Chnl** object. If the **Chnl** object is used in any **mzabin::Nvod** object, it is not destroyed and will raise an exception.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::PersistenceError

mzabin::ChnlFac

The **ChnlFac** interface enables you to create new channel objects.

ChnlFac Data Types

The **ChnlFac** interface uses **Chnl Data Types**.

Methods

The **ChnlFac** interface provides a single method, **create()**.

create

Creates a new **mzabin::Chnl** object.

```
Chnl create(in string label,
            in string netAddr,
            in string chnlNum);
```

Parameters:

label The name of the new **mzabin::Chnl** object.

netAddr The physical network address of the channel.

chnlNum The number for the new **mzabin::Chnl** object.

Returns:

An object reference to the newly created **mzabin::Chnl** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create a Channel on page 4-10

mzabin::ChnlMgmt

The **ChnlMgmt** interface manages groups of **mzabin::Chnl** objects.

ChnlMgmt Data Types

The **ChnlMgmt** interface uses **Chnl Data Types**.

Methods

The **ChnlMgmt** interface provides the following methods:

Method	Description	Page
lstAtr	Returns the attributes of all **mzabin::Chnl** objects in the system.	A-120
lstAtrByNm	Returns the attributes for the specified **mzabin::Chnl** object.	A-121

lstAtr

Returns the attributes of all **mzabin::Chnl** objects in the system.

```
ChnlAtrLst lstAtr(inout mza::Itr itr);
```

Parameters:

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabin::ChnlAtrLst** sequence of **mzabin::ChnlAtr** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mza::BadIterator, **mza::PersistenceError**

Returns the attributes for the specified **mzabin::Chnl** object. You can use regular expression select-type wildcards: "*" to match any sequence of characters or "." to match any single character. If no pattern matching characters are given in the name, only one **mzabin::ChnlAtr** structure can be returned for the specified **mzabin::Chnl** object.

```
ChnlAtrLst lstAtrByNm(in string      name,
                      inout mza::Itr iterator);
```

Parameters:

name The name of the **mzabin::Chnl** object to search for.

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabin::ChnlAtrLst** sequence of **mzabin::ChnlAtr** structures. If no objects are found, an empty list is returned and no exceptions are raised.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**

mzabin::Nvod

The **Nvod** object provides access to the **mzabi::Schd**, **mza::LgCtnt**, and **mzabin::Chnl** information associated with an NVOD event.

Note Like the NVOD service, the name "Nvod" object is a misnomer. The purpose of the **Nvod** object is to define a specific piece of logical content to be played on a specific broadcast channel. While the **Nvod** object can be used for near video-on-demand operations, it is not exclusive to these operations.

Nvod Data Types

The **Nvod** interface uses the following data types:

Data Type	Description
mzabin::NvodAtr	Attribute information of an **mzabin::Nvod** object.
mzabin::NvodAtrLst	Sequence of **mzabin::NvodAtr** structures.
mzabin::NvodSchdAtr	The logical content, channel, and scheduling information associated with an **mzabin::Nvod** object.
mzabin::NvodSchdAtrLst	Sequence of **mzabin::NvodSchdAtr** structures.
mzabin::nvodStatus	Status of an NVOD object.
mza::Itr	Structure for iterating through a list.

Methods

The **Nvod** interface provides the following methods:

Method	Description	Page
getAtr	Returns the attributes of the **Nvod** object.	A-123
setAtr	Sets the attributes of the **Nvod** object.	A-123
setStatus	Sets the status of the **Nvod** object.	A-124
destroy	Destroys the **Nvod** object.	A-124

getAtr

Returns the attributes of the **Nvod** object.

```
void getAtr(out NvodAtr nvodAtr);
```

Parameters:

nvodAtr The **mzabin::NvodAtr** structure to hold the returned attributes.

Returns:

Nothing

Raises:

mza::PersistenceError, **mzabi::badStatus**

setAtr

Sets the attributes of the **Nvod** object.

```
void setAtr(in NvodAtr nvodAtr);
```

Parameters:

nvodAtr The **mzabin::NvodAtr** structure containing the attributes of the **Nvod** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**, **mzabi::badStatus**

setStatus

Sets the status of the **Nvod** object.

```
void setStatus(in nvodStatus sts);
```

Parameters:

sts	An **mzabin::nvodStatus** value that identifies the current status of the **Nvod** object.

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**, **mzabi::badStatus**

destroy

Destroys the **Nvod** object.

```
void destroy();
```

Returns:

Nothing

Raises:

mza::DataConversion, **mza::PersistenceError**

mzabin::NvodFac

The **NvodFac** interface enables you to create new NVOD objects.

NvodFac Data Types

The **NvodFac** interface uses **Nvod Data Types**.

Methods

The **NvodFac** interface provides the following methods:

Method	Description	Page
create	Creates a new **mzabin::Nvod** object.	A-125
createSchd	Creates new **mzabin::Nvod** and **mzabi::Schd** objects.	A-126

create

Creates a new **mzabin::Nvod** object.

```
Nvod create(in mzabi::Schd schdOR,
            in Chnl chnlOR,
            in mza::LgCtnt lgCtntOR,
            in loopType loop);
```

Parameters:

schdOR An object reference to the **mzabi::Schd** object the **mzabin::Nvod** object refers to.

chnlOR An object reference to the **mzabin::Chnl** object the **mzabin::Nvod** object will play to.

lgCtntOR An object reference to the **mza::LgCtnt** object to be played by the **mzabin::Nvod** object.

loop An **mzabin::loopType** value that identifies the type of looping desired.

Returns:

An object reference to the newly created **mzabin::Nvod** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create an Nvod Object on page 4-11

Creates a new **mzabin::Nvod** object and an associated **mzabi::Schd** object.

This method uses defaults where possible and allows for creation of more than one scheduled event and NVOD object at a time using the *interval* parameter. The **mzabi::Schd** object created for the scheduled event uses the given *startDate*, a *stopDate* of NULL, an *eventType* of **synchShort**, the default NVOD exporter group, no exporter change group, and a status of **committed_schdStatus**.

Note The default **mzabi::ExpGrp** for the NVOD exporter is called *Default NVOD Exporter Group*, and is provided as part of the initial database schema install. This default **mzabi::ExpGrp** contains an **mzabi::Exp** object, called *Default NVOD Exporter*, and has an *implId* of **vsnvodsrv**.

The returned **mzabin::Nvod** object references the **mzabi::Schd** object for the new scheduled event. If *interval* is 0 or *stopDate* of NULL, only one scheduled event is created, otherwise multiple scheduled events are created.

```
Nvod createSchd(in Chnl chnlOR,
                in mza::LgCtnt lgCtntOR,
                in mkd::gmtWall startDate,
                in loopType loop,
                in mkd::gmtWall stopDate,
                in long interval,
                out mzabi::Schd schdOR);
```

Parameters:

chnlOR	An object reference to the **mzabin::Chnl** object the **mzabin::Nvod** object will play to.
lgCtntOR	An object reference to the **mza::LgCtnt** object to be played by the **mzabin::Nvod** object.
startDate	An **mkd::gmtWall** value indicating the start time and date of the logical content will be played, in Greenwich Mean Time (GMT).
loop	An **mzabin::loopType** value that identifies the type of looping desired.
stopDate	An **mkd::gmtWall** value indicating the stop time and date of the scheduled event, in Greenwich Mean Time (GMT). When creating a series of scheduled events, set *stopDate* to the last date new scheduled events are to be created. To create only one scheduled event, set *stopDate* to NULL or an empty string.

interval If *stopDate* is non-NULL, scheduled events are created every *interval* seconds until the *stopDate* is reached. If *interval* is 0 in any case, only 1 scheduled event is created.

schdOR An object reference to the **mzabi::Schd** object created. If multiple scheduled events were created, this is the last one.

Returns:

An object reference to the newly created **mzabin::Nvod** object.

Raises:

mza::DataConversion, **mza::PersistenceError**

See Also:

Create a Schedule on page 4-8

mzabin::NvodMgmt

The **NvodMgmt** interface manages groups of **mzabin::Nvod** objects.

NvodMgmt Data Types

The **NvodMgmt** interface uses **Nvod Data Types**.

Methods

The **NvodMgmt** interface provides the following methods:

Method	Description	Page
lstAtrByDate	Returns a list of information about content to be played during the specified time period.	A-128
lstAtrByLCNm	Returns a list of information about content to be played during the specified time period for the specified **mza::LgCtnt** object.	A-129
lstAtrBySchd	Returns a list of information about content to be played for the specified **mzabi::Schd** object.	A-130

Returns a list of information about content to be played during the specified time period. This method returns all of the information about the scheduled content that needs to be played in the time period that starts at *startDate* for the time duration specified in *secs*.

```
NvodSchdAtrLst lstAtrByDate(in mkd::gmtWall startDate,
                            in long secs,
                            in nvodStatus sts,
                            inout mza::Itr itr);
```

Parameters:

startDate — An **mkd::gmtWall** value indicating the start time and date for which information is being requested, in Greenwich Mean Time (GMT).

secs — The duration in seconds from the time specified in *startDate* for which data is being requested.

sts — An **mzabin::nvodStatus** value that specifies which events are to be returned, based on their current status. A value of **unknown_nvodStatus** returns all events regardless of status. Any other **mzabin::nvodStatus** value returns only those scheduled events associated with that status.

itr — The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabin::NvodSchdAtrLst** sequence of **mzabin::NvodSchdAtr** structures for all items scheduled in the time period *startDate* + *secs*.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badStatus**, **mzabin::badLoop**

lstAtrByLCNm

Returns a list of information about content to be played during the specified time period for the specified **mza::LgCtnt** object. This method returns all of the information about the scheduled content that needs to be played in the time period that starts at *startDate* for the time duration specified in *secs*.

```
NvodSchdAtrLst    lstAtrByLCNm(in string lgCtntNm,
                               in mkd::gmtWall startDate,
                               in long secs,
                               in nvodStatus sts,
                               inout mza::Itr itr);
```

Parameters

lgCtntNm	The name of the **mza::LgCtnt** object for which to return the information.
startDate	An **mkd::gmtWall** value indicating the start time and date for which information is being requested, in Greenwich Mean Time (GMT).
secs	The duration in seconds from the time specified in *startDate* for which data is being requested.
sts	An **mzabin::nvodStatus** value that specifies which events are to be returned, based on their current status. A value of **unknown_nvodStatus** returns all events regardless of status. Any other **mzabin::nvodStatus** value returns only those scheduled events associated with that status.
itr	The **mza::Itr** structure that specifies the position and number of items. On input, specifies the maximum number of items to return and the number of the item to start with. On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabin::NvodSchdAtrLst** sequence of **mzabin::NvodSchdAtr** structures for all items scheduled in the time period *startDate* + *secs*.

Raises:

mza::BadIterator, **mza::DataConversion**, **mza::PersistenceError**, **mzabi::badStatus**, **mzabin::badLoop**

Returns a list of information about content to be played for the specified **mzabi::Schd** object. There is a one-to-one mapping between an **mzabi::Schd** object and an **mzabin::Nvod** object.

```
NvodSchdAtrLst    lstAtrBySchd(in mzabi::Schd schdOR,
                               inout mza::Itr itr);
```

Parameters

schdOR An object reference to the **mzabi::Schd** object for which to return the information.

itr The **mza::Itr** structure that specifies the position and number of items.

On input, specifies the maximum number of items to return and the number of the item to start with.

On output, provides updated values for the number of items actually returned and the number of the item to start from on the next pass.

Returns:

An **mzabin::NvodSchdAtrLst** sequence of **mzabin::NvodSchdAtr** structures for all items scheduled in the time period *startDate* + *secs*.

Raises:

mza::BadIterator, **mza::PersistenceError**, **mzabi::badStatus**, **mzabin::badLoop**

mzabix — Schedule Exporter Interface

The Schedule Exporter interface is defined in the **mzabix.idl** file:

- **mzabix::exporter**

The **exporter** interface allows you to start, stop, and change scheduled events. Each implementation-specific service (such as the NVOD exporter service) must implement the **exporter** interface to be initiated by the scheduler service (**vsschdsrv**).

mzabix::exporter

Implements the functionality of the **exporter** object.

exporter Data Types

The **exporter** interface uses the following data types:

Data Type	Description
mzabix::statusInfo	Status of an **mzabix::exporter** object.
mzabix::statusInfoLst	Sequence of **mzabix::statusInfo** structures.
mzabix::eventStatusInfo	Status of a scheduled event the **mzabix::exporter** object is exporting.
mzabix::eventStatusInfoLst	Sequence of **mzabix::eventStatusInfo** structures.
mzabi::schdStatus	Status of an **mzabi::Schd** object.

Methods

The **exporter** interface provides the following methods:

Method	Description	Page
startEvent	Starts the specified scheduled event at the given time.	A-132
stopEvent	Stops the specified scheduled event at the given time.	A-133
changeEvent	The specified scheduled event has changed.	A-133
getStatus	Returns the status of the **exporter** object.	A-134
getEventStatus	Returns the status of the specified scheduled event the **exporter** object is exporting.	A-134
getAllEventStatus	Returns the status of all scheduled events the **exporter** object is exporting.	A-135

Starts the specified scheduled event at the given time.

```
mzabi::schdStatus startEvent(in mzabi::Schd schdOR,
                             in mkd::gmtWall startTime,
                             in mkd::gmtWall stopTime);
```

Parameters:

schdOR An object reference to the **mzabi::Schd** object describing the scheduled event.

startTime An **mkd::gmtWall** value indicating the start time and date of the scheduled event, in Greenwich Mean Time (GMT).

stopTime An **mkd::gmtWall** value indicating the stop time and date of the scheduled event, in Greenwich Mean Time (GMT).

Returns:

An **mzabi::schdStatus** value that identifies the current status of the **mzabi::Schd** object.

Raises:

mzabix::badEvent

stopEvent

Stops the specified scheduled event at the given time.

```
mzabi::schdStatus stopEvent(in mzabi::Schd schdOR,
                            in mkd::gmtWall stopTime);
```

Parameters:

schdOR An object reference to the **mzabi::Schd** object describing the scheduled event.

stopTime An **mkd::gmtWall** value indicating the start time and date of the scheduled event, in Greenwich Mean Time (GMT).

Returns:

An **mzabi::schdStatus** value that identifies the current status of the **mzabi::Schd** object.

Raises:

mzabix::badEvent

changeEvent

The specified scheduled event has changed.

```
mzabi::schdStatus changeEvent(in mzabi::Schd schdOR);
```

Parameters:

schdOR An object reference to the **mzabi::Schd** object describing the scheduled event.

Returns:

An **mzabi::schdStatus** value that identifies the current status of the **mzabi::Schd** object.

Raises:

mzabix::badEvent

getStatus

Returns the status of the **exporter** object.

```
statusInfo getStatus();
```

Returns:

An **mzabi::schdStatus** value that identifies the current status of the **mzabix::exporter** object.

Raises:

mzabix::badEvent

getEventStatus

Returns the status of the specified scheduled event the **exporter** object is exporting.

```
eventStatusInfo getEventStatus(in mzabi::Schd schdOR);
```

Parameters:

schdOR	An object reference to the **mzabi::Schd** object describing the scheduled event.

Returns:

An **mzabix::eventStatusInfo** value that identifies the current status of the specified scheduled event the **exporter** object is exporting.

Raises:

mzabix::badEvent

getAllEventStatus

Returns the status of all scheduled events the **exporter** object is exporting.

```
eventStatusInfoLst getAllEventStatus();
```

Returns:

An **mzabix::eventStatusInfoLst** sequence of **mzabix::eventStatusInfo** structures. If no objects are found, an empty list is returned and no exception is raised.

Raises:

mzabix::badEvent

mzz — Session Interfaces

The session interface provides methods for managing sessions. A session is a collection of server resources allocated by or on behalf of a particular client device. A client device may have only one session active at a time on a server (the client device's unique ID is the session's primary key). A session may contain any number of resources. These resources can be virtual circuits or generic resources (literally the CORBA type "any" with an associated naming string)**.**

The session interface is the main entry point for clients using the session and virtual circuit services. The session interface provides a number of methods for circuit management. Whenever possible, client applications should use the **ses** methods for circuit management in place of those provided by the **mzc::ckt** interface.

The session interface consists of **mzz::factory** and **mzz::ses**.

mzz::factory

The **session** factory object can be used to create a session.

Methods

The **factory** interface provides the following methods:

Method	Description	Page
AllocateSession	Establishes a session for a client device with a single circuit.	A-136
AllocateSessionEx	Establishes a session and allocates multiple circuits for the session in a single call.	A-137

AllocateSession

Establishes a session for a client device and sets up the client device's first (and possibly only) circuit. This call can be made by the client itself or by a server proxy on behalf of the client. The circuit specification passed in as the *req* parameter must be a control circuit. If there is a problem (for example, if the first circuit cannot be allocated), an exception is raised.

```
session AllocateSession(in sessProperty props,
                        in mzc::clientDeviceId clientId,
                        in mzc::cktspec req);
```

Parameters:

props The **mzz::sessProperty** containing the properties of the session to create.

clientId The **mzc::clientDeviceId** that uniquely identifies the client.

req The **mzc::cktspec** specifying the attributes of the control circuit.

Returns:

An **mzz::session** structure that contains the object reference for the session and information about the session.

Raises:

mzc::cktEx, **mzz::sesEx**

See Also:

Allocate a Session and Build a Control Circuit on page 2-7

AllocateSessionEx

Identical to **AllocateSession()** but allocates multiple circuits for the session in a single call. The first circuit must be a control circuit. If any of the requested circuits are unable to be allocated, the entire request fails and an exception is raised.

```
session AllocateSessionEx(in sessProperty props,
                          in mzc::clientDeviceId clientId,
                          in mzc::cktspecs req);
```

Parameters:

props	The **mzz::sessProperty** containing the properties of the session to create.
clientId	The **mzc::clientDeviceId** that uniquely identifies the client.
req	The **mzc::cktspecs** that lists the attributes of the requested circuits.

Returns:

An **mzz::session** structure that contains the object reference for the session and information about the session.

Raises:

mzc::cktEx, **mzz::sesEx**

mzz::ses

ses provides the interface to session objects. It includes **Client Device Management Methods**, **Circuit Management Methods**, **Session Management Methods**, and **Resource Management Methods**.

ses Data Types

The **ses** interface uses the following data types:

Data Type	Description
mzc::cktspec	Discriminated union that specifies a circuit.
mzc::cktspecs	Sequence of **mzc::cktspec** unions.
mzc::circuit	Structure providing access to the circuit object and the associated **mzc::cktInfo** structure.
mzc::circuits	Sequence of **mzc::circuit** structures.
mzz::clientDevice	Structure representing a device that an end user would use.
mzz::resource	Structure representing session resources that are not circuits.
mzz::resources	Sequence of **mzz::resource** structures.
mzz::sess	Sequence of **mzz::ses** objects.
mzz::sesInfo	Session information.

Methods

The **ses** interface provides the following methods:

Method	Description	Page
Session Management Methods		
GetInfo	Returns the attributes of the session.	A-149
Release	Gracefully shuts down a session.	A-140
Client Device Management Methods		
GetClientDevice	Returns the client device associated with the session.	A-140
Circuit Management Methods		
GetCircuits	Returns a sequence containing **mzc::circuit** structures for all circuits used by a session.	A-141
AddCircuit	Allocates a circuit.	A-141
AddCircuits	Allocates multiple circuits in a single call.	A-142
DelCircuit	Disassociates the specified circuit from the session and performs a **mzc::ckt::TearDown()** on the circuit.	A-143
Resource Management Methods		
GetResource	Returns the specified resource.	A-144
GetResources	Returns all non-circuit resources associated with a session.	A-144
AddResource	Associates the specified resource with the session.	A-145
DelResource	Removes the specified resource from the session.	A-145

Session Management Methods

GetInfo

Returns the attributes of the session.

```
sesInfo GetInfo();
```

Returns:

An **mzz::sesInfo** structure that contains the attributes of the session.

Release

Gracefully shuts down a session. This operation shuts down all circuits used exclusively by the session, frees all resources associated with the session, and destroys the session.

```
void Release();
```

See Also:

mzz::factory::AllocateSession()

Delete the Session on page 2-25

Client Device Management Methods

GetClientDevice

Returns the client device associated with the session.

```
clientDevice GetClientDevice();
```

Returns:

An **mzz::clientDevice** structure that identifies the client associated with the session.

Circuit Management Methods

GetCircuits

Returns a sequence containing **mzc::circuit** structures for all circuits used by a session. An **mzc::circuit** structure consists of the object reference for the **mzc::ckt** object and the associated **mzc::cktInfo** structure. If there are no circuits, a zero-length sequence is returned.

```
mzc::circuits GetCircuits();
```

Returns:

An **mzc::circuits** sequence of **mzc::circuit** structures for all circuits used by the session.

AddCircuit

Calls the (system-internal) circuit manager to allocate a circuit that satisfies the given circuit specification and associates the circuit with the session. Returns the object reference of the created circuit. If the call to the circuit manager fails, an exception is raised and nothing happens.

```
mzc::circuit AddCircuit(in mzc::cktspec ckt);
```

Parameters:

ckt The **mzc::cktspec** union specifying the desired attributes for the circuit.

Returns:

An **mzc::circuit** structure that represents the newly created circuit.

Raises:

mzc::cktEx

See Also:

AddCircuits(), **mzc::BuildCircuit()**

Create Additional Circuits for the Session on page 2-8

AddCircuits

Identical to **AddCircuit()** but allocates multiple circuits in a single call. If any of the requested circuits cannot be created, the entire operation fails and an exception is raised.

```
mzc::circuits AddCircuits(in mzc::cktspecs ckts);
```

Parameters:

ckts	An **mzc::cktspecs** sequence of **mzc::cktspec** structures specifying the properties for the circuits.

Returns:

An **mzc::circuits** sequence of **mzc::circuit** structures for all the newly created circuits.

Raises:

mzc::cktEx

See Also:

AddCircuit()

DelCircuit

Disassociates the specified circuit from the session and gracefully shuts down the circuit (via **mzc::ckt::TearDown()**). If the circuit is not associated with the session, an exception is raised and nothing happens.

```
void DelCircuit(in mzc::ckt ckt);
```

Parameters:

ckt An object reference for the **mzc::ckt** to delete.

Returns:

Nothing

Raises:

mzc::cktEx, **mzz::sesEx**

See Also:

Create Additional Circuits for the Session on page 2-8

Resource Management Methods

GetResource

Returns the resource identified by the specified key (a string). If no resource with the specified key exists, an exception is raised.

```
resource GetResource(in string key);
```

Parameters:

key Identification key for the resource.

Returns:

An **mzz::resource** structure that matches the specified *key*.

Raises:

mzz::sesEx

GetResources

Returns a sequence containing the non-circuit resources associated with a session. If the session has no such resources associated with it, a zero-length sequence is returned.

```
resources GetResources();
```

Returns:

An **mzz::resources** sequence of **mzz::resource** structures for all the non-circuit resources associated with the session.

Raises:

mzz::sesEx

AddResource

Associates the specified resource with the session. If a resource already exists under the key given in the **mzz::resource** structure, the old resource is replaced by the new one and a copy of the old resource is returned.

```
resource AddResource(in resource value);
```

Parameters:

value The **mzz::resource** structure used to identify the new resource.

Returns:

An **mzz::resource** structure for the newly-added resource, or a previously existing resource, if one existed.

Raises:

mzz::sesEx

See Also:

AddCircuit()

DelResource

Removes the specified resource from the session. If no resource with the key exists, an exception is raised.

```
void DelResource(in string key);
```

Parameters:

key The key to the **mzz::resource** structure that identifies the resource to be deleted.

Returns:

Nothing

Raises:

mzz::sesEx

See Also:

AddResource(), **GetResources()**

mzc — Circuit Interfaces

The Virtual Circuit Manager (VCM) provides a set of interfaces for managing virtual circuits and their related communications channels.

Circuits represent logical communication paths between the server and the client device. Channels (**mzc::chnl** objects) associated with the circuit represent the physical communications paths between the server and client device. The physical arrangement of channels that form a circuit is referred to as a circuit's **geometry**. Circuits have logical attributes that specify what type of communication may occur over the circuit. You can access these attributes with a **GetInfo()** call and change them with a **Rebuild()** call.

The different circuit types depend on the traffic that may be passed over them. The different circuit types are:

- **control circuit**—May be used for Media Net traffic. A control circuit must be associated with a valid, unique Media Net address.
- **data circuit**—May be used for non-isochronous data, such as http traffic.
- **isochronous circuit**—May be used for isochronous data, such as streamed video and audio. An isochronous circuit may be associated with an **mzs::stream object** that represents the audio / video stream served to the client device.

The properties of the different types of circuit are not mutually exclusive; a circuit may be capable of transporting any and all of these data types.

The circuit interface consists of **mzc::factory** and **mzc::ckt**.

mzc::factory

The **circuit** factory object can be used to allocate a circuit. Clients usually use the **mzz::ses::AddCircuit()** method instead.

Methods

The **factory** interface provides a single method, **BuildCircuit()**.

BuildCircuit

Used by clients (or their proxies) to allocate a circuit using the information provided in the specified **mzc::cktspec** object.

```
circuit BuildCircuit(in clientDeviceId clientId,
                     in cktspec req);
```

Parameters:

clientID The **mzc::clientDeviceId** of the session with which the circuit is to be associated.

req The **mzc::cktspec** union with the properties of the circuit.

Returns:

An **mzc::circuit** structure that describes the newly allocated circuit.

Raises:

mzc::cktEx

See Also:

Create Additional Circuits for the Session on page 2-8

mzc::ckt

The **ckt** interface provides interfaces for working with circuits. In most cases, the **Circuit Management Methods** in the **mzz::ses** interface are actually preferable to the methods in **ckt.**

ckt Data Types

The **ckt** interface uses the following data types:

Data Type	Description
mzc::cktInfo	Circuit information.
mzc::cktspec	Discriminated union that specifies a circuit.

Methods

The **ckt** interface provides the following methods:

Method	Description	Page
Circuit Management Methods		
GetInfo	Returns information about a circuit.	A-149
TearDown	Shuts down a circuit, along with any channels associated with the circuit.	A-149
Rebuild	Changes the properties of a circuit and / or its associated channels.	A-149
Upstream/Downstream Channel Management Methods		
BindXSM	Allocates a new upstream or downstream channel and associates it with the circuit.	A-150
UnBindXSM	Disassociates the channel from the circuit.	A-151
EnableXSM	Marks the given channel as valid.	A-151
DisableXSM	Marks the given channel as invalid.	A-151
Stream Management Methods		
BindStream	Associates the stream with the circuit.	A-152
UnBindStream	Disassociates the stream from the circuit.	A-152

Circuit Management Methods

Circuit management methods can be used by both the client and server for getting the state of the circuit, cleaning up a circuit, and changing properties of a circuit on the fly.

GetInfo

Returns an **mzc::cktInfo** structure that describes the circuit's properties and channel geometry.

```
cktInfo GetInfo();
```

Returns:

An **mzc::cktInfo** structure that describes the circuit.

TearDown

Shuts down a circuit, along with any channels associated with the circuit.

```
void TearDown();
```

Rebuild

Changes the properties of a circuit and / or its associated channels on the fly. This enables the client to efficiently change properties instead of having to delete the circuit and then create a new circuit with the desired properties.

```
cktInfo Rebuild(in cktspec req);
```

Parameters:

req	The **mzc::cktspec** containing information about the desired circuit.

Returns:

An **mzc::cktInfo** structure to hold the attributes of the circuit after the operation.

Raises:

mzc::cktEx

Channel Management Methods

Channel management methods are intended more for the use of server-side proxies for a client than for the client itself. They provide an interface for operations like changing upstreams and / or downstreams on the fly without affecting the external view of the circuit much (if at all).

Note For the next few methods, the *X* used in the name can mean "U" for upstream or "D" for downstream. For example, **Bind*X*SM()** refers to both **BindUSM()** and **BindDSM()**.

Bind*X*SM

Allocates a new channel based on the given **mzc::chnlspec** and associates it with the circuit. It is up to the caller to clean up any previous channel. If there was already a channel associated with the requested slot—upstream or downstream—in the circuit, the channel is returned to the caller.

If no channel was associated with the requested slot in the circuit, a NULL object reference is returned. If the requested channel cannot be created or is inconsistent with the circuit properties, an exception is raised.

The **Bind*X*SM()** operations can be accomplished with a **Rebuild()** if the **mzc::chnlreq** is specified appropriately. Call the **Bind*X*SM()** method if only the channel information needs to be changed. If other changes are required in addition to the channel information, call the **Rebuild()** method.

```
channel BindDSM(in chnlspec dsm);
channel BindUSM(in chnlspec usm);
```

Parameters:

dsm/usm	The **mzc::chnlspec** specifying the downstream / upstream channel.

Returns:

An **mzc::channel** structure that contains the object reference for the new channel and information about the channel.

Raises:

mzc::cktEx

See Also:

Rebuild(), **UnBindXSM()**

UnBind*X*SM

Disassociates the given channel from the circuit. It is up to the caller to clean up the channel. This method is usually used by the server.

```
void UnBindDSM();
void UnBindUSM();
```

See Also:

Rebuild(), **BindXSM()**

Enable*X*SM

Marks the given channel as valid. This method is usually used by the server.

```
void EnableDSM();
void EnableUSM();
```

Disable*X*SM

Marks the given channel as invalid. This method is usually used by the server.

```
void DisableDSM();
void DisableUSM();
```

Stream Management Methods

Stream management methods are usually used by the server or directly by the stream service, but not by the client.

BindStream

Associates the stream with the circuit. If a stream is already bound to the circuit, or the specified stream is invalid, an exception is raised and no operation is performed.

To minimize the cross-module dependencies, **BindStream()** and **UnBindStream()** accept and return type **Object** instead of **mzs::stream**. However, these routines are used to (un)set the *streamRef* field of the **mzc::cktInfo** structure associated with the circuit. **BindStream()** only accepts **mzs::stream** objects.

```
void BindStream(in Object stm);
```

Parameters:

stm	The **mzs::stream** object to associate with the circuit, specified as an object reference.

Returns:

Nothing

Raises:

mzc::cktEx

UnBindStream

Disassociates the stream from the circuit and returns the object reference of the stream that was bound to the circuit. This method does not deallocate the stream.

```
Object UnBindStream();
```

Returns:

The **mzs::stream** object bound to the circuit. If no stream is bound to the circuit, an exception is raised.

Raises:

mzc::cktEx

mzc — Channel Interface

The channel interface describes the methods for managing channels.

mzc::chnl

A **channel** is a generic description of a communications link used to abstract the common properties of upstreams and downstreams. The properties of channels include maximum supported bit rate, maximum available bit rate, whether the channel is enabled or disabled, whether the channel supports transient operation (can disconnect / reconnect), and (optionally) information about the underlying implementation (such as the network software API the associated channel provider would use to implement it, and so on).

Note The **mzc::chnl** methods are called by the **mzc::ckt** interface, and are not usually called directly by the client.

chnl Data Types

The **chnl** interface uses the following data types:

Data Type	Description
mzc::chnlreq	Properties of a new channel.
mzc::chnlreqx	Properties of an **mzabin::Chnl** object.
mzc::chnlInfo	Attribute information of an **mzc::chnl** object.

Methods

The **chnl** interface provides the following methods:

Method	Description	Page
GetInfo	Returns information about the channel.	A-154
Enable	Marks the given **mzc::chnl** object as valid.	A-154
Disable	Marks the given **mzc::chnl** object as invalid.	A-154
Rebuild	Changes the properties of an **mzc::chnl** object.	A-155
TearDown	Disassociates the channel from its current circuit and destroys it.	A-155

GetInfo

Returns an **mzc::chnlInfo** structure that describes the channel's properties.

```
chnlInfo GetInfo();
```

Returns:

An **mzc::chnlInfo** structure that describes the channel.

Enable

Marks the given **mzc::chnl** object as valid. This method is usually used by the server.

```
void Enable();
```

Disable

Marks the given **mzc::chnl** object as invalid. This method is usually used by the server.

```
void Disable();
```

Rebuild

Changes the properties of an **mzc::chnl** object.

```
chnlInfo Rebuild(in chnlreqx req);
```

Parameters:

req The **mzc::chnlreqx** structure that specifies the new properties for the **mzc::chnl** object.

Returns:

An **mzc::chnlInfo** structure that holds the attributes of the **mzc::chnl** object after the operation.

Raises:

mzc::chnlEx

TearDown

Disassociates the channel from its current circuit and destroys it.

```
void TearDown();
```

mzs — Stream Interfaces

Clients communicate with the stream service through the **mzs::stream** interface defined in the **mzs.idl** file. Streams are allocated using the **mzs::factory** interface.

mzs::factory

The **stream** factory generates stream objects that implement the **mzs::stream** methods.

Creating a stream with the factory interface is not a "login" call. The system assumes that all operations pertaining to login, such as authentication, identification, and so on, have already been performed.

The methods that allocate resources on the server need only be called once, since many streams can be played without deallocating and reallocating. If your client device supports multiple simultaneous streams, call these methods once per simultaneous stream. The sum of the bit rates for the streams must not exceed the maximum bit rate supported by the channel to the client device. The client application is responsible for supplying a bit rate that the network can support. When the **stream** object is no longer required, use **mzs::stream::dealloc()** to free up the resource.

factory Data Types

The **factory** interface uses the following data types:

Data Type	Description
mzc::circuit	Structure providing access to the circuit object and the associated **mzc::cktInfo** structure.
mzs::bootMask	Bitmask used when booting.
mzs::capMask	Bitmask to define client capabilities.
mkd::pos	Position type that indicates where the client is positioned in the stream or segment.

Methods

The **factory** interface provides the following methods:

Method	Description	Page
alloc	Allocates an **mzs::stream** object.	A-157
boot	Initiates a boot with the stream service.	A-158

alloc

Allocates an **mzs::stream** object to be used in calls to the **mzs::stream** interface. When allocating a stream, the client passes an **mzs::capMask** to the server that indicates the type of operations the stream can perform on behalf of the client. In networks where bandwidth is allocated rather than hardcoded, the method may also cause the server to ask the network for bandwidth.

```
stream alloc(in mzc::circuit clientCircuit,
             in capMask capabilities,
             in unsigned long maxBitrate);
```

Parameters:

clientCircuit The **mzc::circuit** structure that contains an object reference to the virtual circuit used by the client. This is obtained from some external source, such as the session service.

capabilities The capability mask describing the abilities of the client device (see **mzs::capMask** for details).

maxBitrate The maximum bit rate at which the stream service is to deliver the stream to the client device.

Returns:

An object reference to the **mzs::stream** object for the allocated stream.

Raises:

mzs::client, **mzs::server**

See Also:

mzs::stream::dealloc()

Allocate a Stream on page 2-14

Note This method is not currently supported.

Initiates a boot with the stream service, using either the Oracle Boot Protocol or the DSM-CC (Digital Storage Media Command and Control) download protocol.

This method reduces round trips from the server by encapsulating the functionality of **mzs::factory::alloc()** with an initial call to **mzs::stream::bootMore()**. In this way, the client begins a boot and receives a stream context that holds the state required to complete the boot.

The client can either continue the boot to completion and then manually free the context using **mzs::stream::dealloc()**, or pass the **bootAutoClose** flag to each **boot()** and **bootMore()** method to direct the stream service to automatically deallocate the context when the boot either finishes or fails.

Set-top clients cannot call the **boot()** method since they do not run Media Net locally and cannot make an ORB call to boot themselves. Instead, set-top clients communicate with an upstream manager with some sort of low-level signaling, such as DSM-CC U-N download messages, and the upstream manager calls the stream service to boot the set-top device.

Be sure to free the stream allocated by **boot()** after the boot has completed, as it will contain information inappropriate for playing regular video.

```
stream boot(in mzc::circuit clientCircuit,
            in mkd::pos startPos,
            in mkd::pos endPos,
            in bootMask flags,
            in unsigned short req_blocksz
            out bootRespInfo RespInfoP);
```

Parameters:

clientCircuit	The **mzc::circuit** structure that contains an object reference to the virtual circuit used by the client. The reference is obtained from some external source, such as the session service.
startPos	An **mkd::pos** value indicating the start position to be used for the first boot request.
endPos	An **mkd::pos** value indicating the end position to be used for the first boot request.

flags Flags modifying the boot request (see **mzs::bootMask** for details).

req_blocksz The block size the client would like to use. The stream service denies the request if it cannot be granted.

RespInfoP The server returns an **mzs::bootRespInfo** structure to the upstream server.

Returns:

An **mzs::stream** object for use by the **mzs::stream::bootMore()** method. This stream is only suitable for downloading boot images and must not be used for calls to other stream methods, such as **mzs::stream::prepare()** and **mzs::stream::play()**.

Raises:

mzs::client, **mzs::server**, **mzs::denial**

See Also:

mzs::stream::bootCancel(), **mzs::stream::bootMore()**

mzs::stream

The **stream** interface is used by a client device or by an application server to initiate and control streams originating from the server. In order to use a stream, a client must invoke the **stream::factory::alloc()** method to acquire an object reference to the stream interface.

The stream interface does not provide a name service for streams, although it permits for the existence of such a service. This interface expects the client to provide an **mkd::assetCookie** that identifies the server content to be received in the stream. The client obtains the **mkd::assetCookie** through a content management mechanism, such as **vscontsrv**. If the given **mkd::assetCookie** is not valid for the requested content, an error is returned.

The **stream** interface does not read data from the server, and does not create connections to the server. These services are performed by a circuit (**mzc::ckt**), which is passed to the stream in the **stream::factory::alloc()** method.

stream Data Types

The **stream** interface uses the following data types:

Data Type	Description
mkd::assetCookie	Identifies media assets on the server.
mzs::bootMask	Bitmask used when booting.
mzs::finishFlags	Completed stream prepare values.
mzs::instance	Prepared stream descriptor.
mzs::internals	Stream-specific statistics.
mkd::pos	Position type that indicates where the client is positioned in the stream or segment.
mkd::prohib	Bitmask detailing prohibited rate control operations.
mzs::playFlags	Flags that define how a stream is to be played.
mkd::segInfo	Client information about a segment or stream.
mkd::segInfoList	Sequence of **mkd::segInfo** structures.

Methods

The **stream** interface provides the following methods:

Method	Description	Page
bootCancel	Cancels a boot process if something went wrong in the middle of a **bootMore()** call.	A-162
bootMore	Continues a boot that was started with the **boot()** call.	A-163
prepare	Prepares a media asset for delivery.	A-164
prepareSequence	Prepares one or more media assets for delivery.	A-165
play	Plays the stream, specifying positions, speed, and bit rate.	A-168
pause	Pauses the stream at the current position.	A-170
playFwd	Plays the stream at normal speed.	A-171
playRev	Plays the stream in reverse at normal speed.	A-172
frameFwd	Advances the stream one frame forward.	A-173
frameRev	Reverses the stream one frame.	A-174
query	Gets diagnostic information on the stream context.	A-175
getPos	Returns the current position of the playing stream.	A-176
finish	Stops playing the stream.	A-177
dealloc	Deallocates the stream.	A-178

Note This method is not currently supported.

If something goes wrong in the middle of a **bootMore()** call, the server throws exceptions to the upstream packet pump. The upstream is then responsible for catching these exceptions and calling **bootCancel()** so that cancel error information is propagated to the set-top box.

```
void bootCancel(in mkd::pos startPos,
                in mkd::pos endPos,
                in unsigned short cancel_err);
```

Parameters:

startPos An **mkd::pos** value indicating the starting position for the boot cancel.

endPos An **mkd::pos** value indicating the end position for the boot cancel.

cancel_err The **dsmcc** error the server will return to the set-top box.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**, **mzs::denial**

See Also:

mzs::factory::boot()

Note This method is not currently supported.

Continues a boot that was started with the **mzs::factory::boot()** call. The *startPos* parameter indicates the piece of the boot that is to be sent next. In an Oracle boot protocol, this is just a block number. In a DSM-CC boot, this might be something more complex, like a module and block number. The information is then sent to the client.

```
void bootMore(in mkd::pos startPos,
              in mkd::pos endPos,
              in bootMask flags);
```

Parameters:

startPos	An **mkd::pos** value indicating the starting position for the boot request.
endPos	An **mkd::pos** value indicating the end position for the boot request.
flags	Flags modifying the boot request (see **mzs::bootMask** for details).

Returns:

Nothing

Raises:

mzs::client, **mzs::server**, **mzs::denial**

prepare

Prepares a media asset for delivery. This method returns immediately from the server with information that enables the caller to estimate when the stream is ready. (The stream may have to be loaded from slow media, or the server may buffer a few seconds before starting.) The goal is to instantly display the information on the screen in some form (such as time) so the end user does not perceive server communication as slow.

This method returns a stream instance, which must be used during all other calls to the stream server. The stream instance identifies the prepared assets to the server.

When calling this method, you need to specify an **mkd::segInfoList** sequence of **mkd::segInfo** structures to be filled in with information about the prepared stream. Having this information returned to the client can help reduce latency in applications that might, for example, prepare a movie, then query it to find its length.

```
instance prepare(in mkd::assetCookie cookie,
                 in mkd::pos start,
                 in mkd::pos end,
                 in unsigned long bitrate,
                 in playFlags flags,
                 out mkd::segInfoList status,
                 in Object authRef);
```

Parameters:

cookie	The **mkd::assetCookie** used to identify the server asset the client is requesting for preparation. This cookie is obtained from some content management service, such as **vscontsrv**.
start	An **mkd::pos** value indicating the position to start playing if the **playNow** flag is passed in the *flags* parameter. Without this flag, the server ignores this parameter.
end	An **mkd::pos** value indicating the position to end playing if the **playNow** flag is passed in the *flags* parameter. Without this flag, the server ignores this parameter.
bitrate	The bit rate at which to play the content. A value of 0 means to play at whatever rate the content was encoded. If the **playNow** flag is specified in the *flags* parameter, the bit rate of 0 is used.
flags	The **mzs::playFlags** flags for controlling when and how the content is played.

status An **mkd::segInfoList** sequence to hold the returned **mkd::segInfo** structures. This tells the client what physical content was actually prepared. (There may be more than one **mkd::segInfo** structure for each passed **mkd::assetCookie**, since one **mkd::assetCookie** can resolve to multiple pieces of physical content.)

authRef An object reference to the object to authenticate the client. The source of the **mkd::assetCookie** dictates what object reference to pass here. For example, when passing **mkd::assetCookie**s from **vscontsrv**, no authorization object is required.

Returns:

An **mzs::instance** descriptor for the stream to be passed to further calls, such as **play()** and **finish()**.

Raises:

mzs::client, **mzs::server**, **mzs::denial**

See Also:

finish()

Prepare Video Content on page 2-15, *Stream Looping* on page 2-17

prepareSequence

Prepares one or more media assets for delivery. This method returns immediately from the server with information that enables the caller to estimate when the stream is ready. (The stream may have to be loaded from slow media, or the server may buffer a few seconds before starting.) The goal is to instantly display this information on the screen in some form (such as time) so the end user does not perceive server communication as slow.

This method returns a stream instance, which must be used during all other calls to the stream server. The stream instance identifies the prepared assets to the server.

Multiple assets prepared by this method are treated as one piece of logical content. They are streamed as contiguous frames with as little latency as possible between segments. When repositioning, the client is expected to give positions in the sequence as a whole, as if it were one piece of physical content. On some transports that support it, indicators are sent at segment transitions to help a client application track progress through the sequence.

When calling this method, you need to specify an **mkd::segInfoList** sequence of **mkd::segInfo** structures to be filled in with information about the prepared stream. Having this information returned to the client can help reduce latency in applications that might, for example, prepare a movie, then query it to find its length.

```
instance prepareSequence(in mkd::assetCookieList cookies,
                         out mkd::segInfoList clipStatus,
                         in mkd::pos startPos,
                         in mkd::pos endPos,
                         in unsigned long bitrate,
                         in long playRate,
                         in playFlags flags,
                         in mkd::prohib prohibitions,
                         in Object authRef);
```

Parameters:

cookies An **mkd::assetCookieList** sequence of **mkd::assetCookie**s that identifies the server assets the client is requesting for preparation. These cookies are obtained from some other service, such as **vscontsrv**.

clipStatus An **mkd::segInfoList** to hold the returned **mkd::segInfo** structures. This tells the client what physical content was actually prepared. (There may be more than one **mkd::segInfo** structure for each passed **mkd::assetCookie**, since one **mkd::assetCookie** can resolve to multiple pieces of physical content.)

startPos An **mkd::pos** value indicating the position to start playing at if the **playNow** flag is passed in the *flags* parameter. Without this flag, the server ignores the parameter.

endPos An **mkd::pos** value indicating the position at which to end playing if the **playNow** flag is passed in the *flags* parameter. Without this flag, the server ignores the parameter.

bitrate The bit rate at which the client expects to initially play the content. A value of 0 means to play at whatever rate the content was encoded. If the **playNow** flag is specified in the *flags* parameter, a bit rate of 0 is used.

playrate The presentation rate at which the client expects to initially play the content. If **playNow** is passed in the *flags* parameter, this is the presentation rate at which the content is played.

flags The **mzs::playFlags** flags for controlling when and how the content is played.

prohibitions The **mkd::prohib** bitmask listing the prohibited rate control operations. (Not currently supported.)

authRef An object reference to the object to authenticate the client. The source of the **mkd::assetCookie** dictates what object reference to pass here. For example, when passing **mkd::assetCookie**s from **vscontsrv**, no authorization object is required.

Returns:

An **mzs::instance** descriptor for the stream to be passed to further calls, such as **play()** and **finish()**.

Raises:

mzs::client, **mzs::server**, **mzs::denial**

See Also:

finish()

Prepare Video Content on page 2-15, *Stream Looping* on page 2-17

The **play()** method is the heart of the stream service. All calls to stream data use this method. It provides for play, fast forward, fast rewind, frame advance, frame rewind, and changing of the current bit rate.

You invoke the **play()** method with the parameters *curPos*, *startPos*, and *endPos*. Each parameter can be a time specified with the **mkd::pos** structure described in **mkd.idl**, or one of the special structures described in **mkdc.h**, such as **mkdBeginning**, **mkdCurrent**, **mkdEnd**, and **mkdIsDefaultStart**.

When paused, the client may take advantage of some special semantics. If a paused client passes **&mkdCurrent** for *startPos*, it requests that the server start playing as close as possible to where it was delivering data when the pause arrived. If the client is not paused and sends **&mkdCurrent** for the *startPos*, the results are undefined.

A client must not pass a NULL pointer for any of these positions.

```
void play(in instance inst,
          in mkd::pos curPos,
          in mkd::pos startPos,
          in mkd::pos endPos,
          in long playRate,
          in unsigned long bitrate);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
curPos	An **mkd::pos** value indicating the client's current position in the stream. If possible, the client should always indicate its current position when invoking the **play()** method. This parameter may be passed to an application service for information gathering, or used to enforce rate control limitations with regard to seeking. A client that can gather this information and always send an accurate *curPos* will have more capabilities than one that cannot.
startPos	An **mkd::pos** value indicating the position at which to begin playing. Meaningless when paused.
endPos	An **mkd::pos** value indicating the position at which to stop playing. Meaningless when paused.

playRate The desired speed and direction for the stream. A value of 1000 means play forward at normal speed, -1000 means play reverse at normal speed, 1 means frame advance forward (not currently supported), -1 means frame advance backward (not currently supported), and 0 means pause.

Play rate is described in *Rate and Direction Control* on page 2-20.

bitrate The bit rate at which to deliver the content. The server tries to play the requested stream at a bit rate less than or equal to the requested rate. The server never delivers content at a bit rate higher than the specified bit rate.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Prepare Video Content on page 2-15, *Play Video Content* on page 2-17

Pauses the stream at the current position. You can also pause the stream using the **play()** method, as described in *Pausing Playback* on page 2-18.

```
void pause(in instance inst,
           in mkd::pos curPos);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
curPos	An **mkd::pos** value indicating the current position of the client in the stream. If possible, the client should always indicate its current position when invoking the **pause()** method.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Play Positions: startPos, endPos, and curPos on page 2-18

playFwd

Plays the stream at normal speed. This method can be used to resume a paused stream, or to begin playing for the first time.

```
void playFwd(in instance inst,
             in mkd::pos startPos,
             in mkd::pos endPos);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
startPos	An **mkd::pos** value indicating the position at which to begin playing.
endPos	An **mkd::pos** value indicating the position at which to stop playing.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Play Positions: startPos, endPos, and curPos on page 2-18

Note This method is not currently supported. Its semantics will be clarified when it is implemented.

Plays the stream in reverse at normal speed. This can be used to "unpause" a paused stream, or to play initially if the logical start and end positions are passed in. The **playRev()** method requires single stream rate control, as described in *Rate and Direction Control* on page 2-20.

```
void playRev(in instance inst,
             in mkd::pos startPos,
             in mkd::pos endPos);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
startPos	An **mkd::pos** value indicating the position at which to begin playing.
endPos	An **mkd::pos** value indicating the position at which to stop playing.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Play Positions: startPos, endPos, and curPos on page 2-18

Note This method is not currently supported. Its semantics will be clarified when it is implemented.

Advances the stream one frame forward.

```
void frameFwd(in instance inst,
              in mkd::pos curPos);
```

Parameters:

inst An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.

curPos An **mkd::pos** value indicating the current position of the client in the stream. If possible, the client should always indicate its current position when invoking the **frameFwd()** method.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Play Positions: startPos, endPos, and curPos on page 2-18

Note This method is not currently supported. Its semantics will be clarified when it is implemented.

Reverses the stream one frame.

```
void frameRev(in instance inst,
              in mkd::pos curPos);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
curPos	An **mkd::pos** value indicating the current position of the client in the stream. If possible, the client should always indicate its current position when invoking the **frameRev()** method.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Play Positions: startPos, endPos, and curPos on page 2-18

query

This method can be called by monitoring utilities to get diagnostic information on a per-context basis. The context must be obtained with some other mechanism. A client can also call this method on its own context.

```
void query(out internals sessionState);
```

Parameters:

sessionState An **mzs::internals** structure to hold diagnostic information.

Returns:

Nothing

Raises:

Nothing

Returns the current position of the playing stream. Due to network latencies, this may not be the precise position of the stream. See *Play Positions: startPos, endPos, and curPos* on page 2-18 for more information.

```
mkd::pos getPos(in instance inst,
                out state status,
                out long playRate);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
status	Currently unsupported.
playRate	Holds the current presentation rate of the stream.

Returns:

An **mkd::pos** value indicating the current position of the stream. If the client is playing a sequence of more than one segment, the time stored in the **mkd::pos** value is for the entire sequence, rather than for any of the individual segments.

Raises:

mzs::client, **mzs::server**

See Also:

pause(), **playFwd()**, **playRev()**, **frameFwd()**, **frameRev()**

Get the Current Position in a Stream on page 2-20

Indicates that the client is no longer interested in the stream specified by *inst*. If the client is currently playing the stream, it is stopped immediately. The client can then request that the stream service prepare additional pieces of content. This method "un-prepares" the content. You cannot replay content that has been finished by this method without preparing it again.

This method can also be used to cancel an existing loop. If a stream was prepared with the **playLoop** flag, and **finish()** is called with the **finishLoop** flag, the loop is cancelled. If the stream is currently looping, it will stop at the end of the current iteration.

```
void finish(in instance inst,
            in finishFlags flags);
```

Parameters:

inst	An **mzs::instance** descriptor for the stream returned from either **prepare()** or **prepareSequence()**.
flags	Flags modifying the request. Possible flags are described by **mzs::finishFlags**. The default value is **finishAll**.

Returns:

Nothing

Raises:

mzs::client, **mzs::server**

See Also:

prepare(), **prepareSequence()**

Finish With Video Content on page 2-24

dealloc

Deallocates the stream obtained with **mzs::factory::alloc()** or **mzs::factory::boot()**, indicating that it is no longer needed. Once this method is invoked, the stream object is no longer valid and cannot be passed back to the stream service.

```
void dealloc();
```

See Also:

mzs::factory::alloc(), **mzs::factory::boot()**

Deallocate the Stream on page 2-24

mzs::ec

The **ec** interface is used by an **event supplier** implementation to push events generated by the stream service to an **event consumer**. This interface is described in the **mzsec.idl** file.

ec Data Types

The **ec** interface uses the following data type:

Data Type	Description
mzs::event	Union that describes a generic event for the event channel.

Methods

The **ec** interface provides a single method, **sendEvent()**.

sendEvent

The **sendEvent()** method is described by the **ec** interface for use by implementations of stream event suppliers to send events to consumers of stream events. The **sendEvent()** method is implemented by an event consumer. Since OVS does not provide any event consumer implementations, you must create your own as described in *Implement an Event Consumer* on page 6-2.

```
void sendEvent(in event tev);
```

Parameters:

tev	An **mzs::event** value indicating the stream service event received from the event channel.

Returns:

Nothing

See Also:

Implement the sendEvent() Operation on page 6-9

mzscl — Stream Client Functions

The stream client interface enables clients to be notified of server events. In almost all cases, clients will use the C code in the **mzscl.h** file for this purpose.

Note If your client application is not written in C, or if for some other reason you find the wrappers in **mzscl.h** insufficient for your needs, contact Oracle to discuss using the **mzscli.idl** interface directly.

The client callback function is described by the type **mzs_stream_cliCallbackHdlr**. See ***mzs_stream_cliCallbackHdlr*** on page B-77 for an example callback function.

mzscl

The **mzscl** interface provides the following functions:

Function	Description	Page
cliInit	Initializes the callback environment.	A-180
cliTerm	Terminates the callback environment.	A-180
setCallback	Registers the callback function.	A-181
removeCallback	Terminates the callback function.	A-182

cliInit

Initializes the callback environment. Call this function once before registering a callback function.

```
void mzscl_stream_cliInit(void);
```

cliTerm

Terminates the callback environment. Call this function when you no longer require the callback functionality and do not expect to call **setCallback()** or **removeCallback()** any more.

```
void mzscl_stream_cliTerm(void);
```

setCallback

Registers the callback function to inform the client when interesting events occur in the stream service. The events that may trigger a callback are specified in **mzscli.idl**. When one of these events is sent from the server to the client, the function specified by the *fup* parameter is called.

```
void mzscl_stream_setCallback(mzs_stream orS,
                              yoenv *evCli,
                              mzs_stream_cliCallbackHdlr fup,
                              dvoid *argv);
```

Parameters:

orS	A stream object obtained from the **mzs::stream::alloc()** call.
evCli	A **yoenv** the client wishes to be passed to the server for the remove callback operation.
fup	An **mzs_stream_cliCallbackHdlr** that describes the callback function to be called when an **mzscli.idl** event occurs. This call will occur when Media Net is idle.
argv	A user-specified context pointer. This pointer is passed as a parameter to the callback function when an event occurs.

Returns:

Nothing

removeCallback

"Unsets" the callback function previously registered by **setCallback()**. Call this function when you no longer wish to be informed when server events occur.

```
void mzscl_stream_removeCallback(mzs_stream orS,
                                 yoenv *evCli);
```

Parameters:

orS	A stream object obtained from the **mzs::stream::alloc()** call.
evCli	A **yoenv** the client wishes to be passed to the server for the remove callback operation.

Returns:

Nothing

mzs_clientCB

This is the interface that must be implemented on the client to receive callback events. Clients using the wrapper routines specified in **mzscl.h** and linking in the appropriate client side code get this interface implemented for them.

Functions

This interface provides a single function, **endOfStream()**.

endOfStream

Receives notification that a playing stream has stopped. When the server stops delivering a stream for some reason, it can call the callback interface to indicate that the play stopped, and to inform the client why.

```
void endOfStream(in mzs_notify reason);
```

Parameters:

reason The reason the server stopped the stream.

Returns:

Nothing

Raises:

Nothing

vesw — Real-time Feed Functions

The real-time feed service enables encoders to load content onto the Oracle Video Server in real time. Encoder developers should use the functions in the **vesw.h** file. The **vesw** functions are wrapper functions around the **ves** interfaces defined in the **vesm.idl** file. This wrapper ensures future compatibility should the server interfaces change.

The **vesw** interface supports only synchronous functions.

vesw Data Types

The **vesw** interface uses the following **ves** data types:

Data Type	Description	Page
ves::format	File format of the content.	B-82
ves::audCmp	Audio compression format.	B-82
ves::vidCmp	Video compression format.	B-82
ves::time	Structure specifying time.	B-83
ves::timeType	Enumerated type specifying time.	B-84
ves::SMPTE	Time in hh:mm:ss:frame format.	B-85
ves::frame	Compressed frame type of a tagged frame.	B-85
ves::hdr	Format of the video data for a real-time feed.	B-87
ves::m1sHdr	MPEG-1 sequence header containing stream encoding information.	B-88
ves::m2tHdr	Encoding information about an MPEG-2 stream.	B-89
ves::rkfHdr	Data that the decoder requires before decompressing.	B-90
ves::tag	Frame type, size, current offset, and format-specific information of a tagged frame.	B-90
ves::tagList	Sequence of **ves::tag** structures.	B-91
ves::m1sTag	MPEG-1 specific information about the tagged frame.	B-91
ves::m2tTag	MPEG-2 specific information about the tagged frame.	B-92
ves::rkfTag	RKF-specific information about the tagged frame.	B-93
ves::vendor	Encoding vendor.	B-93

Functions

The **vesw** interface provides the following synchronous functions:

Function	Description	Page
veswInit	Initializes the Media Net object environment.	A-186
veswIdle	Maintains the Media Net object environment after **veswInit()** has been called.	A-187
veswNewFeed	Creates a new feed session with the video server and returns a real-time feed context.	A-188
veswPrepFeed	Creates a new feed session with the video server, but does not send any **ves::hdr** information.	A-189
veswSendHdr	Sends **ves::hdr** information to the server.	A-190
veswSendData	Sends encoded content to the server.	A-191
veswSendTags	Sends information about tagged frames to the server.	A-192
veswSendBlob	Sends non-streaming data associated with the stream to the server.	A-193
veswContBlob	Continues to send non-streaming data associated with the stream to the server.	A-194
veswLastError	Returns a description of the last error that occurred.	A-194
veswClose	Shuts down an individual real-time feed gracefully and releases the related resources.	A-195
veswTerm	Shuts down the Media Net object environment and terminates the Media Net session.	A-195

Initializes the Media Net object environment. The **veswInit()** call is a wrapper around **ysInit()**, **mnInit()**, and **yoInit()**. If your program is already calling these functions, you do not need to call **veswInit()**.

It is best to call **veswInit()** during the program's main initialization phase. The **veswInit()** call must succeed before any other **vesw** calls can be made. It is possible for **veswInit()** to exit before performing all necessary initialization. If **veswInit()** fails, an error occurs and the parent process aborts.

After calling **veswInit()**, call **veswIdle()** or another **vesw** function at least once every 30 seconds to prevent the Media Net connection with the server from timing out.

```
veswLayer veswInit(ub1 *ysCtx,
                   CONST char *progNm);
```

Parameters:

ysCtx	A pointer to a block of memory of size SYSX_OSDPTR_SIZE bytes. This memory must persist until **veswTerm()** is called.
progNm	The name used to identify the encoder process.

Returns:

A **veswLayer** value that indicates if the operation completed successfully or, in the event of a failure, at what layer (**ys**, **mn**, **yo**) the failure occurred.

See Also:

Initialize Media Net on page 7-7

veswIdle

Maintains the Media Net object environment after **veswInit()** has been called.

```
void veswIdle(ub1 *ysCtx);
```

Parameters:

ysCtx	A pointer to the block of memory of size SYSX_OSDPTR_SIZE bytes that was previously allocated by **veswInit()**.

Returns:

Nothing

See Also:

Send Content and Tags to the Server on page 7-13

Creates a new feed session with the video server and returns a real-time feed context.

```
boolean veswNewFeed(veswCtx **newCtx,
                    CONST char *baseName,
                    boolean rolling,
                    ub4 bitrate,
                    CONST ves_time *duration,
                    ub4 eyes,
                    ub4 pees,
                    ub4 bees,
                    CONST ves_hdr *hdr);
```

Parameters:

newCtx	The **veswCtx** structure to contain the returned real-time feed context. This structure must be passed to all other **vesw.h** calls for the feed.
baseName	The MDS-style filename; extensions will be added by the server.
rolling	False indicates a "one-step encode" of fixed length content, such as a commercial or a movie. True indicates a continuous real-time feed, where the encode continues indefinitely, and the server is expected to maintain a rolling buffer of the last (*duration*) portion of the feed.
bitrate	The bit rate for encoded content (for example, 1536000 bps).
duration	The **ves::time** structure indicating the expected duration of content (or amount of content to buffer on the server for a continuous real-time feed).
eyes *pees* *bees*	The target number of frames of each type (I, P, B) per minute. These numbers are used to reserve the appropriate amount of space for the tag file. If in doubt, estimate high.
hdr	The **ves::hdr** structure describing the encoded content.

Returns:

False if the operation completed successfully and has nothing to report. True if the operation failed. For details, call **veswLastError()** before calling **veswClose()**.

See Also:

Open Connection to Video Server on page 7-10

Creates a new feed session with the video server and returns a real-time feed context. Similar to the **veswNewFeed()** function, only it does not send any header information upon opening a connection to the server.

The **veswPrepFeed()** function, in combination with the **veswSendHdr()** function, has the same effect as **veswNewFeed()**, but almost all of the delay is contained in the **veswPrepFeed()** call. This function can be useful if the **ves::hdr** information has to be extracted from the feed after encoding has started.

```
boolean veswPrepFeed(veswCtx **newCtx,
                     CONST char *baseName,
                     boolean rolling,
                     ub4 bitrate,
                     CONST ves_time *duration,
                     ub4 eyes,
                     ub4 pees,
                     ub4 bees,
                     ves_format fmt);
```

Parameters:

newCtx	The **veswCtx** structure to contain the returned real-time feed context. This structure must be passed to all other **vesw.h** calls for the feed.
baseName	The MDS-style filename; extensions will be added by the server.
rolling	False indicates a “one-step encode” of fixed length content, such as a commercial or a movie. True indicates a continuous real-time feed, where the encode continues indefinitely, and the server is expected to maintain a rolling buffer of the last (*duration*) portion of the feed.
bitrate	The bit rate for encoded content (for example, 1536000 bps).
duration	The **ves::time** structure indicating the expected duration of content (or amount of content to buffer on the server for a continuous real-time feed).
eyes *pees* *bees*	The target number of frames of each type (I, P, B) per minute. These numbers are used to reserve the appropriate amount of space for the tag file. If in doubt, estimate high.
fmt	The **ves::format** structure that specifies the file format of the content.

Returns:

False if the operation completed successfully and has nothing to report. True if the operation failed. For details, call **veswLastError()** before calling **veswClose()**.

veswSendHdr

If you have opened a connection to the server using the **veswPrepFeed()** function, you need to use the **veswSendHdr()** function to send the **ves::hdr** information to the server before calling **veswSendTags()**.

```
boolean veswSendHdr(veswCtx *ctx,
                    CONST ves_hdr *hdr);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.
hdr	The **ves::hdr** structure describing the encoded content.

Returns:

False if the operation completed successfully and has nothing to report. True if the operation failed. For details, call **veswLastError()** before calling **veswClose()**.

veswSendData

Sends encoded content to the server.

```
boolean veswSendData(veswCtx *ctx,
                     CONST dvoid *buf,
                     size_t count);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.
buf	The contiguous block of data to be sent to OVS.
count	The number of bytes in *buf*.

Returns:

False if no errors occurred; true if something went wrong. In that case the session is no longer active and you need to call **veswClose()**. For details, call **veswLastError()** before calling **veswClose()**.

See Also:

Send Content and Tags to the Server on page 7-13

veswSendTags

Sends information about tagged frames to the server.

```
boolean veswSendTags(veswCtx *ctx,
                     CONST ves_tag *tagBuf,
                     ub4 count);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.
tagBuf	The contiguous block of **ves::tag** structures to be sent to the server.
count	The number of **ves::tag** structures (not bytes) in *tagBuf*.

Returns:

False if no errors occurred; true if something went wrong. In that case the session is no longer active and you need to call **veswClose()**. For details, call **veswLastError()** before calling **veswClose()**.

See Also:

Send Content and Tags to the Server on page 7-13

veswSendBlob

Sends non-streaming data associated with the stream in the form of BLOBs to the server.

```
boolean veswSendBlob(veswCtx *ctx,
                     CONST char *fileName,
                     CONST char *dataType,
                     CONST char *description,
                     CONST ves_time *start,
                     CONST ves_time *end,
                     boolean more,
                     ub4 dataLen,
                     CONST dvoid *data,
                     dvoid *blobOR);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.
filename	The suggested name for storage on the server.
dataType	The type of information contained in the BLOB.
description	The detailed description of BLOB for an operator to examine.
start	The **ves::time** structure indicating the starting position to be read from in the BLOB file.
end	The **ves::time** structure indicating the last position to be read from in the BLOB file.
more	True if the BLOB is transmitted as a series of pieces, false otherwise. If true, the next call is **veswContBlob()**.
dataLen	The size of the data pointed to by *data*.
data	The first chunk of data.
blobOR	An object reference to the BLOB stream.

Returns:

True if the data was stored on the server.
False if this data type is discarded by the server or an error occurred.

veswContBlob

Continues to send non-streaming data as BLOBs to the server. This function is called after the first piece in the series has already been sent by a call to **veswSendBlob()** with *more* set to true.

```
boolean veswContBlob(veswCtx *ctx,
                     dvoid *blobOR,
                     boolean more,
                     ub4 dataLen,
                     CONST dvoid *data);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.
blobOR	An object reference to the BLOB stream. The first piece of the BLOB must already have been sent by **veswSendBlob()**.
more	True if additional pieces of the BLOB are to be sent with additional calls to **veswContBlob()**, false otherwise.
dataLen	The size of the data pointed to by *data.*
data	The first chunk of data.

Returns:

True if the data was stored on the server.
False if this data type is discarded by the server or if an error occurred.

veswLastError

Returns a description of the last error that occurred.

```
CONST char *veswLastError(veswCtx *ctx);
```

Parameters:

ctx	The **veswCtx** structure containing the real-time feed context.

Returns:

A pointer to the error string.

veswClose

Shuts down an individual real-time feed gracefully and releases the related resources. (In contrast, **veswTerm()** shuts down the Media Net object environment.)

```
boolean veswClose(veswCtx *ctx);
```

Parameters:

ctx The **veswCtx** structure containing the real-time feed context.

Returns:

False if no errors occurred; true if something went wrong. The context is gone regardless of the return value.

See Also:

Shut Down Feed on page 7-14

veswTerm

Shuts down the Media Net object environment and terminates the Media Net session. (In contrast, **veswClose()** closes an individual feed.)

```
veswLayer veswTerm(ub1 *ysCtx);
```

Parameters:

ysCtx A pointer to the block of memory of size SYSX_OSDPTR_SIZE bytes that was previously allocated by **veswInit()**.

Returns:

A **veswLayer** value that indicates if the operation completed successfully or, in the event of a failure, at what layer (**ys**, **mn**, **yo**) the failure occurred.

See Also:

Terminate Media Net on page 7-14

APPENDIX

B OVS Data Types Reference

This appendix describes the various data types, that is, structures, enumerated types, bitmasks, and type definitions used by the OVS methods and functions.

Data Type	Description	Page
MdsnmNtvMaxLen	Maximum length of a native MDS file name.	B-13
mkd Data Types		
mkd::assetCookie	Identifies video content on the server.	B-13
mkd::assetCookieList	Sequence of **mkd::assetCookie**s.	B-14
mkd::compFormat	Compression format of the content file.	B-14
mkd::contFormat	Codec types and format types for the content file.	B-15
mkd::contStatus	Indicates where a piece of content is located.	B-16
mkd::mediaType	Sequence of 4 bytes that specifies a codec, video or audio.	B-16
mkd::pos	Position type that indicates where the client is positioned in the stream or segment.	B-17
mkd::posTime	Time in a stream or segment.	B-19
mkd::wall	Describes time in a wall clock or “real world” fashion.	B-20
mkd::gmtWall	Describes time in Greenwich Mean Time (GMT) wall clock time.	B-20
mkd::localWall	Describes time in local wall clock time.	B-21
mkd::zone	Specifies a time zone.	B-21
mkd::prohib	Bitmask detailing prohibited rate control operations.	B-22
mkd::segCapMask	Bitmask specifying rate control capabilities.	B-23

Data Type	Description	Page
mkd::segInfo	Client information about a segment or stream.	B-24
mkd::segInfoList	Sequence of **mkd::segInfo** structures.	B-26
mkd::segment	A segment of a piece of content.	B-26
mkd::segmentList	Sequence of **mkd::segment** structures.	B-26
mkd::segMask	Segment mask used to specify particular options for a segment of content.	B-27
mtux Data Types		
mtuxLayer	Media Net initialization status.	B-28
mza Data Types		
mza::LgCtntAtr	Attribute information of an **mza::LgCtnt** object.	B-29
mza::LgCtntAtrLst	Sequence of **mza::LgCtntAtr** structures.	B-30
mza::ClipAtr	Attribute information of an **mza::Clip** object.	B-31
mza::ClipAtrLst	Sequence of **mza::ClipAtr** structures.	B-32
mza::CtntAtr	Attribute information of an **mza::Ctnt** object.	B-32
mza::CtntAtrLst	Sequence of **mza::CtntAtr** structures.	B-34
mza::CtntPvdrAtr	Attribute information of an **mza::CtntPvdr** object.	B-34
mza::CtntPvdrAtrLst	Sequence of **mza::CtntPvdrAtr** structures.	B-35

Data Type	Description	Page
mza::Itr	Structure for iterating through a list.	B-35
mza::ObjLst	Generic sequence of objects.	B-35
mza::opstatus	Enumerated type that indicates whether processing was complete before an exception.	B-36
mzabi Data Types		
mzabi::SchdAtr	Attribute information of an **mzabi::Schd** object.	B-37
mzabi::SchdAtrLst	Sequence of **mzabi::SchdAtr** structures.	B-38
mzabi::ExpAtr	Attribute information of an **mzabi::Exp** object.	B-38
mzabi::ExpAtrLst	Sequence of **mzabi::ExpAtr** structures.	B-39
mzabi::ExpGrpAtr	Attribute information of an **mzabi::ExpGrp** object.	B-39
mzabi::ExpGrpAtrLst	Sequence of **mzabi::ExpGrpAtr** structures.	B-39
mzabi::schdStatus	Status of an **mzabi::Schd** object.	B-40
mzabi::expStatus	Status of an **mzabi::Exp** object.	B-41
mzabi::eventType	Type of scheduled event.	B-42
mzabin Data Types		
mzabin::NvodAtr	Attribute information of an **mzabin::Nvod** object.	B-43
mzabin::NvodAtrLst	Sequence of **mzabin::NvodAtr** structures.	B-44
mzabin::NvodSchdAtr	Attribute information of an **mzabin::Nvod** object and its associated **mzabi::Schd** object.	B-44

Data Type	Description	Page
mzabin::NvodSchdAtrLst	Sequence of **mzabin::NvodSchdAtr** structures.	B-45
mzabin::ChnlAtr	Attribute information of an **mzabin::Chnl** object.	B-46
mzabin::ChnlAtrLst	Sequence of **mzabin::ChnlAtr** structures.	B-46
mzabin::nvodStatus	Status of an NVOD object.	B-47
mzabin::loopType	Type of looping desired.	B-48
mzabix Data Types		
mzabix::statusInfo	Status of an **mzabix::exporter** object.	B-48
mzabix::statusInfoLst	Sequence of **mzabix::statusInfo** structures.	B-49
mzabix::eventStatusInfo	Status of a scheduled event the **mzabix::exporter** object is exporting.	B-49
mzabix::eventStatusInfoLst	Sequence of **mzabix::eventStatusInfo** structures.	B-49
mzc Data Types		
mzc::circuit	Structure providing access to the circuit object and the associated **mzc::cktInfo** structure.	B-50
mzc::circuits	Sequence of **mzc::circuit** structures.	B-50
mzc::cktInfo	Circuit information.	B-51
mzc::cktInfos	Sequence of **mzc::cktInfo** structures.	B-51

Data Type	Description	Page
mzc::cktreq	Discriminated union providing information about the kind of circuit to use; either **mzc::cktreqAsym** or **mzc::cktreqSym**.	B-52
mzc::cktreqAsym	Specifies information about an asymmetric circuit to be allocated.	B-52
mzc::cktreqSym	Specifies information about a symmetric circuit to be allocated.	B-53
mzc::ckts	Sequence of **mzc::ckt** objects.	B-53
mzc::cktspec	Discriminated union that specifies a circuit.	B-54
mzc::cktspecs	Sequence of **mzc::cktspec** unions.	B-54
mzc::clientDeviceId	Indicates which client a session is allocated to.	B-55
mzc::channel	Channel information.	B-55
mzc::channels	Sequence of **mzc::channel** structures.	B-55
mzc::chnlInfo	Attribute information of an **mzc::chnl** object.	B-56
mzc::chnlInfos	Sequence of **mzc::chnlInfo** structures.	B-56
mzc::chnlreq	Properties of a new channel.	B-57
mzc::chnlreqx	Properties of an **mzabin::Chnl** object.	B-58
mzc::chnls	Sequence of **mzc::chnl** objects.	B-58
mzc::chnlspec	Discriminated union that specifies the type of channel to use.	B-59

Data Type	Description	Page
mzc::chnlspecs	Sequence of **mzc::chnlspec** structures.	B-59
mzc::commProperty	Bitmask detailing the properties of a channel or circuit.	B-60
mzc::logicalAddress	Media Net logical address.	B-62
mzc::link	Network connection information.	B-62
mzc::netapi	Network setup information.	B-63
mzc::netif	Hardware interface.	B-64
mzc::netproto	Addressing information.	B-65
mzc::netprotos	Sequence of **mzc::netproto** structures.	B-65
mzc::pktinfo	Structure representing packet parameters.	B-66
mzctt::transportType	Transport types recognized by channel providers.	B-67
mzs Data Types		
mzs::bootMask	Bitmask used when booting.	B-68
mzs::bootRespInfo	Number of blocks that the boot file was wrapped in.	B-68
mzs::capMask	Bitmask to define client capabilities.	B-69
mzs::event	Union that describes a generic event for the event channel.	B-71
mzs::finishFlags	Completed stream prepare values.	B-72
mzs::instance	Prepared stream descriptor.	B-72
mzs::internals	Stream-specific statistics.	B-73

Data Type	Description	Page
mzs::mzs_notify	Reason an **mzs::stream** has ended.	B-74
mzs::playFlags	Flags that define how a stream is to be played.	B-75
mzs::state	Stream state.	B-76
mzs_stream_cliCallbackHdlr	Describes a client callback function.	B-77
mzz Data Types		
mzz::sessProperty	Bitmask detailing properties of a session that is being requested.	B-78
mzz::clientDevice	Structure representing a device that an end user would use.	B-79
mzz::resource	Structure representing session resources that are not circuits.	B-79
mzz::resources	Sequence of **mzz::resource** structures.	B-80
mzz::sess	Sequence of **mzz::ses** objects.	B-80
mzz::sesInfo	Session information.	B-80
mzz::sesInfos	Sequence of **mzz::sesInfo** structures.	B-81
mzz::session	Structure providing a session object reference and a **mzz::sesInfo** structure.	B-81
mzz::sessions	Sequence of **mzz::session** structures.	B-81
ves Data Types		
ves::format	File format of the content.	B-82
ves::audCmp	Audio compression format.	B-82
ves::vidCmp	Video compression format.	B-82

Data Type	Description	Page
ves::time	Structure specifying time.	B-83
ves::timeType	Enumerated type specifying time.	B-84
ves::SMPTE	Time in hh:mm:ss:frame format.	B-85
ves::frame	Compressed frame type of a tagged frame.	B-85
ves::hdr	Format of the video data for a real-time feed.	B-87
ves::m1sHdr	MPEG-1 sequence header containing stream encoding information.	B-88
ves::m2tHdr	Encoding information about an MPEG-2 stream.	B-89
ves::rkfHdr	Data that the decoder requires before decompressing.	B-90
ves::tag	Frame type, size, current offset, and format-specific information of a tagged frame.	B-90
ves::tagList	Sequence of **ves::tag** structures.	B-91
ves::m1sTag	MPEG-1 specific information about the tagged frame.	B-91
ves::m2tTag	MPEG-2 specific information about the tagged frame.	B-92
ves::rkfTag	RKF-specific information about the tagged frame.	B-93
ves::vendor	Encoding vendor.	B-93
vesw Data Types		
veswCtx	Context for a real-time feed.	B-94
veswLayer	Specifies either success or at what layer failure occurred.	B-94

mds Data Types

MDS flags

MDS uses the following flags:

Flag	Description
MdsFlgWrt	Acquire the write-lock for the file. Only one file pointer may have write permission at any given time for a particular file.
MdsFlgDel	Include files that have been marked as deleted but are still on disk.
MdsFlgSeq	Sequential access mode. For read operations, MDS discards each buffer as soon as data is read to minimize memory usage. For write operations, MDS initiates a flush of each RAID stripe as soon as data is written.
MdsFlgUbw	Unlimited bandwidth. Client is trusted to remain within the specified bandwidth rate. Use is strongly discouraged.
MdsFlgPrv	Client is allowed to open read-only files for writing. Used by MDS clients trusted to preserve file integrity such as **mdsrebuild** and **mdsdefrag**. Use is strongly discouraged.
MdsFlgRtm	File is accessed in real-time. Not useful since you cannot obtain bandwidth tokens via the public interface.
MdsFlgPxy	File access is with the MDS remote server (**mdsrmtsrv**) even if the process has access to disks directly.
MdsFlgHsm	File may be automatically migrated in from tertiary storage. Client must be prepared to wait a potentially long time for this operation to complete.

Used by **mdsCreate()**, **mdsOpen()**, **mdsnmMatchCreate()**, **mdsnmExpand** **mdsnmExpandOne()**

fileExReason

An enumerated type that specifies the possible reasons why a file exception was raised. The file exception may be dealt with on a per-file basis.

```
enum fileExReason
{
// Attributed to underlying volume
    fileExReadOnlyVol,
    fileExTocFull,
    fileExVolFull,
    fileExMinBw,

// Attributed to file
    fileExMissing,
    fileExIllegal,
    fileExMatch,
    fileExParse,
    fileExReadOnlyFile,
    fileExDeleted,
    fileExLocked,
    fileExHostNop,
    fileExPerm,
    fileExCallBack
};
```

This enumerated type has the following values:

Value	Description
Attributed to underlying volume	
fileExReadOnlyVol	Volume is in read-only mode and cannot be modified.
fileExTocFull	Volume's TOC is full (no space for file metadata).
fileExVolFull	Volume is full (no space for file data).
fileExMinBw	Unable to allocate required bandwidth for volume.
Attributed to file	
fileExMissing	File name does not exist.
fileExIllegal	File name has illegal characters, is too long, and so on (see **mdsnmIsLegal()**).
fileExMatch	Specified wildcard expression has no matches.
fileExParse	Cannot parse file name.

Value	Description
fileExReadOnlyFile	Attempted to perform write-operation on file in read-only mode.
fileExDeleted	File exists but is deleted.
fileExLocked	File is locked by another writer.
fileExHostNop	Operation is not available on host files.
fileExPerm	No permission.
fileExCallBack	File callback time-out.

Used by **mds::fileEx**

mdsBlob

Opaque structure that defines the context for a Binary Large OBject (BLOB).

```
typedef struct mdsBlob mdsBlob;
```

Used by **mdsBlobPrepare()**, **mdsBlobPrepareSeg()**, **mdsBlobTransfer()**

mdsBw

Optional MDS bandwidth specification. Public clients cannot perform explicit bandwidth management and should use the symbol *MdsDfltBitRate* with any **mdsCreate()** or **mdsOpen()** call.

```
#define MdsDfltBitRate    ((mdsBw *)0)
```

Used by **mdsCreate()**, **mdsOpen()**

mdsFile

Opaque structure containing all the information for both MDS-native and host files. It also contains a cache that is used by all synchronous read or write requests to MDS.

Used by **mdsCreate()**

mdsMch

Opaque match context for **mdsnmMatchCreate()**, **mdsnmMatchNext()**, and **mdsnmMatchDestroy()** wildcard calls.

```
typedef struct mdsMch mdsMch;
```

Used by **mdsnmMatchCreate()**, **mdsnmMatchNext()**, **mdsnmMatchDestroy()**

mdsnm Definitions

MdsnmMaxLen

Maximum length of an MDS or host file name.

```
#define MdsnmMaxLen        (max(MdsnmNtvMaxLen, 1024))
```

MdsnmNtvMaxLen

Maximum length of a native MDS file name: /mds/*volume*/*filename*.

```
#define MdsnmNtvMaxLen
       (sizeof(MdsnmNtvFilePrefix)+MdsnmNtvFileLen + MdsnmNtvVolLen +2)
```

mkd Data Types

mkd::assetCookie

An **assetCookie** identifies video content on the server. This is a binary tag, and should not be interpreted as a file name. It is completely opaque to the client, and will generally be obtained from some sort of content manager, such as **vscontsrv**.

```
typedef string assetCookie;
```

See *Prepare Video Content* on page 2-15 for details on using **assetCookie**s.

Used by **mdsBlobPrepare()**, **mdsBlobPrepareSeg()**, **mzs::stream::prepare()**, **mzs::stream::prepareSequence()**

mkd::assetCookieList

A sequence of **assetCookie**s.

```
typedef sequence<assetCookie> assetCookieList;
```

Used by **mzs::stream::prepareSequence()**

mkd::compFormat

A bitmask that indicates the compression format of the content file.

```
typedef unsigned long compFormat;
  const compFormat compFormatError    = 0x00000000;
  const compFormat compFormatMpeg1    = 0x00000001;
  const compFormat compFormatMpeg2    = 0x00000004;
  const compFormat compFormatOrca     = 0x00000008;
  const compFormat compFormatRawKey   = 0x00000400;
  const compFormat compFormatSegWHdr  = 0x00002000;
  const compFormat compFormatSegWoHdr = 0x00004000;
  const compFormat compFormatDsmcc    = 0x00008000;
```

Mask	Value	Description
compFormatError	0x00000000	Error format.
compFormatMpeg1	0x00000001	MPEG-1 format.
compFormatMpeg2	0x00000004	MPEG-2 format.
compFormatOrca	0x00000008	No longer supported.
compFormatRawKey	0x00000400	Raw key format.
compFormatSegWHdr	0x00002000	Oracle boot file (with header).
compFormatSegWoHdr	0x00004000	Oracle boot file (without header).
compFormatDsmcc	0x00008000	DSM-CC boot.

Used by **mkd::segInfo**

mkd::contFormat

A 4-byte code used for identifying codec types and format types.

```
struct contFormat
{
    string         mkd_contFormatVendor;
    compFormat     mkd_contFormatFmt;
    mediaType      mkd_contFormatAud;
    mediaType      mkd_contFormatVid;

// Client display information
    unsigned short mkd_contFormatHeightInPixels;
    unsigned short mkd_contFormatWidthInPixels;
    long           mkd_contFormatPelAspectRatio;
    unsigned long  mkd_contFormatFrameRate;
};
```

This structure has the following fields:

Field	Type	Description
mkd_contFormat Vendor	string	Content vendor.
mkd_contFormat Fmt	**mkd::compFormat**	Format of content.
mkd_contFormat Aud	**mkd::mediaType**	Format of audio content.
mkd_contFormat Vid	**mkd::mediaType**	Format of video content.
mkd_contFormat HeightInPixels	unsigned short	Height of frame, in pixels.
mkd_contFormat WidthInPixels	unsigned short	Width of frame, in pixels.
mkd_contFormat PelAspectRatio	long	Height*10000 / width of *individual* pixels.
mkd_contFormat FrameRate	unsigned long	Frame rate (fps) *1000, for example, 29.97 fps -> 29970.

Used by **mza::CtntAtr**

mkd::contStatus

An enumerated type to indicate where a piece of content is located. Where a piece of content is located affects how long it will take before the content can be delivered to a client.

```
enum contStatus
{
    contStatusDisk,
    contStatusTape,
    contStatusFeed,
    contStatusRolling,
    contStatusTerminated,
    contStatusUnavailable
};
```

This enumerated type has the following values:

Value	Description
contStatusDisk	The content is already located on disk.
contStatusTape	The content is stored on tape.
contStatusFeed	The content is still being encoded, that is, “one-step encoding.”
contStatusRolling	The content is being encoded and may also be deleted, that is, “continuous real-time feed.”
contStatusTerminated	The content is a terminated, continuous feed.
contStatusUnavailable	Access is restricted for some reason.

Used by **mkd::segInfo**

mkd::mediaType

A sequence of 4 bytes that specifies a codec, video or audio.

```
typedef sequence<octet, 4> mediaType;
```

Used by Currently unused.

mkd::pos

The **mkd::pos** type describes a position in a stream or segment. It can be specified in any of the time formats listed.

Once a stream has been prepared and is in use, the play methods expect positions to describe the current position of the client in the stream, where a request should begin, and where it should end. These positions are commonly known as the **curPos**, **startPos**, and **endPos**. They are expressed using the position type, **mkd::pos**.

The semantics have not yet been defined for the times in beginning, current, and end. The wisest course of action is to set them to 0:0:0:0.

The **pos** type is designed to be expressed in terms of any of the types listed below. The stream interface expresses **pos** using the **mkd::posTime** structure. The boot service expresses **pos** using the **posBlock** type.

```
union pos switch (posType)
{
    case posTypeBeginning:    posTime   mkd_posBegin;
    case posTypeCurrent:      posTime   mkd_posCur;
    case posTypeEnd:          posTime   mkd_posEnd;
    case posTypeByte:         posByte   mkd_posBytePos;
    case posTypeTime:         posTime   mkd_posTimePos;
    case posTypeMillisecs:    posMs     mkd_posMsPos;
    case posTypeMpegSCR:      posSCR    mkd_posSCRPos;
    case posTypeMpegPCR:      posPCR    mkd_posPCRPos;
    case posTypeBlock:        posBlock  mkd_posBlockPos;
    case posTypeDsmcc:        posBlock  mkd_posDsmcc;
    case posTypeContFeed:     posTime   mkd_posFeed;
    case posTypeUnformed:     posTime   mkd_posNoWay;
    case posTypeDefaultStart: posTime   mkd_posDefaultStart;
};
```

This union has the following fields:

Field	Description
mkd_posBegin	Beginning of the stream.
mkd_posCur	Current position of stream.
mkd_posEnd	End of the stream.
mkd_posBytePos	Specify position of a byte. Currently unused.
mkd_posTimePos	hh:mm:ss:cc. Currently unused.

Field	Description
mkd_posMsPos	Millisecs from start. Currently unused.
mkd_posSCRPos	33-bit SCR. Currently unused.
mkd_posPCRPos	42-bit PCR. Currently unused.
mkd_posBlockPos	Block number for boot. Currently unused.
mkd_posDsmcc	DSM-CC boot. Currently unused.
mkd_posFeed	hh:mm:ss:cc. Currently unused.
mkd_posNoWay	Stream beginning. Currently unused.
mkd_posDefaultStart	Default start.

Used by **mzs::stream::prepare()**, **mzs::stream::prepareSequence()**, **mzs::stream::play()**, **mzs::stream::pause()**, **mzs::stream::playFwd()**, **mzs::stream::playRev()**, **mzs::stream::frameFwd()**, **mzs::stream::frameRev()**, **mzs::stream::getPos()**, **mzs::stream::finish()**

mkd::posTime

The **posTime** structure specifies time in a stream or segment. This time can be specified in terms of hours, minutes, seconds, and hundredths of a second (from the beginning of the stream or segment).

```
struct posTime
{
    unsigned long mkd_posTimeHour;
    octet    mkd_posTimeMinute;
    octet    mkd_posTimeSecond;
    octet    mkd_posTimeHundredth;
};
```

The begin, end, and current types are constants defined in **mkdc.h**:

```
externref CONST_DATA mkd_pos mkdBeginning;
#define mkdIsBeginning(pos)     ((pos)->_d == mkd_posTypeBeginning)

externref CONST_DATA mkd_pos mkdEnd;
#define mkdIsEnd(pos)           ((pos)->_d == mkd_posTypeEnd)

externref CONST_DATA mkd_pos mkdCurrent;
#define mkdIsCurrent(pos)       ((pos)->_d == mkd_posTypeCurrent)

externref CONST_DATA mkd_pos mkdDefaultStart;
#define mkdIsDefaultStart(pos) ((pos)->_d == mkd_posTypeDefaultStart)
```

Value	Description
mkdBeginning	Beginning of the stream.
mkdEnd	End of the stream.
mkdCurrent	Indicates the current position. Instead of using this field, callers should specify the current position explicitly whenever possible.
mkdDefaultStart	Default start.

Clients that previously passed NULL pointers to **prepare()** or **play()** should be able to easily convert these to **&mkdCurrent**, **&mkdBeginning**, or **&mkdEnd**, as appropriate.

Used by **mkd::pos**

mkd::wall

A structure that describes time in a wall clock or “real world” fashion. Since the **mkd::wall** data type does not contain time zone information, it is expected that most functions will be prototyped to use either **mkd::gmtWall** or **mkd::localWall**.

```
struct wall
{
    unsigned long  mkd_wallNano;
    unsigned short mkd_wallSec;
    unsigned short mkd_wallMin;
    unsigned short mkd_wallHour;
    unsigned short mkd_wallDay;
    unsigned short mkd_wallMonth;
    short          mkd_wallYear;
};
```

This structure has the following fields:

Field	Type	Description
mkd_wallNano	unsigned long	0 - 999999999 nanoseconds
mkd_wallSec	unsigned short	0 - 59 seconds
mkd_wallMin	unsigned short	0 - 59 minutes
mkd_wallHour	unsigned short	0 - 23 hours
mkd_wallDay	unsigned short	1 - 31 days
mkd_wallMonth	unsigned short	1 - 12 months
mkd_wallYear	short	year; positive is A.D., negative is B.C.

Used by **mdsCreateWall()**, **mkd::gmtWall**, **mkd::localWall**

mkd::gmtWall

A typedef that describes time in Greenwich Mean Time (GMT) wall clock time.

```
typedef wall gmtWall;
```

Used by **mkd::segInfo**, **mzz::sesInfo**, **mzabi::SchdFac::create()**

mkd::localWall

A typedef that describes time in local wall clock time.

```
typedef wall localWall;
```

Used by **mdsCreateWall()**

mkd::zone

A structure that specifies a time zone.

The **zone** structure can be used in addition to the **mkd::wall** structure for conversion of Greenwich Mean Time (GMT) times before display.

```
struct zone
{
    long        mkd_zoneOffset;
    boolean     mkd_zoneDaylight;
    string      mkd_zoneAbbrev;
};
```

This structure has the following fields:

Field	Type	Description
mkd_zoneOffset	long	Seconds offset of Greenwich Mean Time (GMT); positive is west of GMT, negative is east of GMT. Does not account for Daylight Savings Time.
mkd_zoneDaylight	boolean	True if Daylight Savings Time is active for given time zone, false otherwise.
mkd_zoneAbbrev	string	Timezone description, such as, PDT, EST, GMT, and so on.

Used by Currently unused.

mkd::prohib

A bitmask detailing prohibited rate control operations.

```
typedef unsigned long prohib;
    const prohib prohibError    = 0x80000000;
    const prohib prohibPause    = 0x00000001;
    const prohib prohibStop     = 0x00000002;
    const prohib prohibBlindFF  = 0x00000004;
    const prohib prohibBlindRW  = 0x00000008;
    const prohib prohibVisualFF = 0x00000010;
    const prohib prohibVisualRW = 0x00000020;
    const prohib prohibFrameAdv = 0x00000040;
    const prohib prohibFrameRew = 0x00000080;
```

Mask	Value	Description
prohibError	0x80000000	An invalid value.
prohibPause	0x00000001	Don't allow pause operation.
prohibStop	0x00000002	Don't allow stop operation. Currently unsupported.
prohibBlindFF	0x00000004	Don't allow scan forward operation. Currently unsupported.
prohibBlindRW	0x00000008	Don't allow scan backward operation. Currently unsupported.
prohibVisualFF	0x00000010	Don't allow visual fast-forward operation.
prohibVisualRW	0x00000020	Don't allow visual rewind operation.
prohibFrameAdv	0x00000040	Don't allow frame advance operation. Currently unsupported.
prohibFrameRew	0x00000080	Don't allow frame rewind operation. Currently unsupported.

Used by **mkd::segInfo**, **mkd::segment**

mkd::segCapMask

A bitmask that specifies rate control capabilities allowed or prohibited by an entity.

```
typedef unsigned long segCapMask;
    const segCapMask segCapVideo = 0x00000001;
    const segCapMask segCapVCBR  = 0x00000002;
    const segCapMask segCapVSeek = 0x00000004;
    const segCapMask segCapVScan = 0x00000008;
    const segCapMask segCapAudio = 0x00010000;
    const segCapMask segCapACBR  = 0x00020000;
    const segCapMask segCapASeek = 0x00040000;
    const segCapMask segCapAScan = 0x00080000;
```

Capability	Value	Description
segCapVideo	0x00000001	Can handle video.
segCapVCBR	0x00000002	Can handle constant bit rate video.
segCapVSeek	0x00000004	Can handle blind seeking.
segCapVScan	0x00000008	Can handle frame dropping.
segCapAudio	0x00010000	Can handle audio.
segCapACBR	0x00020000	Can handle constant bit rate audio.
segCapASeek	0x00040000	Can handle blind audio seeking.
segCapAScan	0x00080000	Can handle sample dropping.

Used by **mkd::segInfo**

mkd::segInfo

The **segInfo** structure contains the client information about a segment or stream. This may be information returned by the content manager as the result of a query, or information provided on a prepared stream. In particular, it includes rate control prohibitions imposed by the server, and gives display information (pixel size, frame rate, and so on) about the piece of content.

```
struct segInfo
{
    segment        mkd_segInfoSeg;
    string         mkd_segInfoType;
    string         mkd_segTitle;
    gmtWall        mkd_segCreateTime;
    unsigned long  mkd_segBitrate;
    long           mkd_segPresRate;
    compFormat     mkd_segCmpFmt;
    segCapMask     mkd_segCapabilities;
    unsigned long  mkd_segMilliseconds;
    long long      mkd_segByteLength;
    unsigned short mkd_segFrameHeight;
    unsigned short mkd_segFrameWidth;
    long           mkd_segAspectRatio;
    unsigned long  mkd_segFrameRate;
    long long      mkd_segInfoStartT;
    long long      mkd_segInfoEndT;
    contStatus     mkd_segInfoCStat;
    prohib         mkd_segProhibitions;
};
```

This structure has the following fields:

Field	Type	Description
mkd_segInfoSeg	**mkd::segment**	The segment.
mkd_segInfoType	string	Type of file (currently this is the file suffix).
mkd_segTitle	string	Title of segment or stream.
mkd_segCreateTime	**mkd::gmtWall**	Time when the segment or stream was created.
mkd_segBitrate	unsigned long	Bit rate at which stream is played.
mkd_segPresRate	long	Stored presentation rate.

Field	Type	Description
mkd_segCmpFmt	**mkd::compFormat**	Compression format of the content.
mkd_segCapabilities	**mkd::segCapMask**	Rate control capabilities.
mkd_segMilliseconds	unsigned long	Length of stream or segment, in milliseconds.
mkd_segByteLength	long long	Length of stream or segment, in Kbytes.
mkd_segFrameHeight	unsigned short	Height of frame, in pixels.
mkd_segFrameWidth	unsigned short	Width of frame, in pixels.
mkd_segAspectRatio	long	Height*10000 / width of *individual* pixels
mkd_segFrameRate	unsigned long	Frame rate (fps) *1000, for example, 29.97 fps -> 29970.
mkd_segInfoStartT	long long	Starting time of the file.
mkd_segInfoEndT	long long	Ending time of the file.
mkd_segInfoCStat	**mkd::contStatus**	Indicates whether content is on disk or tape.
mkd_segProhibitions	**mkd::prohib**	Prohibited rate control operations.

Used by **mkd::segInfoList**

mkd::segInfoList

A sequence of **mkd::segInfo** structures.

```
typedef sequence<segInfo> segInfoList;
```

Used by **mzs::stream::prepareSequence()**

mkd::segment

A structure used to describe a segment of a piece of content. It is generally used when requesting the delivery of such a segment.

```
struct segment
{
    string       mkd_segFile;
    pos          mkd_segStart;
    pos          mkd_segEnd;
    segMask      mkd_segFlags;
    prohib       mkd_segProhib;
};
```

This structure has the following fields:

Field	Type	Description
mkd_segFile	string	Name of segment file.
mkd_segStart	**mkd::pos**	Start position in segment.
mkd_segEnd	**mkd::pos**	Stop position in segment.
mkd_segFlags	**mkd::segMask**	Currently unused.
mkd_segProhib	**mkd::prohib**	Prohibited rate control operations.

Used by **mkd::segmentList**

mkd::segmentList

A sequence of **mkd::segment** structures.

```
typedef sequence<segment> segmentList;
```

Used by **mtcr::resolve::name()**

mkd::segMask

A segment mask can be used to specify particular options for a segment of content.

```
typedef unsigned long segMask;
    const segMask segMaskStatic     = 0x00000001;
    const segMask segMaskDynApp     = 0x00000002;
    const segMask segMaskRolling    = 0x00000004;
    const segMask segMaskTerminated = 0x00000008;
    const segMask segMaskTape       = 0x00000010;
```

Capability	Value	Description
segMaskStatic	0x00000001	Segment of a static file on disk.
segMaskDynApp	0x00000002	Segment of a dynamic file that is still being encoded, that is, “one-step encoding.”
segMaskRolling	0x00000004	Segment of a dynamic file that is being encoded and may also be deleted, that is, “continuous real-time feed.”
segMaskTerminated	0x00000008	Segment is a terminated, continuous feed.
segMaskTape	0x00000010	Segment of a file stored on tape.

Used by **mkd::segment**

mtux Data Types

mtuxLayer

An enumerated list that indicates the status of a Media Net initialization performed by either the **mtuxInit()** or **mtuxSimpleInit()** function or the termination performed by the **mtuxTerm()** function. Following a Media Net initialization or termination operation, the status is either success or the layer at which the operation failed.

```
typedef enum mtuxLayer
{
    mtuxLayerSuccess = 0,
    mtuxLayerInfo,
    mtuxLayerBadParam,
    mtuxLayerYs,
    mtuxLayerMn,
    mtuxLayerYo
} mtuxLayer;
```

This enumerated type has the following values:

Value	Description
mtuxLayerSuccess	Media Net successfully initialized or terminated.
mtuxLayerInfo	A help or version message was printed in response to a **-h** or **-V** argument in the *argLst* of the initialization function.
mtuxLayerBadParam	Either a malformed argument was found in the *argLst* of the initialization function or in a resource file, such as YSRESFILE.
mtuxLayerYs	System layer failure. This generally means there is a build inconsistency. Check your installation. YS failures usually lead immediately to core dumps.
mtuxLayerMn	Transport layer failure. This usually means that the Media Net address server cannot be connected; make sure your OMN_ADDR variable is set and the **mnaddrsrv** process is running.

Value	Description
mtuxLayerYo	Object layer failure. This indicates that the **mnorbsrv** process is not reachable.

Used by **mtuxInit()**, **mtuxSimpleInit()**, **mtuxTerm()**

mza Data Types

mza::LgCtntAtr

A structure containing the attribute information of an **mza::LgCtnt** object.

```
struct LgCtntAtr
{
    LgCtnt            lgCtntOR;
    string            name;
    string            desc;
    mkd::gmtWall      createDate;
    long              msecs;
    long              maxSugBufSz;
    long              maxRate;
    long              numClips;
    mkd::assetCookie  cookie;
    boolean           longFmt;
    ClipAtrLst        clipAtrLst;
    CtntAtrLst        ctntAtrLst;
};
```

This structure has the following fields:

Field	Type	Description
lgCtntOR	**mza::LgCtnt**	Object reference to the logical content object. This field is read-only and cannot be modified by **setAtr()**.
name	string	Name of the **mza::LgCtnt** object. Must be unique.
desc	string	Description of the logical content (optional).
createDate	**mkd::gmtWall**	Date the logical content was created.

Field	Type	Description
msecs	long	Calculated runtime length of all the clips, in milliseconds. (read-only)
maxSugBufSz	long	Maximum size of buffer for this **mza::LgCtnt** object. This is the largest *sugBufSz* value of all the **mza::Ctnt** objects referenced by this **mza::LgCtnt**. (read-only)
maxRate	long	Maximum bit rate. This is the largest *rate* value of all the **mza::Ctnt** objects referenced by this **mza::LgCtnt**. (read-only)
numClips	long	Number of clips in the logical content (calculated). This field is read-only and cannot be modified by **setAtr()**.
cookie	**mkd::assetCookie**	The asset cookie for this logical content.
longFmt	boolean	If true, then the **mza::Clip** and **mza::Ctnt** information is included.
clipAtrLst	**mza::ClipAtrLst**	If *longFmt* is true, this list is filled in with a list of all the clips assigned to this logical content.
ctntAtrLst	**mza::CtntAtrLst**	If *longFmt* is true, this list is filled in with the content information that corresponds to the clips in **mza::ClipAtrLst**.

Used by **mza::LgCtnt::getAtr()**

mza::LgCtntAtrLst

A sequence of **mza::LgCtntAtr** structures.

```
typedef sequence <LgCtntAtr> LgCtntAtrLst;
```

Used by **mza::LgCtntMgmt::lstAtr()**, **mza::LgCtntMgmt::lstAtrByNm()**, **mza::BlobMgmt::lstAtrByNm()**

mza::ClipAtr

A structure containing the attribute information of an **mza::Clip** object.

```
struct ClipAtr
{
    Clip      clipOR;
    Ctnt      ctntOR;
    string    name;
    string    desc;
    string    ctntNm;
    mkd::pos  startPos;
    mkd::pos  stopPos;
    boolean   assigned;
};
```

This structure has the following fields:

Field	Type	Description
clipOR	**mza::Clip**	Object reference to the clip object. This field is read-only and cannot be modified by **setAtr()**.
ctntOR	**mza::Ctnt**	Object reference to the content object this clip is from.
name	string	Name of the **mza::Clip** object.
desc	string	Description of the clip (optional).
ctntNm	string	Name of the content represented by the *ctntOR*.
startPos	**mkd::pos**	Start position of this clip from the beginning.
stopPos	**mkd::pos**	End position of this clip from the beginning.
assigned	boolean	True if clip is currently assigned to some logical content, false otherwise.

Used by **mza::LgCtnt::getAtrClipByPos()**, **mza::Clip::getAtr()**

mza::ClipAtrLst

A sequence of **mza::ClipAtr** structures.

```
typedef sequence <ClipAtr> ClipAtrLst;
```

Used by **mza::LgCtnt::lstAtrClips()**, **mza::ClipMgmt::lstAtr()**, **mza::ClipMgmt::lstAtrByCtnt()**, **mza::ClipMgmt::lstAtrByCtntNm()**, **mza::ClipMgmt::lstAtrByNm()**

mza::CtntAtr

A structure containing the attribute information of an **mza::Ctnt** object.

```
struct CtntAtr
{
    Ctnt              ctntOR;
    CtntPvdr          ctntPvdrOR;
    string            name;
    string            desc;
    mkd::gmtWall      createDate;
    string            filename;
    long long         len;
    long              msecs;
    long              rate;
    long long         firstTime;
    long long         lastTime;
    mkd::contFormat   format;
    mkd::prohib       prohibFlags;
    boolean           tagsFlag;
    boolean           multiRateFlag;
    boolean           reliableFlag;
    string            volLocation;
    mkd::contStatus   contStatus;
    boolean           assigned;
    long              sugBufSz;
};
```

This structure has the following fields:

Field	Type	Description
ctntOR	**mza::Ctnt**	Object reference to the content object. This attribute is read-only and cannot be modified by a call to **setAtr()**.
ctntPvdrOR	**mza::CtntPvdr**	Object reference to the content provider object that created and owns the content (optional).

Field	Type	Description
name	string	Name of the **mza::Ctnt** object.
desc	string	Description of the **mza::Ctnt** object (optional).
createDate	**mkd::gmtWall**	Date the content was created.
filename	string	The MDS tag file name for this content.
len	long long	Size of the content in bytes.
msecs	long	Total run length of the content, in milliseconds.
rate	long	Encoding rate of the content, in bps.
firstTime	long long	First time stamp in tag file.
lastTime	long long	Last time stamp in the tag file.
format	**mkd::contFormat**	All the format information (see **mkd.idl**).
prohibFlags	**mkd::prohib**	Restrictions on seek, scan, pause, and so on (see **mkd.idl**).
tagsFlag	boolean	True if there tags in the file, false otherwise.
multiRateFlag	boolean	True if this is multirate content, false otherwise.
reliableFlag	boolean	True if reliable transport is required, false otherwise.
volLocation	string	Volume location of the content (optional).
contStatus	**mkd::contStatus**	Location status (tape, disk) of the content.
assigned	boolean	True if content object is assigned to one or more clips, false otherwise.

Field	Type	Description
sugBufSz	long	Suggested buffer size (in bytes) to be set by client. If *Unknown*, value will be **mza::BufSzUnknown**.

Used by **mza::LgCtntFac::createCtnt()**, **mza::Ctnt::getAtr()**, **mza::Ctnt::setAtr()**, **mza::CtntFac::create()**

mza::CtntAtrLst

A sequence of **mza::CtntAtr** structures.

```
typedef sequence <CtntAtr> CtntAtrLst;
```

Used by **mza::CtntMgmt::lstAtr()**, **mza::CtntMgmt::lstAtrByNm()**, **mza::CtntMgmt::lstAtrByFileNm()**

mza::CtntPvdrAtr

A structure containing the attribute information of an **mza::CtntPvdr** object.

```
struct CtntPvdrAtr
{
    CtntPvdr  ctntPvdrOR;
    string    name;
    string    desc;
};
```

This structure has the following fields:

Field	Type	Description
ctntPvdrOR	**mza::CtntPvdr**	Object reference to the content provider object.
name	string	Name of the **mza::CtntPvdr** object.
desc	string	Description of the **mza::CtntPvdr** object (optional).

Used by **mza::CtntPvdr::getAtr()**, **mza::CtntMgmt::getAtrByNm()**

mza::CtntPvdrAtrLst

A sequence of **mza::CtntPvdrAtr** structures.

```
typedef sequence <CtntPvdrAtr> CtntPvdrAtrLst;
```

Used by **mza::CtntPvdrMgmt::lstAtr()**

mza::Itr

A structure for iterating through a list.

See *Using an Iterator* on page 3-12 for a discussion on how to use the **mza::Itr** structure.

```
struct Itr
{
    long Position;
    long NumItems;
};
```

This structure has the following fields:

Field	Type	Description
Position	long	On input, specifies the position in the list for the next group of items to be returned. On output, specifies the new position in the list following the last group of items returned, or -1 if all items have been returned.
NumItems	long	On input, specifies the maximum number of items to return at one time. On output, specifies the actual number of items returned.

Used by All **lst*** methods that return a ***Lst** sequence of datatypes.

mza::ObjLst

A generic sequence of objects.

```
typedef sequence <Object> ObjLst;
```

mza::opstatus

An enumerated type that indicates whether processing was complete before an exception.

```
enum opstatus
{
    completed,
    notComplete
}
```

This enumerated type has the following values:

Value	Description
completed	Operation successful.
notComplete	Operation failed.

mzabi Data Types

mzabi::SchdAtr

A structure containing the attribute information of an **mzabi::Schd** object.

```
struct SchdAtr
{
    Schd           schdOR;
    mkd::gmtWall   startDate;
    mkd::gmtWall   stopDate;
    eventType      type;
    ExpGrpAtr      expGrpAtr;
    ExpGrpAtr      chgGrpAtr;
    schdStatus     sts;
};
```

This structure has the following fields:

Field	Type	Description
schdOR	**mzabi::Schd**	Object reference to the scheduled event object.
startDate	**mkd::gmtWall**	The start time and date of the scheduled event.
stopDate	**mkd::gmtWall**	The stop time and date of the scheduled event.
type	**mzabi::eventType**	The type of scheduled event.
expGrpAtr	**mzabi::ExpGrpAtr**	The exporter group that should be informed of the scheduled event.
chgGrpAtr	**mzabi::ExpGrpAtr**	The exporter group that should be informed if the scheduled event changes.
sts	**mzabi::schdStatus**	The current status of the scheduled event.

Used by **mzabi::Schd::getAtr()**, **mzabi::Schd::setAtr()**

mzabi::SchdAtrLst

A sequence of **mzabi::SchdAtr** structures.

```
typedef sequence <SchdAtr> SchdAtrLst;
```

Used by **mzabi::SchdMgmt::lstAtrByDate()**

mzabi::ExpAtr

A structure containing the attribute information of an **mzabi::Exp** object.

```
struct ExpAtr
{
    Exp         expOR;
    string      name;
    string      implId;
    long        setupTime;
    expStatus   sts;
};
```

This structure has the following fields:

Field	Type	Description
expOR	**mzabi::Exp**	Object reference to the exporter object.
name	string	The name of the exporter.
implId	string	The *implId* of the **mzabix::exporter** object to bind to when making calls to the exporter interface methods.
setupTime	long	The time the exporter needs to set up prior to calling the **startEvent** method, in microseconds.
sts	**mzabi::expStatus**	The current status of the exporter.

Used by **mzabi::Exp::getAtr()**, **mzabi::Exp::setAtr()**

mzabi::ExpAtrLst

A sequence of **mzabi::ExpAtr** structures.

```
typedef sequence <ExpAtr> ExpAtrLst;
```

Used by **mzabi::ExpMgmt::lstAtr()**, **mzabi::ExpMgmt::lstAtrByNm()**

mzabi::ExpGrpAtr

A structure containing the attribute information of an **mzabi::ExpGrp** object.

```
struct ExpGrpAtr
{
    ExpGrp      expGrpOR;
    string      name;
    ExpAtrLst   expAtrLst;
};
```

This structure has the following fields:

Field	Type	Description
expGrpOR	**mzabi::ExpGrp**	Object reference to the exporter group object.
name	string	The name of the exporter group.
expAtrLst	**mzabi::ExpAtrLst**	List of **mzabi::ExpAtr** structures describing the **mzabi::Exp** objects belonging to the exporter group.

Used by **mzabi::ExpGrp::getAtr()**, **mzabi::ExpGrp::setAtr()**

mzabi::ExpGrpAtrLst

A sequence of **mzabi::ExpGrpAtr** structures.

```
typedef sequence <ExpGrpAtr> ExpGrpAtrLst;
```

Used by **mzabi::ExpGrpMgmt::lstAtr()**, **mzabi::ExpGrpMgmt::lstAtrByNm()**

mzabi::schdStatus

An enumerated type that reports the status of a scheduled event.

```
enum schdStatus
{
    scheduled_schdStatus,
    committed_schdStatus,
    exporting_schdStatus,
    exported_schdStatus,
    started_schdStatus,
    finished_schdStatus,
    finishing_schdStatus,
    exportError_schdStatus,
    executeError_schdStatus,
    unknown_schdStatus
};
```

This enumerated type has the following values:

Value	Description
scheduled_schdStatus	First state when created (not exportable).
committed_schdStatus	Scheduled event committed and ready for export.
exporting_schdStatus	Started exporting scheduled event to exporters.
exported_schdStatus	Scheduled event exported to all exporters.
started_schdStatus	Scheduled event started on one or more exporters.
finished_schdStatus	Scheduled event finished on all exporters.
finishing_schdStatus	Scheduled event is in the process of finishing.
exportError_schdStatus	Unable to export scheduled event to one or more exporters.
executeError_schdStatus	Failure in execution of one or more exporters.
unknown_schdStatus	Unknown status.

Used by **mzabix::exporter::startEvent()**, **mzabix::exporter::stopEvent()**

mzabi::expStatus

An enumerated type that reports the status of an exporter.

```
enum expStatus
{
    initializing_expStatus,
    operating_expStatus,
    problem_expStatus,
    stopping_expStatus,
    unknown_expStatus
};
```

This enumerated type has the following values:

Value	Description
initializing_expStatus	Exporter not yet ready to process events.
operating_expStatus	Exporter ready to process events.
problem_expStatus	Exporter detected an internal problem.
stopping_expStatus	Exporter shutting down.
unknown_expStatus	Exporter in an unknown state.

Used by **mzabi::Exp::setStatus()**

mzabi::eventType

An enumerated type that reports the type of scheduled event.

```
enum eventType
{
    synchShort_eventType,
    synchLong_eventType,
    asynch_eventType,
    change_eventType,
    unknown_eventType
};
```

This enumerated type has the following values:

Value	Description
synchShort_eventType	Scheduler service tells the exporter the start and stop times of a scheduled event in one call.
synchLong_eventType	Scheduler service tells the exporter the start and stop times of a scheduled event in different calls.
asynch_eventType	End of scheduled event is not determined by Scheduler service.
change_eventType	Scheduler service notifies change in schedule event only.
unknown_eventType	Unknown event type.

Used by **mzabi::SchdFac::create()**

mzabin Data Types

mzabin::NvodAtr

A structure containing the attribute information of an **mzabin::Nvod** object.

```
struct NvodAtr
{
    Nvod          nvodOR;
    mzabi::Schd   schdOR;
    Chnl          chnlOR;
    mza::LgCtnt   lgCtntOR;
    loopType      loop;
    nvodStatus    sts;
};
```

This structure has the following fields:

Field	Type	Description
nvodOR	**mzabin::Nvod**	Object reference to the NVOD object.
schdOR	**mzabi::Schd**	Object reference to the scheduled event object.
chnlOR	**mzabin::Chnl**	Object reference to the channel object to play the logical content on.
lgCtntOR	**mza::LgCtnt**	Object reference to the logical content object to play for the scheduled event.
loop	**mzabin::loopType**	Type of looping desired.
sts	**mzabin::nvodStatus**	The current status of the NVOD object.

Used by **mzabin::Nvod::getAtr()**, **mzabin::Nvod::setAtr()**

mzabin::NvodAtrLst

A sequence of **mzabin::NvodAtr** structures.

```
typedef sequence <NvodAtr> NvodAtrLst;
```

Used by Currently unused.

mzabin::NvodSchdAtr

A structure containing the logical content, channel, and scheduling information associated with an **mzabin::Nvod** object.

```
struct NvodSchdAtr
{
    Nvod            nvodOR;
    mzabi::Schd     schdOR;
    mkd::gmtWall    startDate;
    mkd::gmtWall    stopDate;
    mza::LgCtnt     lgCtntOR;
    string          assetCookie
    string          lgCtntName;
    string          chnlLabel;
    sb4             chnlNum;
    string          netAddr;
    long            maxBitRate;
    nvodStatus      sts;
    mkd::compFormat format;
    loopType        loop;
};
```

This structure has the following fields:

Field	Type	Description
nvodOR	**mzabin::Nvod**	Object reference to the NVOD object.
schdOR	**mzabi::Schd**	Object reference to the scheduled event object.
startDate	**mkd::gmtWall**	The start time and date of the scheduled event.
stopDate	**mkd::gmtWall**	The stop time and date of the scheduled event.

Field	Type	Description
lgCtntOR	**mza::LgCtnt**	Object reference to the logical content object to play for the scheduled event.
assetCookie	string	Asset cookie for the logical content object.
lgCtntName	string	Name of the logical content object.
chnlLabel	string	Label or name assigned to the channel object.
chnlNum	sb4	Number assigned to the channel object.
netAddr	string	Physical network address of the channel object.
maxBitRate	long	The maximum bit rate for the logical content object.
sts	**mzabin::nvodStatus**	The current status of the NVOD object.
format	**mkd::compFormat**	Compression format of the logical content file.
loop	**mzabin::loopType**	Type of looping desired.

Used by Currently unused.

mzabin::NvodSchdAtrLst

A sequence of **mzabin::NvodSchdAtr** structures.

```
typedef sequence <NvodSchdAtr> NvodSchdAtrLst;
```

Used by **mzabin::NvodMgmt::lstAtrByDate()**, **mzabin::NvodMgmt::lstAtrByLCNm()**

mzabin::ChnlAtr

A structure containing the attribute information of an **mzabin::Chnl** object.

```
struct ChnlAtr
{
    Chnl    chnlOR;
    string  label;
    long    chnlNum;
    string  netAddr;
};
```

This structure has the following fields:

Field	Type	Description
chnlOR	**mzabin::Chnl**	Object reference to the channel object.
label	string	Label or name assigned to the channel object.
chnlNum	long	Number assigned to the channel object.
netAddr	string	Physical network address of the channel object.

Used by **mzabin::Chnl::setAtr()**, **mzabin::Chnl::getAtr()**

mzabin::ChnlAtrLst

A sequence of **mzabin::ChnlAtr** structures.

```
typedef sequence <ChnlAtr> ChnlAtrLst;
```

Used by **mzabin::ChnlMgmt::lstAtr()**, **mzabin::ChnlMgmt::lstAtrByNm()**

mzabin::nvodStatus

An enumerated type that reports the status of an NVOD object.

```
enum nvodStatus
{
    unknown_nvodStatus,
    defined_nvodStatus,
    prepared_nvodStatus,
    playing_nvodStatus,
    played_nvodStatus,
    errBadContent_nvodStatus,
    errPlayout_nvodStatus,
    errOther_nvodStatus
};
```

This enumerated type has the following values:

Value	Description
unknown_nvodStatus	Unknown status.
defined_nvodStatus	Initial value indicating NVOD object has been defined.
prepared_nvodStatus	Event has been prepared for playout.
playing_nvodStatus	Event is playing.
played_nvodStatus	Event has completed normally.
errBadContent_nvodStatus	Event failed due to missing or bad logical content.
errPlayout_nvodStatus	Event failed to play.
errOther_nvodStatus	Some other error.

Used by **mzabin::Nvod::setStatus()**

mzabin::loopType

An enumerated type that sets the looping characteristics on logical content scheduled.

```
enum loopType
{
    loopNone_loopType,
    loopAll_loopType
};
```

This enumerated type has the following values:

Value	Description
loopNone_loopType	No looping.
loopAll_loopType	Loop the entire logical content.

Used by **mzabin::NvodFac::create(), mzabin::NvodFac::createSchd()**

mzabix Data Types

mzabix::statusInfo

A structure containing the status of an **mzabix::exporter** object.

```
struct statusInfo
{
    mzabi::expStatus     sts;
    string               stsMsg;
};
```

This structure has the following fields:

Field	Type	Description
sts	**mzabi::expStatus**	The current status of the exporter.
stsMsg	string	A status text message.

Used by **mzabix::exporter::getStatus()**

mzabix::statusInfoLst

A sequence of **mzabix::statusInfo** structures.

```
typedef sequence <statusInfo> statusInfoLst;
```

Used by Currently unused.

mzabix::eventStatusInfo

A structure containing the status of a scheduled event the **mzabix::exporter** object is exporting.

```
struct eventStatusInfo
{
    mzabi::Schd         schdOR;
    mzabi::schdStatus   sts;
    string              stsMsg;
};
```

This structure has the following fields:

Field	Type	Description
schdOR	**mzabi::Schd**	Object reference to the scheduled event object.
sts	**mzabi::schdStatus**	The current status of the scheduled event.
stsMsg	string	A status text message.

Used by **mzabix::exporter::getEventStatus()**

mzabix::eventStatusInfoLst

A sequence of **mzabix::eventStatusInfo** structures.

```
typedef sequence <eventStatusInfo> eventStatusInfoLst;
```

Used by **mzabix::exporter::getAllEventStatus()**

mzc Data Types

mzc::circuit

A structure providing access to both the circuit object and the associated **mzc::cktInfo** structure.

```
struct circuit
{
    ckt      or;
    cktInfo  info;
};
```

This structure has the following fields:

Field	Type	Description
or	**mzc::ckt**	Object reference for this circuit.
info	**mzc::cktInfo**	Structure containing information about this circuit.

Used by **mzz::ses::AddCircuit()**, **mzc::factory::GetCircuits()**, **mzc::circuit**

mzc::circuits

A sequence of **mzc::circuit** objects.

```
typedef sequence<circuit> circuits;
```

Used by **mzz::ses::GetCircuits()**

mzc::cktInfo

A structure that holds information about a circuit. If a stream is associated with a circuit, it is found in the *streamRef* field of this structure. Clients cannot change the value of this field.

```
struct cktInfo
{
    commProperty    props;
    logicalAddress  mna;
    mzs::stream     streamRef;
    channel         upstream;
    channel         downstream;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Communications properties.
mna	**mzc::logicalAddress**	Media Net address associated with circuit. Valid only if a control circuit.
streamRef	**mzs::stream**	Stream associated with circuit. Valid only if an isochronous data circuit. May be NULL if no stream is open.
upstream	**mzc::chnl**	Channel for client -> server data.
downstream	**mzc::chnl**	Channel for server -> client data.

Used by **mzc::ckt::GetInfo()**, **mzc::ckt::Rebuild()**, **mzc::cktInfos**

mzc::cktInfos

A sequence of **mzc::cktInfo** structures.

```
typedef sequence<cktInfo> cktInfos;
```

Used by Currently unused.

mzc::cktreq

A discriminated union providing information about the kind of circuit to use; either symmetric (**mzc::cktreqSym**) or asymmetric (**mzc::cktreqAsym**).

```
enum cktreqType
{
    cktreqTypeSymmetric,
    cktreqTypeAsymmetric
};

union cktreq switch(cktreqType)
{
    case cktreqTypeSymmetric: cktreqSym sym;
    case cktreqTypeAsymmetric: cktreqAsym asym;
};
```

Used by **mzc::cktspec**

mzc::cktreqAsym

A structure containing information used when allocating an asymmetric **mzc::cktreq** circuit.

```
struct cktreqAsym
{
    commProperty props;
    chnlspec     upchnl;
    chnlspec     downchnl;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Communications properties.
upchnl	**mzc::chnlspec**	Value that describes the upstream channel specification for this circuit.
downchnl	**mzc::chnlspec**	Value that describes the downstream channel specification for this circuit.

Used by **mzc::cktreq**

mzc::cktreqSym

A structure containing information used when allocating a symmetric **mzc::cktreq** circuit.

```
struct cktreqSym
{
    commProperty props;
    chnlspec     chnl;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Communications properties.
chnl	**mzc::chnlspec**	Value that describes the channel specification for this circuit.

Used by **mzc::cktreq**

mzc::ckts

A sequence of **mzc::ckt** objects.

```
typedef sequence<ckt> ckts;
```

Used by Currently unused.

mzc::cktspec

A discriminated union that specifies a circuit. Can point to **nothing**, an already existing circuit, or contain a request for a new circuit.

```
enum cktspecType
{
    cktspecTypeNone,
    cktspecTypeRequest,
    cktspecTypeCircuit
};

union cktspec switch(cktspecType)
{
    case cktspecTypeNone: long none;
    case cktspecTypeRequest: cktreq req;
    case cktspecTypeCircuit: circuit ckt;
};
```

This union has the following fields:

Field	Type	Description
none	long	Nothing.
req	**mzc::cktreq**	A discriminated union containing information for a new circuit.
ckt	**mzc::circuit**	Structure containing information for an existing circuit.

Used by **mzz::factory::AllocateSession()**, **mzz::ses::AddCircuit()**, **mzc::factory::BuildCircuit()**, **mzc::ckt::Rebuild()**, **mzc::chnlspecs**

mzc::cktspecs

A sequence of **mzc::cktspec** unions.

```
typedef sequence<cktspec> cktspecs;
```

Used by Currently unused.

mzc::clientDeviceId

Indicates which client a session is allocated to. A *clientDeviceId* has to be provided by the client (or a proxy sever) when the session is allocated. See *Allocate a Session and Build a Control Circuit* on page 2-7.

```
typedef sequence<octet> clientDeviceId;
```

Used by **mzc::factory::BuildCircuit()**, **mzz::factory::AllocateSession()**

mzc::channel

A structure providing an object reference for a channel as well as the structure containing information about the channel.

```
struct channel
{
    chnl or;
    chnlInfo info;
};
```

This structure has the following fields:

Field	Type	Description
or	**mzc::chnl**	Object reference for an **mzc::chnl** object.
info	**mzc::chnlInfo**	Structure containing information about the **mzc::chnl** object.

Used by **mzc::ckt::BindXSM**, **mzc::chnlInfo**, **mzc::chnlspec**

mzc::channels

A sequence of **mzc::channel** structures.

```
typedef sequence<channel> channels;
```

Used by Currently unused.

mzc::chnlInfo

A structure containing the attribute information of an **mzc::chnl** object.

```
struct chnlInfo
{
    commProperty   props;
    transportType  transport;
    link           comm;
    unsigned long  bitrate;
    logicalAddress mna;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Channel properties.
transport	**mzctt::transportType**	Transport wrapping used. Currently unused.
comm	**mzc::link**	Communications link to use.
bitrate	unsigned long	Channel bit rate.
mna	**mzc::logicalAddress**	Valid only if the channel is a control channel.

Used by **mzc::chnl::GetInfo()**, **mzc::chnl::Rebuild()**

mzc::chnlInfos

A sequence of **mzc::chnlInfo** structures.

```
typedef sequence<chnlInfo> chnlInfos;
```

Used by Currently unused.

mzc::chnlreq

A structure that specifies the properties of a new channel. Clients pass this structure when calling the **mzc::BuildCircuit()** method to request a new circuit. This structure is described by the **mzc::chnlspec** value, which is described in the **mzc::cktreq** value passed to the **mzc::BuildCircuit()** method.

```
struct chnlreq
{
    commProperty  props;
    transportType transport;
    netproto      protocol;
    unsigned long bitrate;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Requested channel properties.
transport	**mzctt::transportType**	Transport wrapping used. Currently unused.
protocol	**mzc::netproto**	Requested protocol/address info.
bitrate	unsigned long	Requested bit rate.

Used by **mzc::chnlspec**

mzc::chnlreqx

A structure that specifies the properties of an **mzc::chnl** object. The contents of this structure are derived by the circuit manager from the client-specified **mzc::chnlreq** structure and information in the configuration file.

```
struct chnlreqx
{
    commProperty  props;
    transportType transport;
    netproto      protocol;
    netapi        software;
    netif         hardware;
    pktinfo       packet;
    unsigned long bitrate;
};
```

This structure has the following fields:

Field	Type	Description
props	**mzc::commProperty**	Requested channel properties.
transport	**mzctt::transportType**	Requested transport wrapping. Currently unused.
protocol	**mzc::netproto**	Requested protocol/address info.
software	**mzc::netapi**	API used to access network hardware (sockets, TLI, and so on).
hardware	**mzc::netif**	Hardware interface to network.
packet	**mzc::pktinfo**	Packet size parameters for link.
bitrate	unsigned long	Requested bit rate.

Used by **mzc::chnl::Rebuild()**

mzc::chnls

A sequence of **mzc::chnl** objects.

```
typedef sequence<chnl> chnls;
```

Used by Currently unused.

mzc::chnlspec

A discriminated union that specifies the type of channel to use. Can be none, point to an existing channel, or refer to a new channel by pointing to an **mzc::chnlreq**. See *cktspec Circuit Specification* on page 2-8 for more information.

```
enum chnlspecType
{
    chnlspecTypeNone,
    chnlspecTypeRequest,
    chnlspecTypeChannel
;

union chnlspec switch(chnlspecType)
{
    case chnlspecTypeNone: long    none;
    case chnlspecTypeRequest: chnlreq req;
    case chnlspecTypeChannel: channel chnl;
};
```

This union has the following fields:

Field	Type	Description
none	long	Nothing.
req	**mzc::chnlreq**	Structure containing information for a new channel.
ckt	**mzc::channel**	Structure containing information for an existing channel.

Used by **mzc::ckt::BindXSM()**, **mzc::cktreqAsym**, **mzc::cktreqSym**

mzc::chnlspecs

A sequence of **mzc::chnlspec** structures.

```
typedef sequence<chnlspec> chnlspecs;
```

Used by Currently unused.

mzc::commProperty

A bitmask detailing the properties of a channel or circuit. As described in *Circuits* on page 1-5, synchronous circuits specify a single channel, which is either unidirectional (**propUp** *or* **propDown**) or bidirectional (**propUp** *and* **propDown**). Asynchronous circuits specify two channels that are either **propUp** *or* **propDown** for unidirectional asynchronous circuits, or **propUp** *and* **propDown** for bidirectional asynchronous circuits.

```
typedef unsigned long commProperty;
  const commProperty propNull              = 0x00000000;
  const commProperty propDown              = 0x00000001;
  const commProperty propUp                = 0x00000002;
  const commProperty propPointcast         = 0x00000010;
  const commProperty propMulticast         = 0x00000020;
  const commProperty propBroadcast         = 0x00000040;
  const commProperty propControl           = 0x00000100;
  const commProperty propData              = 0x00000200;
  const commProperty propIsochronousData   = 0x00000400;
  const commProperty propTransientConnect  = 0x00001000;
  const commProperty propPersistantConnect = 0x00002000;
  const commProperty propDisabled          = 0x00010000;
  const commProperty propGroupDisabled     = 0x00020000;
```

Mask	Value	Description
propNull	0x00000000	No special properties.
Direction (mandatory property) **Not mutually exclusive: select one or both**		
propDown	0x00000001	Channel / circuit server --> client.
propUp	0x00000002	Channel / circuit server <-- client.
Link allocation type (mandatory property) **Mutually exclusive: select at least one and only one**		
propPointcast	0x00000010	Channel / circuit is point to point.
propMulticast	0x00000020	Channel / circuit is point to multipoint, for some specified set of points.
propBroadcast	0x00000040	Channel / circuit is point to multipoint, for some unspecified set of points.

Mask	Value	Description
Traffic type (mandatory property) **Not mutually exclusive: select one, two, or all three**		
propControl	0x00000100	Channel/circuit is Media Net transport addressable and as such has an associated Media Net address. A control channel/circuit must have a valid Media Net address.
propData	0x00000200	Channel/circuit may transport non-isochronous data.
propIsochronousData	0x00000400	Channel/circuit is capable of transporting isochronous data (such as the video pump). An isochronous circuit will have an associated **mzs::stream** reference (which may be NULL if unbound).
Persistence (mandatory property) **Mutually exclusive: select at least one and only one**		
propTransientConnect	0x00001000	Channel may disconnect/reconnect within the lifetime of a containing circuit (such as a 28.8 KB dial-up POTS connection).
propPersistantConnect	0x00002000	Channel remains connected throughout the containing circuit's lifetime.
Availability for allocation (mandatory property)		
propDisabled	0x00010000	Channel is disabled (and may not be assigned to a circuit).
propGroupDisabled	0x00020000	Channel belongs to a channel group that is disabled.

Used by **mzc::cktreqAsym**, **mzc::cktreqSym**, **mzc::chnlInfo**, **mzc::chnlreq**, **mzc::cktInfo**

mzc::logicalAddress

A Media Net logical address. This address is in network order and 8 octets long.

```
typedef sequence <octet, 8> logicalAddress;
```

Used by **mzc::chnlInfo**, **mzc::cktInfo**

mzc::link

A structure containing information about the network connection.

```
struct link
{
    string    name;
    netproto  protocol;
    netapi    software;
    netif     hardware;
    pktinfo   packet;
};
```

This structure has the following fields:

Field	Type	Description
name	string	Identifier for link.
protocol	**mzc::netproto**	The protocol, for example “UDP,” “AAL5-SVC,” or “TCP.”
software	**mzc::netapi**	API used to access network hardware (sockets, TLI, and so on).
hardware	**mzc::netif**	Hardware interface to network (FDDI card and so on).
packet	**mzc::pktinfo**	Packet size parameter for link.

Used by **mzc::chnlInfo**

mzc::netapi

A structure containing some information about the network setup.

Given the same set of hardware and the same protocol stack, there may be different ways for a client to communicate. For example, there is no standard way of communicating with an ATM driver. In theory, some very sophisticated clients may get the information about the communications software from the **netapi** structure. In practice, this structure is usually used internally by the server.

```
struct netapi
{
    string           name;
    sequence<octet>  info;
};
```

This structure has the following fields:

Field	Type	Description
name	string	“TLI,” “BSD sockets,” and so on.
info	sequence <octet>	API-specific information used only by channel providers.

Used by **mzc::link**

mzc::netif

A structure representing the hardware interface used by the client. Currently used by the server only, but also used in the **mzc::link** structure.

```
struct netif
{
    string           name;
    string           hostname;
    string           devicename;
    sequence<octet>  info;
    unsigned long    curbr;
    unsigned long    maxbr;
};
```

This structure has the following fields:

Field	Type	Description
name	string	Identifier for interface.
hostname	string	Name of host containing the hardware.
devicename	string	“/dev/fddi0,”and so on.
info	sequence<octet>	API-specific information used only by channel providers.
curbr	unsigned long	Current bit rate allocated on device (bps).
maxbr	unsigned long	Maximum bit rate the device supports (bps).

Used by **mzc::link**

mzc::netproto

A structure representing addressing information.

```
struct netproto
{
    string name;
    string info;
};
```

This structure has the following fields:

Field	Type	Description
name	string	“UDP,” “TCP,” “AAL5-SVC,” and so on.
info	string	Protocol-specific information such as IP address and port for UDP.

Used by **mzc::link**, **mzc::chnlreq**

mzc::netprotos

A sequence of **mzc::netproto** structures.

```
typedef sequence<netproto> netprotos;
```

Used by Currently unused.

mzc::pktinfo

A structure providing a way of representing packet parameters using minimum and maximum size and packet moduli. Moduli specify in what steps can you change the packet size, for example, if the minimum size is 4 K and the maximum size is 8 K, and the packet modulos is 2 K, valid sizes for the packet are 4 K, 6 K, and 8 K.

```
struct pktinfo
{
    unsigned long pref_size;
    unsigned long max_size;
    unsigned long modulos;
};
```

This structure has the following fields:

Field	Type	Description
pref_size	unsigned long	Preferred packet size, in bytes.
max_size	unsigned long	Maximum packet size, in bytes.
modulos	unsigned long	Packets must be an integral multiple of this size, in bytes.

Used by **mzc::link**

mzctt::transportType

Transport types recognized by channel providers. Currently unused.

```
enum transportType
{
    transportTypeNone,
    transportTypeOGF,
    transportTypeMPEG2,
    transportTypeDSMCC_UN
};
```

This enumerated type has the following values:

Value	Description
transportTypeNone	No transport wrapping.
transportTypeOGF	Oracle Generic Framing (downstream only).
transportTypeMPEG2	MPEG-2 transport (downstream only).
transportTypeDSMCC_UN	DSM-CC User-to-Network (upstream only).

mzs Data Types

mzs::bootMask

The **bootMask** currently contains a single flag, **bootAutoClose**, that is passed to the **mzs::factory::boot()** and **mzs::stream::bootMore()** methods to deallocate the stream context when the boot process has completed.

```
typedef unsigned long bootMask;

const bootMask bootAutoClose = 0x00000001;
```

Used by **mzs::factory::boot(), mzs::stream::bootMore()**

mzs::bootRespInfo

A structure containing the number of blocks that the boot file was wrapped in; passed to the upstream server when **mzs::factory::boot()** is called.

```
struct bootRespInfo
{
    unsigned short mzsbootBlock_sz;
};
```

Used by **mzs::factory::boot()**

mzs::capMask

When you allocate a stream, the **capMask** contains flags to inform the stream service of the client's capabilities.

```
typedef unsigned long capMask;
    const capMask capAudio  = 0x00000001;
    const capMask capVideo  = 0x00000002;
    const capMask capMultipleStreams = 0x0000004;
    const capMask capMpeg1  = 0x00000010;
    const capMask capMpeg2  = 0x00000020;
    const capMask capJpeg   = 0x00000040;
    const capMask capOrca   = 0x00000080;
    const capMask capRawKey = 0x00000100;
    const capMask capSeek   = 0x00001000;
    const capMask capScan   = 0x00002000;
    const capMask capPause  = 0x00004000;
    const capMask capFrame  = 0x00008000;
    const capMask capSkip   = 0x00100000;
    const capMask capNoSeqChange   = 0x01000000;
    const capMask capBoot   = 0x00040000;
    const capMask capBootDSMCC = 0x00080000;
```

The **capMask** provides a flag for each client capability**.**

Flag	Description
General Flags	
capAudio	Client device can play audio.
capVideo	Client device can play video.
capMultipleStreams	Client device can play multiple streams at once (not currently supported).
Video Flags	
capMpeg1	Client device can play MPEG-1 system streams.
capMpeg2	Client device can play MPEG-2 transport streams.
capJpeg	Client device can display JPEG images.
capOrca	Client device can play Orca streams (not supported).
capRawKey	At least one raw key format (AVI, WAV, OSF, and so on).

Flag	Description
capSeek	Client device can do a blind seek in a stream.
capScan	Client device can scan forward or backward in a stream. This, in effect, means visual fast forward and rewind.
capPause	Client device can pause with a single frame on-screen.
capFrame	Client device can advance, frame-by-frame, forwards or backwards in a stream.
capSkip	Client device can display a legal MPEG-2 stream created by skipping frames in another MPEG-2 stream.
capNoSeqChange	The stream service will not prepare a sequence where the bit rate, format, pixel aspect ratio, frame rate, or dimensions change between segments.
Special Flags (Consult Oracle before using)	
capBoot	Used internally when booting the client device. An application should never pass this flag.
capBootDSMCC	Used internally to aid set-top box boot.

Used by **mzs::factory::alloc()**, **mzs::internals**

mzs::event

A union that describes a generic event for the event channel. These events are documented in Appendix D.

```
union event switch(evType)
   {
      case evTypeAlloc: evAlloc event_alloc;
      case evTypePrepare: evPrepare event_prepare;
      case evTypeSeqPrepare: evSeqPrepare event_seqPrepare;
      case evTypePlay: evPlay event_play;
      case evTypeFinish: evFinish event_finish;
      case evTypeDealloc: evDealloc event_dealloc;
      case evTypeDenial: evDenial event_denial;
   };
```

This union has the following fields:

Event	Value	Description
event_alloc	**evAlloc**	Indicates an **mzs::factory::alloc()** call.
event_prepare	**evPrepare**	Indicates an **mzs::stream::prepare()** call.
event_seqPrepare	**evSeqPrepare**	Indicates an **mzs::stream::prepareSequence()** call.
event_play	**evPlay**	Indicates an **mzs::stream::play()** call.
event_finish	**evFinish**	Indicates an **mzs::stream::finish()** call.
event_dealloc	**evDealloc**	Indicates an **mzs::stream::dealloc()** call.
event_denial	**evDenial**	Indicates an **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** request was denied.

mzs::finishFlags

Values used to identify how a stream prepare is to be completed.

```
typedef unsigned long finishFlags;
```

```
const finishFlags finishOne = 0x00000001;
const finishFlags finishAll = 0x00000002;
const finishFlags finishLoop = 0x00000004;
```

Because the system currently does not support multiple outstanding prepared instances, **finishOne** and **finishAll** are effectively equal.

Flag	Value	Description
finishOne	0x00000001	Finish the current prepare.
finishAll	0x00000002	Finish all prepares for this context.
finishLoop	0x00000004	Terminates an existing loop, but *does not* finish the current prepare.

Used by mzs::stream::**finish()**

mzs::instance

A descriptor for the prepared stream returned by **prepare()** or **prepareSequence()**. Eventually, the stream service might allow a client to have multiple, simultaneously prepared instances to pass interchangeably to calls like **mzs::stream::play()**. If a client prepares an instance when one is already prepared, the first becomes unusable.

```
typedef unsigned long instance;
```

Used by **mzs::stream::prepare()**, **mzs::stream::prepareSequence()**, **mzs::stream::play()**, **mzs::stream::pause()**, **mzs::stream::playFwd()**, **mzs::stream::playRev()**, **mzs::stream::frameFwd()**, **mzs::stream::frameRev()**, **mzs::stream::getPos()**, **mzs::stream::finish()**

mzs::internals

Stream-specific statistics. Exposed to monitoring facilities through the **mzs::stream::query()** method.

```
struct internals
{
    state              mzs_intActivity;
    capMask            mzs_intCaps;
    unsigned long      mzs_intMaxBitrate;
    unsigned long      mzs_intLastBitrate;
    unsigned long      mzs_intCurrentBitrate;
    long               mzs_intPlayRate;
    mkd::segmentList   mzs_intCurSegs;
};
```

This structure has the following fields:

Field	Type	Description
mzs_intActivity	**mzs::state**	Current state of stream.
mzs_intCaps	**mzs::capMask**	Capabilities.
mzs_intMaxBitrate	unsigned long	Maximum bit rate.
mzs_intLastBitrate	unsigned long	Last requested bit rate.
mzs_intCurrentBitrate	unsigned long	The current bit rate requested of the video pump. (When playing a sequence with varying bit rates, this is not necessarily the bit rate currently being delivered by the video pump.)
mzs_intPlayRate	long	Current presentation rate.
mzs_intCurSegs	**mkd::segmentList**	Current prepared segments.

Used by **mzs::stream::query()**

mzs::mzs_notify

An enumerated type that lists the events that may cause the server to call the client back.

```
enum mzs_notify
{
    mzs_notify_playDone,
    mzs_notify_finish,
    mzs_notify_badFile,
    mzs_notify_srvInternal
};
```

This enumerated type has the following values:

Value	Description
mzs_notify_playDone	Play request from the client has completed.
mzs_notify_finish	Client called **mzs::stream::finish()**.
mzs_notify_badFile	Server was asked to play a content file that does not exist or read errors occurred.
mzs_notify_srvInternal	When trying to process a request, server had to stop due to an internal error.

Used by **mzs_stream_cliCallbackHdlr**

mzs::playFlags

Flags passed to **mzs::stream::prepare()** and **mzs::stream::prepareSequence()** to define how the stream is to be played.

It is illegal to pass conflicting flags, such as both **playWait** and **playNow**. It is also illegal to pass **playLoop** and **playLoopLast**, or to send **playLoopLast** when only submitting one segment.

```
typedef unsigned long playFlags;
    const playFlags playWait  = 0x00000001;
    const playFlags playNow   = 0x00000002;
    const playFlags playNext  = 0x00000004;
    const playFlags playLoop  = 0x00000008;
    const playFlags playLoopLast = 0x00000010;
```

Flag	Value	Description
playWait	0x00000001	The default. The stream service prepares the requested assets and then waits for further commands.
playNow	0x00000002	Stream service plays the clip as soon as the **prepare** operation has been completed. Clients that intend to immediately follow a prepare operation with a play operation are urged to use this flag to reduce latency.
playNext	0x00000004	Not defined or implemented.
playLoop	0x00000008	Asks the stream service to prepare to loop the entire submitted play request until told to stop.
playLoopLast	0x00000010	Asks the stream service to loop the last segment in a submitted sequence. Currently unsupported.

Used by **mzs::stream::prepare()**, **mzs::stream::prepareSequence()**

mzs::state

An enumerated type that specifies the stream state. Derived from DSM-CC stream states, described in the International Standard ISO / IEC 13818-6 (*Information Technology—Generic coding of moving pictures and associated audio information—Part 6: Extension for Digital Storage Media Command and Control*).

The stream state is exposed to monitoring tools on a per-stream basis. The states enumerated here do not exactly correspond to DSM-CC stream states and the server is never in states ST or STP.

```
enum state
{
    stateDead,
    stateBoot,
    stateStream,
    stateClip,
    stateMoreClips,
    stateIdle,
    statePause,
    stateFinished,
    statePrepared
};
```

This enumerated type has the following states:

State	Corresponding DSM-CC state	Description
stateDead	U	Stream is dead.
stateBoot	U	Stream is booting.
stateStream	T	Will pause at end of stream (entering EOS).
stateClipW	TP	Will pause at endPos specified (entering P).
stateMoreClips	PST	Playing with multiple segments pending.
stateIdle	O	No segments pending.
statePause	P	Play rate of zero, one segment pending.

State	Corresponding DSM-CC state	Description
stateFinished	EOS	Same as pause, except initiated by server.
statePrepared	P	Best represented as P.

Used by **mzs::stream::getPos**

mzs_stream_cliCallbackHdlr

The **mzs_stream_cliCallbackHdlr** type describes a client callback function. The client passes a pointer to such a function into the mzs client-side interface. When a callback event arrives from the server, this function is called to inform the client.

```
typedef void (*mzs_stream_cliCallbackHdlr)
             (dvoid *ctx, mzs_notify reason);
```

This is an example callback function that a client might provide:

```
void TestCliFunc1(dvoid *argv, mzs_notify reason)
{
  ClientDefinedType *p = (ClientDefinedType *) argv;

(void)fprintf(stderr,
               “TestCliFunc1 called from mzs, argv=%d, *reason=%d\n”,
                p->ClientDefinedFields, (ub4)reason);
}
```

Used by **mza::stream::setCallback()**

mzz Data Types

mzz::sessProperty

A bitmask detailing properties of a session that is being requested.

```
typedef unsigned long sessProperty;
const sessProperty sessNull      = 0x00000000;
const sessProperty sessProxyId   = 0x00000001;
const sessProperty sessNeedsBoot = 0x00000002;
const sessProperty sessNeedsMnA  = 0x00000004;
const sessProperty sessProxySes  = 0x00000008;
```

Mask	Value	Description
sessNull	0x00000000	No special properties.
sessProxyID	0x00000001	Proxy ID specified is not known to client device and must be communicated to the client.
sessNeedsBoot	0x00000002	Client device has to be booted.
sessNeedsMnA	0x00000004	Client device cannot contact Media Net Address Server independently and therefore needs to be assigned an address via a proxy (such as the downstream).
sessProxySes	0x00000008	Session is being allocated by a third party on behalf of the client device.

Used by **mzz::factory::AllocateSession()**

mzz::clientDevice

A structure representing a device that an end user would use to access the real-time server (for example a PC, set-top box, or network computer). A client device is uniquely identified by some opaque key. This key may vary from deployment to deployment because it is up to the service provider to decide what to use as a client device identifier. The Oracle Video Server assumes that the client ID is unique but makes no other assumptions.

```
struct clientDevice
{
    mzc::clientDeviceId id;
};
```

This structure has the following field:

Field	Type	Description
id	**mzc::clientDeviceId**	Unique ID for the set-top box (MAC address, serial number, network port, and so on). It is opaque to the server.

Used by **mzz::ses::GetClientDevice()**, **mzz::sesInfo**

mzz::resource

A structure representing every session resource that is not a circuit. Resources allow applications or servers to associate arbitrary named attributes with a session.

```
struct resource
{
    string key;
    any    value;
}
```

This structure has the following fields:

Field	Type	Description
key	string	String used to identify this resource.
value	any	The session resource represented by this structure.

Used by **mzz::resources**, **mzz::ses::GetResource**, **mzz::ses::AddResource**

mzz::resources

A sequence of **mzz::resource** structures.

```
typedef sequence<resource> resources;
```

Used by **mzz::sesInfo**, **mzz::ses::GetResource()**

mzz::sess

A sequence of **mzz::ses** objects.

```
typedef sequence<ses> sess;
```

Used by Currently unused.

mzz::sesInfo

A structure containing information about a session.

```
struct sesInfo
{
    clientDevice  client;
    mzc::circuits circuits;
    resources     resources;
    mkd::gmtWall  startTime;
};
```

This structure has the following fields:

Field	Type	Description
client	**mzz::clientDevice**	Client device this session is for.
circuits	**mzc::circuits**	Circuits allocated for this session.
resources	**mzz::resources**	Resources allocated for this session.
startTime	**mkd::gmtWall**	Time and date session was established.

Used by **mzz::ses::GetInfo()**

mzz::sesInfos

A sequence of **mzz::sesInfo** structures.

```
typedef sequence<sesInfo> sesInfos;
```

Used by Currently unused.

mzz::session

A structure containing both the object reference for a session and a structure containing information about it.

```
struct session
{
    ses      or;
    sesInfo  info;
};
```

This structure has the following fields:

Field	Type	Description
or	**mzz::ses**	Object reference for this session.
info	**mzz::sesInfo**	Attributes of this session.

Used by **mzz::factory::AllocateSession()**

mzz::sessions

A sequence of **mzz::session** structures.

```
typedef sequence<session> sessions;
```

Used by Currently unused.

ves Data Types

ves::format

An enumerated type that specifies the file format of the content.

```
enum format
{
    formatInvalid,
    formatMpeg1SS,
    formatMpeg2Trans,
    formatRKF
};
```

This enumerated type has the following values:

Value	Description
formatInvalid	An invalid format.
formatMpeg1SS	MPEG audio and video in MPEG-1 system stream.
formatMpeg2Trans	MPEG audio and video in MPEG-2 transport wrap.
formatRKF	Oracle raw key format.

Used by **ves::hdr**, **veswPrepFeed()**

ves::audCmp

A sequence of bytes that specifies the audio compression format. This datatype is not used directly by OVS, but is provided to help clients locate the correct decoder.

```
typedef sequence<octet, 20> audCmp;
```

Used by **ves::hdr**

ves::vidCmp

A sequence of bytes that specifies the video compression format.

```
typedef sequence<octet, 20> vidCmp;
```

Used by **ves::hdr**

ves::time

The **time** structure (together with **ves::timeType**) specifies time. Time may specify a duration for a real-time feed, or it may be used for the timestamps that go into tags.

```
union time switch (timeType)
{
    case timeTypeInvalid:   long long  ves_timeInvalid;
    case timeTypeSMPTE:     SMPTE      ves_timeSMPTE;
    case timeTypeMillisecs: long long  ves_timeMs;
    case timeTypeMpegSCR:   long long  ves_timeSCR;
    case timeTypeMpegPCR:   long long  ves_timePCR;
    case timeTypeBytes:     long long  ves_timeBytes;
};
```

This union has the following fields:

Field	Type	Description
ves_timeInvalid	long long	An invalid time type.
ves_timeSMPTE	**ves::SMPTE**	hh:mm:ss:ff.
ves_timeMs	long long	Millisecs from start.
ves_timeSCR	long long	33-bit SCR.
ves_timePCR	long long	42-bit PCR.
ves_timeBytes	long long	Only valid for durations (and discouraged there).

Used by **ves::tag**, **veswNewFeed()**

ves::timeType

An enumerated type that (together with **ves::time**) specifies time.

```
enum timeType
{
    timeTypeInvalid,
    timeTypeSMPTE,
    timeTypeMillisecs,
    timeTypeMpegSCR,
    timeTypeMpegPCR,
    timeTypeBytes
};
```

This enumerated type has the following values:

Value	Description
timeTypeInvalid	An invalid time type.
timeTypeSMPTE	hh:mm:ss:ff.
timeTypeMillisecs	Millisecs from start.
timeTypeMpegSCR	33-bit SCR.
timeTypeMpegPCR	42-bit PCR.
timeTypeBytes	Only valid for durations (and discouraged there).

Used by **ves::time**

ves::SMPTE

A structure that specifies time in hh:mm:ss:frame format.

```
struct SMPTE
{
    unsigned long   hour;
    octet           minute;
    octet           second;
    unsigned short  frame;
};
```

This structure has the following fields:

Field	Description
hour minute second	Hours, minutes, and seconds.
frame	A sub-second accuracy. For a 10 fps clip, frame 5 would be a 500 milliseconds.

Used by **ves::time**

ves::frame

An enumerated type that specifies the compressed frame type of a tagged frame.

```
enum frame
{
    frameInvalid,
    frameMpegI,
    frameMpegP,
    frameMpegB,
    frameRawKey,
    frameRawBias,
    frameRawAudio
};
```

This enumerated type has the following values:

Value	Description
frameInvalid	An invalid frame type.

Value	Description
frameMpegI	MPEG Intra-frame (I-frame). An I-frame is the closest approximation to a stand-alone frame that MPEG provides. It relies on data from the sequence header, but is otherwise self-describing.
frameMpegP	MPEG Predictive frame (P-frame). This frame is a group of motion vectors that describes how to warp the previous I or P frame into this P frame.
frameMpegB	MPEG Bias frame (B-frame). Usually, there are a few frames between I-frames and P-frames. These frames are interpolated. The B-frame data adjusts the interpolation.
frameRawKey	This is a raw key access point, or blind-seek point. If you jump to this point in the file, the decoder is expected to handle the transition.
frameRawBias	A frame that is delta from the last key frame. This frame is not useful unless the biased key frame is present.
frameRawAudio	Audio doesn't use frames, so these are random access points where playback can begin. For uncompressed audio, you can choose the location and frequency; once per second would give you the seek accuracy of a typical CD player. This tag is only meant for pure audio streams (WAV files); audio in mixed audio/video clips is not tagged. All cueing is done from the video frames.

Used by **ves::tag**

ves::hdr

A structure that specifies the format of the video data. Most of the fields in this structure apply to any sort of video: how many frames per second there are, height and width of the screen in pixels, and so on. If the video is an MPEG video, the additional format-specific information is stored in **ves::m1sHdr** or **ves::m2tHdr**. In the case of raw key frame video, the additional information is stored in **ves::rkfHdr**.

```
struct hdr
{
    vendor          vend;
    format          fmt;
    vidCmp          vid;
    audCmp          aud;
    unsigned short heightInPixels;
    unsigned short widthInPixels;
    long           pelAspectRatio;
    unsigned long  frameRate;
    union hdrcompData switch (format)
    {
        case formatMpeg1SS:     m1sHdr  m1s;
        case formatMpeg2Trans:  m2tHdr  m2t;
        case formatRKF:         rkfHdr  rkf;
    } compData;
};
```

This structure has the following fields:

Field	Type	Description
vend	**ves::vendor**	Specifies the video vendor.
fmt	**ves::format**	Specifies the file format.
vid	**ves::vidCmp**	Specifies the video compression format.
aud	**ves::audCmp**	Specifies the audio compression format.
Client display information		
heightIn Pixels	unsigned short	Height of the client device display, in pixels.
widthIn Pixels	unsigned short	Width of the client device display, in pixels.
pelAspect Ratio	long	Height*10000 / width of *individual* pixels.

Field	Type	Description
frameRate	unsigned long	Frame rate (fps) *1000, for example, 29.97 fps -> 29970.
compData	**ves::m1sHdr**	MPEG-1 specific information.
	ves::m2tHdr	MPEG-2 specific information.
	ves::rkfHdr	RKF-specific information.

Used by **veswNewFeed()**

ves::m1sHdr

MPEG-1 sequence header containing encoding information about the stream.

```
struct m1sHdr
{
    sequence<octet, 4096> seqHdr;
};
```

This structure has the following field:

Field	Type	Description
seqHdr	sequence <octet, 4096>	The sequence header may **not** change during the course of the feed.

Used by **ves::hdr**

ves::m2tHdr

A structure containing encoding information about an MPEG-2 stream.

```
struct m2tHdr
{
    unsigned short  videoPid;
    unsigned short  audioPid;
    unsigned short  clockPid;
    octet           videoStream;
    sequence<octet, 4096> pat;
    sequence<octet, 4096> pmt;
    sequence<octet, 4096> seqHdr;
};
```

This structure has the following fields:

Field	Type	Description
videoPid	unsigned short	Packet ID for the video packet.
audioPid	unsigned short	Packet ID for the audio packet.
clockPid	unsigned short	Packet ID for the PRC (program clock reference) packet.
videoStream	octet	Identifies which video stream is used.
pat	sequence <octet, 4096>	Entire contents of the PAT, which tells the decoder where the PMT (program map table) is. This field may **not** change during the course of the feed.
pmt	sequence <octet, 4096>	Entire contents of the PMT, which may **not** change during the course of the feed.
seqHdr	sequence <octet, 4096>	Includes the start code for the sequence header. The sequence header may **not** change during the course of the feed except for minor variations in the video bit rate.

Used by **ves::hdr**

ves::rkfHdr

A structure containing data that the decoder requires before decompressing.

```
struct rkfHdr
{
    sequence<octet> initData;
};
```

Used by **ves::hdr**

ves::tag

A structure that defines the frame type, size, current offset, and format-specific information of a tagged frame.

```
struct tag
{
    frame          frameType;
    time           timestamp;
    long long      byteOffset;
    unsigned long  videoByteLength;
    union tagCompData switch (format)
    {
        case formatMpeg1SS:    m1sTag m1s;
        case formatMpeg2Trans: m2tTag m2t;
        case formatRKF:        rkfTag rkf;
     } compData;
};
```

This structure has the following fields:

Field	Type	Description
frameType	**ves::frame**	Compressed frame type.
timestamp	**ves::time**	Time of the stream.
byteOffset	long long	Offset of **ves::time** to beginning of content file.
videoByteLength	unsigned long	Length of the frame.
compData	**ves::m1sTag**	MPEG-1 specific information.
	ves::m2tTag	MPEG-2 specific information.
	ves::rkfTag	RKF-specific information.

Used by **ves::tagList**

ves::tagList

A sequence of **ves::tag** structures.

```
typedef sequence<tag, 65536> tagList;
```

Used by Currently used by a function in **ves.idl**.

ves::m1sTag

MPEG-1 specific information about the tagged frame.

```
struct m1sTag
{
    octet videoStreamCode;
};
```

This structure has the following field:

Field	Type	Description
videoStreamCode	octet	The stream id found in the system header for the video frame. The value “1110 XXXX” indicates an ISO-11172-2 video stream.

Used by **ves::tag**

ves::m2tTag

MPEG-2 specific information about the tagged frame.

```
struct m2tTag
{
    octet          continuityCounter;
    unsigned long  leadingZeros;
    unsigned long  trailingZeros;
    unsigned short headerPesLength;
    unsigned short trailerPesLength;
    unsigned short nonVideoPackets;
    unsigned short nullPackets;
};
```

This structure has the following fields:

Field	Type	Description
continuityCounter	octet	Continuation counter from packet header of the first video packet in the frame.
leadingZeros	unsigned long	Offset into the transport packed payload of the picture start code for this frame.
trailingZeros	unsigned long	Number of bytes left in the final transport packet of the frame after the end of the picture data.
headerPesLength	unsigned short	Length of the Program Elementary Stream (PES) header. For most video applications, this field should be zero. If your encoder specifies non-zero PES lengths for video elementary streams, see the documentation on writing to the **vesw** specification for instructions.
trailerPesLength	unsigned short	Length of the PES header. For most video applications, this field should be zero. If your encoder specifies non-zero PES lengths for video elementary streams, see the documentation on writing to the **vesw** specification for instructions.

Field	Type	Description
nonVideoPackets	unsigned short	Packet count for non-video packets in this frame.
nullPackets	unsigned short	Packet count for null packets.

Used by **ves::tag**

ves::rkfTag

RKF-specific information about the tagged frame.

```
struct rkfTag
{
    octet nothing;
};
```

Used by **ves::tag**

ves::vendor

A string indicating the encoding vendor.

```
typedef string<256> vendor;
```

Used by **ves::hdr**

vesw Data Types

veswCtx

Opaque structure that holds the context for a real-time feed. This structure is returned by the **veswNewFeed()** function.

```
typedef struct veswCtx veswCtx;
```

Used by **veswNewFeed()**, **veswPrepFeed()**, **veswSendHdr()**, **veswClose()**, **veswSendData()**, **veswSendTags()**, **veswSendBlob()**, **veswContBlob()**, **veswLastError()**

veswLayer

An enumerated type that specifies either success or at what layer the failure occurred. See the *Oracle Media Net Developer's Guide* for more information on any Media Net failures.

```
typedef enum veswLayer
{
    veswLayerSuccess = 0,
    veswLayerInternal,
    veswLayerNetwork,
    veswLayerObject
} veswLayer;
```

This enumerated type has the following values:

Value	Description
veswLayerInternal	System level failure (**ys**). This is usually caused by a compilation problem.
veswLayerNetwork	Media Net Address Server failure (**mn**). This usually means the OMN_ADDR environment variable is not set or the **mnaddrsrv** daemon is not running.
veswLayerObject	Media Net ORB Daemon failure (**yo**). This usually means the **mnorbsrv** daemon is not running.

Used by **veswInit()**, **veswTerm()**

APPENDIX

C

OVS Exceptions Reference

This appendix describes the exceptions that can be raised or errors that can be returned by the methods in the OVS API.

Exception/Error	Description	Page
MDS exceptions		
mds::fileEx	Object that is thrown when a file exception is raised.	C-4
mds::io	An IO error occurred.	C-4
MTCR Exceptions		
mtcr::authFailed	Content resolver cannot authenticate an asset cookie.	C-5
mtcr::badImpl	No content resolver with the given implementation id can be found.	C-5
mtcr::badName	Name cannot be translated.	C-6
mtcr::noImpl	Resolver implementation cannot be determined from a passed asset cookie.	C-6

Exception/Error	Description	Page
MZA Exceptions		
mza::BadIterator	Detected bad iterator values passed into method.	C-7
mza::BadObject	No persistent data associated with the passed reference.	C-8
mza::BadProhib	Prohibited errors for content.	C-9
mza::BadPosition	Position errors for clips.	C-10
mza::CommunicationFailure	Oracle Media Net error.	C-11
mza::DataConversion	Data conversion error.	C-12
mza::Internal	Unresolvable system error.	C-13
mza::LgCtntClipsFull	No more room for another clip in the logical content.	C-13
mza::NoMemory	Memory allocation failed.	C-14
mza::NoPermission	Insufficient privilege.	C-15
mza::PersistenceError	Error related to the Oracle database occurred.	C-16
mza::XaException	Insufficient privilege.	C-17
MZABI Exceptions		
mzabi::badEventType	Event type could not be converted to its database representation.	C-18
mzabi::badStatus	Status could not be converted to its database representation.	C-19
MZABIN Exceptions		
mzabin::badLoop	Loop type could not be converted to its database representation.	C-20

Exception/Error	Description	Page
MZABIX Exceptions		
mzabix::badEvent	An event an exporter was processing is invalid.	C-20
MZB Exceptions		
mzb::err	Possible return values from **mzb** calls.	C-21
MZC Exceptions		
mzc::cktEx	Raised for circuit problems or for circuit dealings with channels.	C-22
mzc::chnlEx	Exception returned by methods in the channel interface.	C-24
MZS Exceptions		
mzs::client	Indicates what error occurred in a client call.	C-25
mzs::denial	Client was denied access to a requested asset.	C-27
mzs::network	Error occurred in the communications infrastructure between client and server.	C-28
mzs::server	An error occurred in the server complex.	C-29
MZZ Exceptions		
mzz::sesEx	Problem with a session.	C-30

mds Exceptions

From a caller's perspective, the MDS exceptions of note are MDS_EX_FILEEX and MDS_EX_IO. MDS_EX_FILEEX is raised with an accompanying object containing a reason code and the name of the offending volume or file. If the file is a host file and the name is longer than 256 bytes, it will be truncated.

mds::fileEx

Object that is thrown when a file exception is raised. The actual reason is passed in the **fileExReason** structure.

```
exception fileEx
{
    fileExReason    fileExSts;
    char            fileExNm[256];
};
```

This exception has the following fields:

Field	Description
fileExSts	Reason code.
fileExNm	Name of the offending file.

Raised By **mdsCreate()**, **mdsOpen()**, **mdsLock()**, **mdsUnlock()**, **mdsRemove()**, **mdsUnremove()**, **mdsRename()**, **mdsWrite()**, **mdsBlobPrepare()**, **mdsBlobPrepareSeg()**

mds::io

An IO error occurred. Whenever this exception is raised, a message is logged.

```
exception io {};
```

Raised By **mdsInit(), mdsTerm(), mdsClose()**, **mdsTruncClose()**, **mdsRead()**, **mdsWrite()**, **mdsFlush()**, **mdsCopySeg()**, **mdsBlobPrepare()**, **mdsBlobPrepareSeg()**

mtcr Exceptions

mtcr::authFailed

The content resolver refused to resolve an asset cookie because of insufficient authentication.

```
exception authFailed {};
```

Raised By **mtcr::resolve::name()**

mtcr::badImpl

OVS cannot find a content resolver with the given implementation id. Note that anyone catching this exception must call **yoFree()** to free the name.

```
exception badImpl
{
    string implName;
};
```

This exception has the following field:

Field	Description
implName	Name of content resolver.

Raised By **mtcr::resolve::name()**

mtcr::badName

Name cannot be translated. Note that anyone catching this exception must call **yoFree()** to free the name.

```
exception badName
{
    string theName;
};
```

This exception has the following field:

Field	Description
theName	Name of asset cookie.

Raised By **mtcr::resolve::name()**

mtcr::noImpl

Content resolver implementation cannot be determined from a passed **mkd::assetCookie**.

```
exception noImpl {};
```

Raised By Currently unused.

mza Exceptions

mza::BadIterator

Detected bad iterator values passed into method.

```
exception   BadIterator
{
    long    Position;
    long    NumItems;
};
```

This exception has the following fields:

Field	Description
Position	The position in the list for the next group of items to be returned.
NumItems	The maximum number of items to return at one time.

Raised By Most **lst*** methods that return a ***Lst** sequence of datatypes.

mza::BadObject

No persistent data associated with the passed reference.

```
exception BadObject
{
    opstatus  status;
    string    description;
    Object    objRef;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
objRef	Object reference that failed.
errorCode	SQL error code.

Raised By Currently unused.

mza::BadProhib

Identifies prohibited errors for content.

An error was discovered while trying to convert an **mkd::prohib** type to or from its database representation. This is usually the result of a bad **mkd::prohib** value in the database.

```
exception BadProhib
{
    opstatus  status;
    string    description;
    string    prohib;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
prohib	Prohibit flags that could not be resolved.
errorCode	SQL error code.

Raised By **mza::LgCtnt::getAtr()**, **mza::LgCtntMgmt::lstAtrByNm()**, **mza::CtntMgmt::lstAtr()**, **mza::CtntMgmt::lstAtrByNm()**, **mza::CtntMgmt::lstAtrByFileNm()**

mza::BadPosition

Identifies position errors for clips.

An error was discovered while trying to convert an **mkd::pos** type to or from its database representation. This is usually the result of a bad **mkd::pos** value in the database.

```
exception BadPosition
{
    opstatus   status;
    string     description;
    string     position;
    long       errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
position	Position flags that could not be resolved.
errorCode	SQL error code.

Raised By **mza::LgCtnt::getAtr()**, **mza::LgCtnt::lstAtrByNm()**, **mza::LgCtnt::lstAtrClips()**, **mza::LgCtntMgmt::lstAtrByNm()**, **mza::ClipMgmt::lstAtr()**, **mza::ClipMgmt::lstAtrByCtnt()**, **mza::ClipMgmt::lstAtrByNm()**

mza::CommunicationFailure

Oracle Media Net error.

```
exception CommunicationFailure
{
    opstatus  status;
    string    description;
    long      omnErrorNum;
    string    omnErrorMsg;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
omnErrorNum	Media Net error number.
omnErrorMsg	Media Net error message.
errorCode	SQL error code.

Raised By Currently unused.

mza::DataConversion

Error while trying to convert input parameters to database representation. Usually caused by an input string that is either empty, too long, or NULL.

```
exception DataConversion
{
    opstatus  status;
    string    description;
    short     paramPosition;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
paramPosition	Position of parameter within method.
errorCode	SQL error code.

Raised By Most **mza*** methods.

mza::Internal

Some other internal **mza** related error.

```
exception Internal{};
```

Raised By Currently unused.

mza::LgCtntClipsFull

No more room for another clip in the logical content.

```
exception LgCtntClipsFull
{
    opstatus  status;
    string    description;
    long      memorySize;
    long      errorCode;
    string    errorMsg;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
memorySize	Size of memory.
errorCode	SQL error code.
errorMsg	SQL error message.

Raised By Currently unused.

mza::NoMemory

Memory allocation failed (**malloc()** returned NULL).

```
exception   NoMemory
{
    opstatus  status;
    string    description;
    long      memorySize;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
memorySize	Size of memory.
errorCode	SQL error code.

Raised By Currently unused.

mza::NoPermission

Insufficient privilege.

```
exception NoPermission
{
    opstatus  status;
    string    description;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
errorCode	SQL error code.

Raised By Currently unused.

mza::PersistenceError

Error related to the Oracle database occurred.

```
exception   PersistenceError
{
    opstatus  status;
    string    description;
    long      sqlcode;
    string    sqlerrmc;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
sqlcode	SQL error code.
sqlerrmc	SQL error message.
errorCode	Internal error code.

Raised By All **mza*** methods.

mza::XaException

Insufficient privilege.

```
exception XaException
{
    opstatus  status;
    string    description;
    long      errorCode;
};
```

This exception has the following fields:

Field	Description
status	Processing was either complete or incomplete.
description	Additional detail information.
errorCode	SQL error code.

Raised By All methods.

mzabi Exceptions

mzabi::badEventType

Thrown if an **mzabi::eventType** could not be converted to its database representation, or if the database representation could not be converted to a valid event type.

```
exception badEventType
{
    string     msg;
    eventType  event;
};
```

This exception has the following fields:

Field	Description
msg	The database representation of the bad event type.
event	The actual value of the scheduled event.

Raised By Most **get***, **set***, and **lst*** methods that return a ***Lst** sequence of datatypes.

mzabi::badStatus

Thrown if an **mzabi::schdStatus** or **mzabi::expStatus** could not be converted to its database representation, or if the database representation could not be converted to a valid status value.

```
exception badStatus
{
    string  msg;
    sb4     status;
};
```

This exception has the following fields:

Field	Description
msg	The database representation of the bad status.
status	The actual value of the status: either **mzabi::schdStatus** or **mzabi::expStatus**.

Raised By Most **get***, **set***, and **lst*** methods that return a ***Lst** sequence of datatypes.

mzabin Exceptions

mzabin::badLoop

Thrown if an **mzabin::loopType** could not be converted to its database representation, or if the database representation could not be converted to a valid loop type.

```
exception badLoop
{
    string loop;
};
```

This exception has the following field:

Field	Description
loop	The database representation of the bad loop type.

Raised By **mzabin::NvodMgmt::lstAtrByDate()**, **mzabin::NvodMgmt::lstAtrByLCNm()**, **mzabin::NvodMgmt::lstAtrBySchd()**

mzabix Exceptions

mzabix::badEvent

Raised by an exporter if the event it was processing is invalid.

```
exception badEvent {};
```

Raised By **mzabix::exporter::startEvent()**, **mzabix::exporter::stopEvent()**, **mzabix::exporter::changeEvent()**

mzb Exceptions

mzb::err

The **mzb::err** enumeration lists the possible return values from **mzb** calls.

```
enum err
{
    errSucc,
    errBadClient,
    errBufSmall,
    errBadFile,
    errNoMem,
    errBadPhys,
    errRcInt,
    errInfo,
    errLast
};
```

This enumeration has the following values:

Value	Description
errSucc	Call successfully completed.
errBadClient	Unidentifiable client was passed to **mzb**.
errBufSmall	Server could not find a return value due to a small buffer.
errBadFile	Server was passed a bad configuration file name.
errNoMem	Server ran out of memory.
errBadPhys	Malformed physical address was detected.
errRcInt	Internal error during remote call.
errInfo	Placeholder for informational values.
errLast	A token for the last message.

Raised By All **mzb** calls.

mzc Exceptions

mzc::cktEx

Exception raised when setting up a circuit or dealing with its channels improperly, or when attempting operations on a circuit that does not exist.

```
enum circuitException
{
    cktExNotImplemented,
    cktExNoChannelAvailable,
    cktExNoChannelBandwidth,
    cktExBadChannelProtocol,
    cktExBadChannelAddress,
    cktExNoUpstreamAvailable,
    cktExNoUpstreamBandwidth,
    cktExBadUpstreamProtocol,
    cktExBadUpstreamAddress,
    cktExNoDownstreamAvailable,
    cktExNoDownstreamBandwidth,
    cktExBadDownstreamProtocol,
    cktExBadDownstreamAddress,
    cktExNoCircuitAvailable
    cktExInvalidClient,
    cktExInvalidRequest,
    cktExNotControl,
    cktExNotData,
    cktExNotIsochronous,
    cktExInvalidStream,
    cktExStreamAlreadyBound,
    cktExStreamNotBound
};
exception cktEx {circuitException circuitFailType;};
```

This exception returns one of the following values:

Value	Description
cktExNotImplemented	Requested operation not (yet) implemented.
cktExNoChannelAvailable	Could not allocate channel.
cktExNoChannelBandwidth	No channel found that can handle requested bandwidth.
cktExBadChannelProtocol	No channel found that supports the requested protocol.

Value	Description
cktExBadChannelAddress	No channel found that can communicate with the specified address.
cktExNoUpstreamAvailable	Could not allocate upstream channel.
cktExNoUpstreamBandwidth	No upstream found that can handle requested bandwidth.
cktExBadUpstreamProtocol	No upstream found that supports requested protocol.
cktExBadUpstreamAddress	No upstream found that can receive from the specified address.
cktExNoDownstreamAvailable	Could not allocate downstream channel.
cktExNoDownstreamBandwidth	No downstream channel found that can handle requested bandwidth.
cktExBadDownstreamProtocol	No downstream channel found that supports requested protocol.
cktExBadDownstreamAddress	No downstream channel found that can send to the specified address.
cktExNoCircuitAvailable	Could not allocate circuit.
cktExInvalidClient	Client device is not allowed on network.
cktExInvalidRequest	Circuit properties inconsistent with channel properties.
cktExNotControl	Operation requires a control circuit.
cktExNotData	Operation requires a data circuit.
cktExNotIsochronous	Operation requires an isochronous data circuit.
cktExInvalidStream	Operation requires a valid **mzs::stream** object.
cktExStreamAlreadyBound	Stream already bound to circuit.

Value	Description
cktExStreamNotBound	No stream bound to circuit.

Raised By **mzc::factory::BuildCircuit(), mzc::ckt::Rebuild(), mzc::ckt::BindXSM(), mzc::BindStream(), mzz::factory::AllocateSession(), mzz::ses::AddCircuit(), mzz::ses::DelCircuit()**

mzc::chnlEx

Exception raised when allocating a channel improperly or when attempting operations on a channel that does not exist. With the exclusion of **chnlExNotImplemented**, these exceptions may be raised during operations that allocate channels.

```
enum channelException
{
    chnlExNotImplemented,
    chnlExOutOfMemory,
    chnlExOutOfChannels,
    chnlExOutOfBandwidth,
    chnlExBadProtocol,
    chnlExBadAddress
};
exception chnlEx {channelException channelFailType;};
```

This exception returns one of the following values:

Value	Description
chnlExNotImplemented	Requested operation not (yet) implemented.
chnlExOutOfMemory	Channel provider is out of memory.
chnlExOutOfChannels	Channel provider is out of channels.
chnlExOutOfBandwidth	Channel provider does not have sufficient bandwidth to fulfill request.
chnlExBadProtocol	Channel provider does not support the requested protocol.
chnlExBadAddress	Requested address is invalid.

Raised By **mzc::chnl::Rebuild()**

mzs Exceptions

mzs::client

Exception raised in response to an error in a client call. The client should attempt to determine what problem in the application or stored content caused the problem and remedy it.

```
enum clientException
{
    clientBadStreamID,
    clientInvContext,
    clientInvArgs,
    clientFinishBadInst,
    clientBadPosition,
    clientPastStreamEnd,
    clientNoSuchMember,
    clientNoPrepare,
    clientFileNotFound,
    clientNonIndexedStream,
    clientStbIncapable,
    clientStreamIncapable,
    clientPlayFailure,
    clientBadPrate,
    clientNoCallbackSet,
    clientAutoDeallocDone,
    clientNotPlaying,
    clientNotLooping,
    clientUnsupported,
    clientSeqChanged,
};
exception client {clientException clientFailType;};
```

This exception returns one of the following values:

Value	Description
clientBadStreamID	Told to play a stream that does not exist or is invalid.
clientInvContext	Invalid context passed to stream service.
clientInvArgs	An argument was not used properly.
clientFinishBadInst	Instance passed to finish is invalid.
clientBadPosition	Illegal or unsupported position structure (**mkd::pos**).

Value	Description
clientPastStreamEnd	A position beyond the end of the stream was supplied.
clientNoSuchMember	Requested compression format or bit rate did not match stored content.
clientNoPrepare	Must call **prepare()** or **prepareSequence()** before playing.
clientFileNotFound	Missing tag file.
clientNonIndexedStream	Client asked for something other than a tag file.
clientStbIncapable	Client requested a rate control operation that their client device does not support.
clientStreamIncapable	Client requested a rate control operation that is not allowed on their stream.
clientPlayFailure	Error occurred during rate control. Report this server error to Oracle.
clientBadPrate	Client requested a play rate that was not stored on disk and could not be generated on the fly.
clientNoCallbackSet	Client tried to remove a client callback, but had not previously set one.
clientAutoDeallocNone	Normal termination condition during boot: boot finished and the context was automatically deallocated.
clientNotPlaying	Client tried to get current position when not playing.
clientNotLooping	Tried to cancel a loop when not looping.
clientUnsupported	Passed an unsupported option to the stream service.
clientSeqChanged	Tried to prepare a sequence where segments had different properties after passing the **capNoSeqChange** flag.

Raised By **mzs::factory::alloc(), mzs::factory::boot(), mzs::stream::bootCancel(), mzs::stream::bootMore(), mzs::stream::prepare(), mzs::stream::prepareSequence(), mzs::stream::play(), mzs::stream::pause(), mzs::stream::playFwd(), mzs::stream::playRev(), mzs::stream::frameFwd(), mzs::stream::frameRev(), mzs::stream::getPos(), mzs::stream::finish()**

mzs::denial

Client was denied access to a requested asset.

```
enum denialException
{
    denialPrepareRefused,
    denialDsmccBootRefused
};
exception denial {denialException denialFailType;};
```

This exception returns one of the following values:

Value	Description
denialPrepareRefused	Content resolver refused access.
denialDsmccBootRefused	DsmccBoot was refused.

Raised By **mzs::factory::boot(), mzs::stream::bootCancel(), mzs::stream::bootMore(), mzs::stream::prepare(), mzs::stream::prepareSequence()**

mzs::network

An error occurred in the communications infrastructure between the client and the server.

```
enum networkException
{
    networkDown,
    networkSendFailure,
    networkMarshal
};
exception network {networkException networkFailType;};
```

This exception returns one of the following values:

Value	Description
networkDown	Network problem establishing session.
networkSendFailure	Missing server component.
networkMarshal	Error preparing to make a call.

Raised By Currently unused.

mzs::server

An error occurred in the server complex, and was not caused by the client. Report these exceptions to Oracle.

```
enum serverException
{
    serverBadIndex,
    serverStreamIncompatible,
    serverInternal,
    serverOutMem,
    serverPumpError,
    serverOutOfMemory
};
exception server {serverException serverFailType;};
```

This exception returns one of the following values:

Value	Description
serverBadIndex	Error when reading index file.
serverStreamIncompatible	Index file incompatible with code version.
serverInternal	Internal error; call Oracle Developer Support.
serverOutMem	Stream service out of memory.
serverPumpError	Error communicating with **vspump**.
serverOutOfMemory	Stream service out of memory.

Raised By **mzs::factory::alloc()**, **mzs::factory::boot()**, **mzs::stream::bootCancel()**, **mzs::stream::bootMore()**, **mzs::stream::prepare()**, **mzs::stream::prepareSequence()**, **mzs::stream::play()**, **mzs::stream::pause()**, **mzs::stream::playFwd()**, **mzs::stream::playRev()**, **mzs::stream::frameFwd()**, **mzs::stream::frameRev()**, **mzs::stream::getPos()**, **mzs::stream::finish()**

mzz Exceptions

mzz::sesEx

This exception is raised in a few different situations:

- if something goes wrong when allocating a session (sesExNoSessionAvailable and sesExInvalidClient)
- if a resource is requested by a given name and no resource with that name has been associated with a session (sesExNoSuchResource)
- if a request is made to disassociate a circuit from a session and that circuit was never associated with that session (sesExNoSuchCircuit)

```
enum sessionException
{
    sesExNotImplemented,
    sesExNoSessionAvailable,
    sesExInvalidClient,
    sesExNoSuchResource,
    sesExNoSuchCircuit
};
exception sesEx {sessionException sessionFailType;};
```

This exception returns one of the following values:

Value	Description
sesExNotImplemented	Requested operation not (yet) implemented.
sesExNoSessionAvailable	Could not allocate session.
sesExInvalidClient	Client device is not allowed on network.
sesExNoSuchResource	No resource found with given key.
sesExNoSuchCircuit	Circuit is not associated with the session.

Raised By **mzz::factory::AllocateSession()**, **mzz::ses::DelCircuit()**, **mzz::ses::GetResources()**, **mzz::ses::GetResource()**, **mzz::ses::AddResource()**, **mzz::ses::DelResource()**

APPENDIX

D

OVS Stream Events

This appendix describes the events that can be generated by the stream service. See Chapter 6 for details on how to write a Media Net event consumer to capture these events.

Event	Description
evAlloc	Request was made to allocate a stream.
evPrepare	Request was made to prepare a stream.
evSeqPrepare	Request was made to prepare a sequence.
evPlay	Request was made to play a stream.
evFinish	Request was made to finish a stream.
evDealloc	Request was made to deallocate a stream.
evDenial	Prepare authentication request was denied.

evAlloc

Event that indicates an **mzs::factory::alloc()** call.

```
struct evAlloc
{
  mzc::circuit     cliCircuit;
  mzs::capMask     capabilities;
  unsigned long    maxBitrate;
};
```

Field	Type	Description
cliCircuit	**mzc::circuit**	The circuit requesting the stream.
capabilities	**mzs::capMask**	The capabilities of the client requesting the stream.
maxBitrate	unsigned long	The maximum bit rate of the stream.

evPrepare

Event that indicates an **mzs::stream::prepare()** call.

```
struct evPrepare
{
  mkd::assetCookie    cookie;
  Object              authRef;
};
```

Field	Type	Description
cookie	**mkd::assetCookie**	The asset being prepared.
authRef	Object	Object reference to the authorization object (if any) to authorize the client.

evSeqPrepare

Event that indicates an **mzs::stream::prepareSequence()** call.

```
struct evSeqPrepare
{
  mkd::assetCookieList   cookies;
  Object                 authRef;
};
```

Field	Type	Description
cookies	**mkd::assetCookieList**	The assets being prepared.
authRef	Object	Object reference to the authorization object (if any) to authorize the client.

evPlay

Event that indicates an **mzs::stream::play()** call.

```
struct evPlay
  {
    Object            authRef;
    mkd::pos          startPos;
    mkd::pos          endPos;
    long              playRate;
    unsigned long     bitRate;
  };
```

Field	Type	Description
authRef	Object	Object reference to the authorization object (if any) to authorize the client.
startPos	**mkd::pos**	The start position for the play.
endPos	**mkd::pos**	The end position for the play.
playRate	long	The presentation rate for the play.

Field	Type	Description
bitRate	unsigned long	The bit rate for the play.

evFinish

Event that indicates an **mzs::stream::finish()** call.

```
struct evFinish
{
  Object       authRef;
  boolean      loopCancel;
};
```

Field	Type	Description
authRef	Object	Object reference to the authorization object (if any) to authorize the client.
loopCancel	boolean	True if loop is being cancelled, false otherwise.

evDealloc

Event that indicates an **mzs::stream::dealloc()** call.

```
struct evDealloc
{
  mzc::circuit      cliCircuit;
};
```

Field	Type	Description
cliCircuit	**mzc::circuit**	The circuit passed to **mzs::factory::alloc()**.

evDenial

Event that indicates an **mzs::stream::prepare()** or **mzs::stream::prepareSequence()** request was denied.

```
struct evDenial
{
  mkd::assetCookieList   cookies;
  Object                 authRef;
};
```

Field	Type	Description
cookies	**mkd::assetCookieList**	The assets being requested.
authRef	Object	Object reference to the authorization object (if any) to authorize the client.

APPENDIX

E Video Server Metadata

This appendix describes the format of the header and tag information that make up the metadata used by video content encoded using the MPEG-1, MPEG-2, and Raw Key Compression video formats.

Header Fields

This section describes the generic and compression-specific fields that make up the header information stored in the **ves::hdr** structure.

Generic Header Fields

The goal of these fields is to enable a client to determine whether it can or cannot display a given stream. For many encoders, none of these fields will change between sessions. The fields expected to change more between sessions (bit rate, length, and frame rate) are parameters to the call used to create a session.

vendor	A string indicating the encoding vendor.
format	The rules the video server uses to provide seeking and fast-forward / rewind capabilities for video content.

video format	The codec used to compress the video portion of the stream. The video server passes this information to clients so that the clients can make sure to have the appropriate decoder available. (It is usually the 4-character code used in ActiveMovie.)
audio format	The codec used to compress the audio portion of the stream. The video server passes this information to clients so that the clients can make sure to have the appropriate decoder available. (It is usually the 4-character code used in ActiveMovie.)
height in pixels	The target decode height for the picture. If the stream is audio only, this is zero.
width in pixels	The target decode width for the picture. If the stream is audio only, this is zero.
pel aspect ratio	The aspect ratio of an individual pixel. (Do not confuse this with the picture aspect ratio.) Those unfamiliar with MPEG should note that this value is expressed as height / width, instead of the other way around. The value is then multiplied by 10,000 in an effort to avoid the use of a float.
frame rate	Frame rate (fps) *1000, for example, 29.97 fps -> 29970.

Compression-specific Header Fields

The discriminator for the compression-specific header fields is a duplicate of the format field in the generic portion of the header. This information is not made available to the client; rather, it is used by the video server to play the content.

MPEG-1 System Stream Compression Specific Header Fields

The lack of scanning support in the current implementation of the video server for MPEG-1 system streams limits the complexity of the compression-specific fields required to support the format.

sequence header	The elementary video stream's sequence header, including the start code.

MPEG-2 Transport Compression Specific Header Fields

MPEG-2 transport is complex. It contains a lot of stream state information that is expected to remain consistent throughout the course of a stream, and is extremely inhospitable to the types of digital splicing operations that are performed in the video server. (In fact, only recently have nonlinear MPEG editors begin to hit the market.) One result of this complexity is a wide disparity in the decoding capabilities of MPEG clients and the need to test each encoder in a system against each decoder to ensure compatibility.

video PID	The Program Identifier containing the video elementary stream.
audio PID	The Program Identifier containing the audio elementary stream.
clock PID	The Program Identifier containing the Program Clock Reference.
video stream	The elementary stream identifier for the video.
PAT	The Program Association Table in the transport stream.
PMT	The Program Map Table from the transport stream.
sequence header	The video sequence header, including the start code.

Raw Key Compression Specific Header Fields

The simplicity of the raw key format makes for short header and tag fields.

init data	This data will be inserted into the video stream any time a seeking operation occurs, including play from the beginning of the file. **Note:** initialization data is not stored (and therefore not used) in the current version of the video server.

Tag Fields

This section describes the generic and compression-specific fields that make up the tag information stored in the **ves::tag** structure.

Generic Tag Fields

The generic part of the tag includes the frame's byte offset from the beginning of the stream and its type, length, and presentation time stamp. The compression-specific fields include system or stream information, which is contained in the frame, that may need to be modified before delivery to maintain a valid stream during rate control operation.

frame type	The type of frame. Valid values are described by the **ves::frame** data type. The simplest encoding for MPEG would include only I-frames.
timestamp	The presentation time for the frame. Only position types **MpegSCR**, **MpegPCR**, **Time**, and **Millisecs** can be used for this value. Timestamps are expected to increase during the course of the feed, with the possible exception of the Mpeg SCR or PCR wrapping. Time does not need to begin at zero, although it is recommended if the choice is arbitrary. The presentation timestamp is the target display time on the client. For instance, if the content used a constant 10fps display rate, the timestamps would be 0ms, 100ms, 200ms, and so forth.
byte offset	The relative byte offset of the frame from the first byte of the stream (which would be byte 0). Byte offsets are eight-byte fields. While wrapping is not supported, it is hard to imagine a feed going through 16 quintillion bytes with the current generation of hardware.
video byte length	The offset of the last *video* byte in the frame from the first byte of the frame. For example, several MPEG audio packets associated with the frame might follow the last video packet, and **vstag** will explicitly ignore them.

Compression-specific Tag Fields

This section describes the compression-specific information in the **ves::tag** structure.

MPEG-1 System Stream Compression Specific Tag Fields

video stream code	The stream ID in the system header containing the video frame (see ISO / IEC 11172-1 section 2.4.4.2 for table). The first nibble of this byte is always 0xE, indicating an MPEG-1 video stream. Although this field exists here in the tag, it is very unlikely that the value would ever change during the course of a feed.

MPEG-2 Transport Compression Specific Tag Fields

The MPEG-2 transport specific tag information is easily the most complicated. A detailed example follows in the next section. The offset for an MPEG-2 frame points to the first 0x00 of the picture start code. Several of the tag fields refer to this transport packet.

continuity counter	The four-bit continuity counter in the packet containing the first zero of the video start code of the frame. The continuity counter is found in the small nibble of the fourth byte of the transport header.
leading zeros	The offset of the picture start code into the payload of its transport packet. Since the feed must consist only of complete, contiguous transport packets, this value is essentially the byte offset modulo 188.
trailing zeros	Analogous to leading zeros, trailing zeros refers to the number of bytes that follow the last video byte in its transport packet.

header PES length	In section 2.4.3.7 of the MPEG-2 specification, under "PES_packet_length" it states: "... A value of 0 indicates that the PES packet length is neither specified nor bounded and is enabled only in PES packets whose payload is a video elementary stream contained in Transport Stream packets." If you do this, both header and trailer PES length are zero. However, if you encode non-zero PES lengths, the video server will need the header and trailer PES length fields to generate valid PES headers. The "header PES length" is what the size of the PES packet containing the picture start code would be if the first byte of the PES packet was the picture start code, which is exactly what happens when that frame is selected for a seek or scan operation.
trailing PES length	If you recorded a non-zero header PES length, you will also need a trailing PES length. The "trailing PES length" is the number of bytes in the PES packet following the last video byte of the frame.
non-video packets	The number of transport packets between the first and last video packets with a PID other than the video PID. Packets with any PID other than the video PID may be dropped during scanning operations.
null packets	The number of null packets between the first and last video packets.

Raw Key Compression Specific Tag Fields

The stateless nature of the raw key format means that there are no raw key compression-specific tag fields.

Parameters for Creating Feed Sessions

In addition to the header fields, the **veswNewFeed()** function includes parameters that complete the specification of the behavior of the feed.

base name	The root of the file names created on the MDS file system for the feed. File extensions are hard coded based on file type: tag files have the extension **mpi**, and content files use **mpg**, **m2t**, or **rk**, based on their content type. Content files also have a timestamp embedded in their file name. For instance, the directory might look something like:
	`% mdsdir -l /mds/video`
	`Volume /mds/video (rw): 15 matches`
	`  9.7k Dec 18 17:32:27 rw mojo.mpi`
	`  3.4m Dec 18 17:31:20 rw mojo32b89ae8.rk`
	`  3.4m Dec 18 17:31:41 rw mojo32b89afd.rk`
	`  1.7m Dec 18 17:32:01 rw mojo32b89b11.rk`

duration	The **ves::time** structure indicating the expected duration of content (or amount of content to buffer on the server for a continuous real-time feed).
average I/P/B frames per minute	The video server uses this information to establish an appropriate size for the tag file. These values are specified separately since the server may be configured to discard some types of tags independently of the encoder's wishes. Slightly overestimating these numbers is strongly encouraged if your encoder performs automatic I-frame insertion on scene changes. In fact, gross overestimation probably won't have any noticeable effect.

APPENDIX

F

Oracle Generic Framing

The Oracle Generic Framing (OGF) transport header is described by the **mktgen.h** file.

When the OVS video pump streams video data encoded in the MPEG-1 System Streams or Oracle Streaming Format (OSF) format, it divides the stream into a series of chunks and wraps each chunk with an OGF header. Each chunk of video data wrapped with an OGF header is called an **OGF packet**, which is suitable to be sent over the network to the client. The client receives the packets from the network and uses the information in the OGF headers to reorganize the video data back into a video stream.

Note OGF headers are not needed when using the MPEG-2 transport, since it has similar mechanisms already built into its format. (Clients receiving MPEG-2 streams must still reorganize the packets back into a contiguous stream, but they don't need OGF header information.)

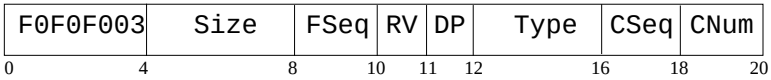
When receiving MPEG-1 and OSF data, the information in the OGF header is used by the client to determine whether any packets have been dropped or have arrived out of order. When receiving a TCP byte stream, the client can use the OGF information to determine where the packets begin and end.

There is a lag between when the client issues a command to the server and when the server processes the command and responds by sending the appropriate data. The OGF header also contains information the client can use to recognize the packets of data that have arrived in response to a previous seek command.

OGF Header Format

The OGF header is a 20-byte value, as shown below.

Note The term “frame” in this context refers to a **network frame**, which is the same thing as a packet.




```
#define MktGenEyeCatch     0xF0F0F003             /* Eye catcher value */

/* Size defines */

#define MktGenHdrLen       20  /* Length of a generic transport header */
#define MktGenFrameLen     8192    /* Length of a Oracle generic frame */
#define MktGenPayloadLen   (MktGenFrameLen-MktGenHdrLen) /* Payload Sz */
#define MktGenDataPid      0x100         /* Pid indicating data frames */

/* Offset defines */

#define MktGenEcOff        0        /* Offset in header for Eyecatcher */
#define MktGenSzOff        4      /* Offset in header for payload size */
#define MktGenFSeqOff      8   /* Offset in header for sequence number */
#define MktGenRevOff       10   /* Offset in header for revision level */
#define MktGenDpOff        11      /* Offset in header for data offset */
#define MktGenTypOff       12       /* Offset in header for frame type */
#define MktGenCSeqOff      16     /* Offset in header for cmd sequence */
#define MktGenCNumOff      18       /* Offset in header for cmd number */

#define MktGenRevision     1                /* Current revision level */

#define MktGenDataPos       MktGenHdrLen   /* Offset of 1st byte of data */

/* Types */

#define MktGenVidTyp        0x0001                    /* Video type */
#define MktGenCtlTyp        0x0002            /* Control Channel type */
#define MktGenDatTyp        0x0100                     /* Data type */
```

Description of OGF Header Fields

F0F0F003	Eye catcher / magic value / line sync code. This value appears at the beginning of every OGF packet.
Size	The actual length of the video data, not including the length of the header. This length may vary from the length of the received transport packet. For example, a client may receive a series of 8192-byte UDP packets for a file. The last UDP packet for the file contains only 5000 bytes of data. The **Size** field allows the client to detect how much valid data is contained in the packet.
FSeq	Sequence number of each packet. The video pump increments this value by 1 for each packet sent. The server resets this value to 1 when a new client connection is established. This field can be used by the client to detect dropped or reordered packets.
RV	Revision level of OGF format. This constant should only change when OGF data fields are removed or resized. This value should never change during the life of a connection.
DP	Start of data offset, or Data Position. This value indicates the offset within the packet at which the first byte of video data is located. (By definition, this value also indicates the length of the OGF header.)
Type	Type of data in the packet (video, control, data, and so on). This value should only indicate video data at this time. Non-video data is currently not supported.
CSeq	Current Sequence in the stream. Transitions from one sequence of packets to another are identified by the server as a **discontinuity** in the data. When a discontinuity occurs, **CSeq** is set to the same value as **FSeq** for the first packet of the new sequence. The **CSeq** value remains constant for all packets in the sequence until the next discontinuity. This field allows the client to identify dropped packets at the beginning of a discontinuity. See *Detecting Discontinuities in the Data* on page F-4 for an example. See *Using the Current Sequence Number* on page 2-23 for an example of how to use this field.

CNum	Command Number. When a discontinuity in the data occurs, the value of this field is increased by 1. The **CNum** will remain constant for all packets until the next discontinuity. This field allows the receiver to definitively detect wholly dropped sets of packets associated with a specific discontinuity. See *Detecting Discontinuities in the Data* for an example. See *Using the Command Number* on page 2-23 for an example of how to use this field.

Detecting Discontinuities in the Data

While the server is delivering OGF packets to the client, a discontinuity in originating content may occur. This may happen because the client requested the server to seek to a different location in the content, or because the current content is composed of a sequence of several pieces of different physical content (see the discussion of **clips** in *Logical Content Model* on page 3-2). Note that in the case of sequence transitions, the discontinuity is not the direct result of a client command, so *clients cannot assume that every discontinuity corresponds to a client request*.

Not all discontinuities occur on the first byte of the packet. It is entirely possible for the actual data change to occur somewhere in the middle of the packet. This may happen in stream sequencing, but not when seeking.

The **CNum** keeps track of the number of discontinuities since the beginning of the content, while the **CSeq** indicates the number of the packet at which the last discontinuity occurred.

Example Table F-1 illustrates how the OSF header fields might detect the transitions between three segments. The first segment consists of three packets, the second segment consists of two packets, and the length of the third segment is undetermined at this point.

	FSeq	**CSeq**	**CNum**
	1	1	1
	2	1	1
	3	1	1
discontinuity ->>	4	4	2
	5	4	2
discontinuity ->>	6	6	3
	7	6	3
	8	6	3

Table F-1: Example of FSeq, CSeq, and CNum Fields

One use of the **CSeq** field is illustrated in Table F-2, where the packet **FSeq** 4 was dropped by the network. By checking the **CSeq** value in **FSeq** 5, you can detect that a discontinuity occurred on **FSeq** 4.

	FSeq	**CSeq**	**CNum**
	1	1	1
	2	1	1
	3	1	1
discontinuity ->>	**5**	**4**	2
discontinuity ->>	6	6	3
	7	6	3
	8	6	3

Table F-2: Detecting a Dropped Packet

Note in Table F-1 that **FSeq** packets 4 and 5 comprise an entire sequence. Should both these packets be dropped as shown in Table F-3, the **CSeq** value could not be used to detect the missing sequence. However, you could detect this missing sequence by noting that the **CNum** value incremented by 2.

	FSeq	**CSeq**	**CNum**
	1	1	1
	2	1	1
	3	1	**1**
discontinuity ->>	6	6	**3**
	7	6	3
	8	6	3

Table F-3: Detecting a Missing Sequence

APPENDIX

G

Setting Up OVS Sample Applications

This appendix describes how to run the OVS sample applications listed in Appendix H. These sample applications illustrate the different ways of using Oracle Video Server components. The source code for most of the sample applications contains comments detailing each step.

The sample applications presented in this appendix assume that:

- you are already familiar with the basics of Oracle Media Net (OMN) and the UNIX operating system (see the *Oracle Media Net Developer's Guide* for more information)
- you have successfully installed OVS on your system (see the *Oracle Video Server Installation Guide* for more information)
- you understand the basic OVS fundamentals and terminology (see Chapters 1, 2, and 3 for more information)

Important These sample applications are only examples. They are not intended to be incorporated directly into your applications.

Initial Setup

The sample applications are located in the **$ORACLE_HOME/vs30/demo** directory created during OVS installation. Take a minute to read the file **README.txt**, which contains descriptions of most of the sample applications. Files and directories used by the sample applications are provided and/or created during the **make** process (see *Building the Sample Applications*). Necessary environment variables are automatically set during OVS installation.

Building the Sample Applications

When OVS is installed, the **demo** directory contains the directories:

```
bin             content          eventdemo        ovsdemo
clipdemo        ctntdemo         include          vesdemo
```

Each of the ***demo** directories (**clipdemo**, **ctntdemo**, and so on) contain files particular to the specific sample application. For example, the **vesdemo** directory contains the files:

```
Makefile        vesdemo.c
```

You can compile the sample applications before or after starting OVS by running **make** in the specific sample application directory. For example, the compiled **vesdemo** directory contains the files:

```
Makefile        vesdemo.c        vesdemo.o        ves.h
```

After running **make** in the specific sample application directory, the **bin** directory contains the actual binary executable files. For example, after running **make** in the **vesdemo** directory, the **bin** directory contains the following files:

```
vesdemo
```

Additionally, if any changes are made in any of the directories, use **make** to update and recompile the affected sample applications.

Installing the Demonstration Database Schema

Important You should **not** run these sample applications while **vscontsrv** is connected to the production database schema; instead, install your own demonstration database schema by running the **vsdbbuild** utility in the **$ORACLE_HOME/bin** directory.

The **vsdbbuild** utility creates the demonstration account in the database; builds empty tables, views, indexes, and sequences; and populates some of the tables with required new data. You must know certain information prior to running **vsdbbuild**, such as your system password, user name and password, and remote connect string. See the *Oracle Video Server Administrator's Guide and Command Reference* for complete information on **vsdbbuild**.

```
% $ORACLE_HOME/bin/vsdbbuild -T -D -s MANAGER OVSDEMO/OVSDEMO@OVS
System Password: MANAGER
User Tablespace: USERS
Temp Tablespace: TEMP
Username/Password: OVSDEMO/OVSDEMO
TNS Alias: OVS
Connecting to OVS as system...Connected
Getting ready to drop user OVSDEMO. OK to proceed (y/n): y
.
.
.
Disconnecting from OVS
```

Setting the username and password to **OVSDEMO/OVSDEMO** ensures that the content service connects to the database as the correct user.

Stop the content server (see *Starting and Stopping OVS*) and add your **vscontsrv.connect** resources.

```
% $ORACLE_HOME/vs30/admin/ovsstop
Deregistering the Oracle Video Server ......
% vi $ORACLE_HOME/vs30/admin/mnrc
vscontsrv.connect=OVSDEMO/OVSDEMO@OVS
```

Restart your content server, connecting to your new demonstration database schema (see *Starting and Stopping OVS*).

```
% $ORACLE_HOME/vs30/admin/ovsstart
starting Media Net ...
OMN_ADDR is UDP:127.0.0.1:5000
starting address server ....... [1] 6624
starting name server .......... [2] 6626
starting object server ........ [3] 6628
starting logger ............... [4] 6630
starting object name server ... [5] 6632
starting event server ......... [6] 6634
starting media data store ..... [1] 6636
starting remote server ........ [1] 6638
starting session manager ...... [1] 6640
starting video pump ........... [2] 6642
starting stream service ....... [2] 6644
starting content service ...... [3] 6646
starting broadcast initiator .. [4] 6648
starting NVOD exporter ........ [5] 6650
starting scheduler ............ [6] 6652
starting MDS FTP Server ....... [7] 6654
```

Lastly, copy a piece of content into your MDS volume (using **mdscopy**) and generate a tag file for the content (using **vstag**).

```
% mdscopy $ORACLE_HOME/vs30/demo/content/ovs_mpg1_1536k.mpg /mds/video
% $ORACLE_HOME/bin/vstag -E mpi /mds/video/ovs_mpg1_1536k.mpg
now tagging /mds/video/ovs_mpg1_1536k.mpg into
  /mds/video/ovs_mpg1_1536k.mpi
file /mds/video/ovs_mpg1_1536k.mpg tagged in 0:39
```

Starting and Stopping OVS

Important If you are currently running Oracle Video Server Manager (VSM), quit VSM before invoking **ovsstart** or **ovsstop**.

To start OVS, invoke the **ovsstart** script. You can invoke **ovsstart** with or without a database (see *Logical Content Services with a Database* on page 3-8 and *Logical Content Services without a Database – Stand-alone Mode* on page 3-9). Sample applications that can run without a database are noted. To stop OVS, use **ovsstop**.

See the *Oracle Video Server Installation Guide* for complete information on **ovsstart** and **ovsstop**.

Important The sample applications in this appendix assume that OVS has been started with the database.

Extra Processes When Exiting OVS

Generally, **ovsstop** will stop processes started by **ovsstart**. In certain cases, however, **ovsstop** may not stop all processes. If, for example, you start a process after **ovsstart**, **ovsstop** may not stop the process.

In the following example, **dcontsrv** and **eclisten** were invoked after **ovsstart** was started, and appear in the list of active processes (see the arrows).

```
%  ps
PID TTY          TIME COMD
24184 pts/5    0:00 mnnmsrv
24210 pts/5    0:01 vsstrmsr
24206 pts/5    1:16 vspump
24474 pts/5    0:00 dcontsrv      <----
24180 pts/5    0:02 mnlogsrv
24218 pts/5    0:02 vsnvodsr
24172 pts/5    0:01 mnrpcnms
24194 pts/5    0:02 mdsdirsr
24168 pts/5    0:01 mnaddrsr
24214 pts/5    0:02 vscontsr
24176 pts/5    0:02 mnorbsrv
24473 pts/5    0:12 eclisten      <----
24202 pts/5    0:01 vscsmsrv
24198 pts/5    0:01 mdsrmtsr
24475 pts/5    0:00 ps
24188 pts/5    0:01 yeced
```

After running **ovsstop**, the **eclisten** and **dcontsrv** processes remain active.

```
% ps
PID      TTY      TIME    COMD
21323    pts/4    0:00    ps
21101    pts/4    1:10    eclisten
21100    pts/4    0:01    dcontsrv
```

It is preferable to stop these processes before calling **ovsstop**, but if that is not possible, use **kill** to remove these processes.

```
% kill -9 21101 21100
%
[2]  + Killed            eclisten
[1]  + Killed            dcontsrv
```

Demonstration Programs

Following is a list of the sample application binaries generated by the **make** operation.

Directory	Name	Description	Details on Page
ovsdemo	**ovsdemo.c**	Simple (non-socket mode) client demonstration that streams video and accesses BLOBs. Complex (socket mode) client demonstration that streams video using network sockets and OGF headers. Can be run with or without a database.	G-7 G-10
clipdemo	**ctntdemo.c**	OVS logical-content binary. Creates / queries / destroys content and content provider objects.	G-14
clipdemo	**clipdemo.c**	OVS clip binary. Creates / destroys logical content, adds / destroys clips, and displays logical content information.	G-16
ctntdemo	**dcontsrv.c** **contclnt.c**	OVS content-resolver-service binary. An example of how to write a content service and resolver for OVS 3.0. OVS content-resolver-client binary. Can be run with or without a database.	G-17
eventdemo	**eclisten.c**	Sample listener for stream event channel.	G-19
vesdemo	**vesdemo.c**	Real-time feed service client.	G-21

You can run these sample applications using the commands exactly as shown in the following sections, with the results shown. You can also supply your own options, if appropriate. There are a number of standard OVS command options that are available to all OVS binaries, see the *Oracle Video Server Administrator's Guide and Command Reference* for more information. You can learn more about each sample application by examining its code in Appendix H.

ovsdemo (non-socket mode)

Syntax `ovsdemo -s [ -b bitrate ] [ -c control-address ] [ -C control-protocol ] [ -d data-address ] [ -D data-protocol ] [ -i clientDeviceId ] blob-name video-tagfile`

-s	Socket mode is active.
-b *bitrate*	Bits per second of content. The default is 2,048,000 bits per second.
-c *control-address*	Address used for the Media Net connection. The default is *local machine IP address*:0.
-C *control-protocol*	Protocol used for the Media Net connection. Values are UDP or TCP. The default is UDP.
-d *data-address*	Downstream address to send the video to. The default is *local machine IP address*:2856.
-D *data-protocol*	Downstream protocol used for video. Values are UDP or TCP. The default is UDP.
-i *clientDeviceId*	String used to uniquely identify the client to the session manager. The default is '**demo**'.
blob-name	Name of the **BLOB** (Binary Large OBject) to transfer from an MDS volume.
video-tagfile	MDS tag file name if **vscontsrv** is running without a database, or logical content name if **vscontsrv** is running with a database.

Command `ovsdemo /mds/video/ovs_mpg1_1536k.mpi ovs_mpg1_1536k`

Interfaces Used **mza::LgCtntMgmt**, **mzs::factory**, **mzs::stream**, **mzz::factory**, and **mzz::ses**

There are two modes of the **ovsdemo** sample application.The non-socket mode version demonstrates a simple way for a client to stream video and access BLOBs. The socket mode version on page G-10 demonstrates a more complex way for a client to stream video using network sockets and OGF headers.

Before the sample application streams the video to the client device, it creates a session (and sets up a control circuit) by:

1. Binding an implementation of the session factory interface and returning an object reference.
2. Allocating a session with OVS. The **mzz::factory::AllocateSession()** method allocates the session and sets up the control circuit.
3. Adding a downstream circuit. The **mzz::ses::AddCircuit()** method allocates a downstream circuit for video and associates the circuit with the session. The **mzs::factory::alloc()** method allocates a stream for the downstream circuit.

After the sample application creates the session, it then:

1. Queries the BLOB interface of the content service to get an asset cookie.
2. Prepares and transfers one BLOB from MDS.
3. Queries the content service for a stream asset to play using the **mza::LgCtntMgmt::lstAtrByNm()** method.

Once the **ovsdemo** sample application obtains the asset cookie for the sample content and an object reference to the stream object, it calls the **mzs::stream::prepare()** method to prepare the stream for delivery. The sample application, which is now ready to stream the video to the client device, finishes its demonstration by:

1. Playing the stream for 10 seconds. The stream service plays the clip as soon as the **prepare** operation is complete, based on the **playNow** flag passed into the **prepare()** method.
2. Pausing once for 3 seconds. You can use either the **mzs::stream::pause()** or the **mzs::stream::play()** method to pause a stream. The **ovsdemo** sample application uses the **play()** method to pause the stream because doing so offers more control over network latencies (see *Pausing Playback* on page 2-18).
3. Resuming and playing again for 5 more seconds.The **mzs::stream::play()** method plays the stream.
4. Stopping the video. The client is finished with the stream. The **mzs::stream::finish()** method “unprepares” the content.

Before the **ovsdemo** sample application exits, a certain amount of cleanup must occur, including deallocating the stream and disconnecting from the server. The **mzs::stream::dealloc()** method deallocates the stream. The **mzz::ses::Release()** method releases the session (shuts down all circuits, frees all resources, and destroys the session).

The following screen shows the results when running **ovsdemo** in non-socket mode; see *ovsdemo (socket mode)* on page G-10 for running in socket mode.

```
% ovsdemo /mds/video/ovs_mpg1_1536k.mpi ovs_mpg1_1536k
allocating session (id demo, addr UDP:127.0.0.1:0) ... allocated.
adding downstream circuit ( addr UDP:127.0.0.1:2856, bitrate 2048000) ... added
initializing BLOB library ... initialized.
querying content service (file /mds/video/ovs_mpg1_1536k.mpi) ... done.
preparing BLOB ... prepared.
transferring BLOB ...
  transfer started ...
  blob alloc() called, offset = 0
  ... transfer complete.
Terminating BLOB library  .... terminated.
allocating stream ... allocated
querying content service (file ovs_mpg1_1536k) ... done.
  title:  <Unavailable>
  length:  43258 msecs
preparing stream ( playing immediately ) ... prepared.
sleeping for 10 seconds
pausing stream ... paused
sleeping for 3 seconds
resuming stream ... resumed
sleeping for 5 seconds
finishing stream ... finished.
deallocating stream ... deallocated.
releasing session ... released.
OVS demo exiting successfully ...
```

ovsdemo (socket mode)

Syntax `ovsdemo -s [ -b bitrate ] [ -c control-address ]`
`[ -C control-protocol ] [ -d data-address ] [ -D data-protocol ]`
`[ -i clientDeviceId ] blob-name video-tagfile`

-s	Socket mode is active.
-b *bitrate*	Bits per second of content. The default is 2,048,000 bits per second.
-c *control-address*	Address used for the Media Net connection. The default is *local machine IP address*:0.
-C *control-protocol*	Protocol used for the Media Net connection. Values are UDP or TCP. The default is UDP.
-d *data-address*	Downstream address to send the video to. The default is *local machine IP address*:2856.
-D *data-protocol*	Downstream protocol used for video. Values are UDP or TCP. The default is UDP.
-i *clientDeviceId*	String used to uniquely identify the client to the session manager. The default is '**demo**'.
blob-name	Name of the **BLOB** (Binary Large OBject) to transfer from an MDS volume.
video-tagfile	MDS tag file name if **vscontsrv** is running without a database, or logical content name if **vscontsrv** is running with a database.

Command `ovsdemo -s /mds/video/ovs_mpg1_1536k.mpi ovs_mpg1_1536k`

Interfaces Used **mza::LgCtntMgmt**, **mzs::factory**, **mzs::stream**, **mzz::factory**, and **mzz::ses**

There are two modes of the **ovsdemo** sample application.The non-socket mode version on page G-7 demonstrates a simple way for a client to stream video and access BLOBs. The socket mode version demonstrates a more complex way for a client to stream video; it expands on the code from the non-socket mode version to demonstrate network sockets and OGF headers (see Chapter 2, Creating a Session and Using the Stream Service). The key difference between the two modes is that the socket mode version of the **ovsdemo** sample application:

- Creates a network socket to receive the video data.
- Receives the video stream in a series of **packets**. For streams consisting of data encoded using MPEG-1 or Oracle Streaming Format (OSF), the video pump divides the video stream into a series of chunks and wraps each chunk with an Oracle Generic Framing (OGF) header.
- Uses the information in the OGF headers to reassemble the video data back into a contiguous stream prior to decoding.

Before the sample application streams the video to the client device, it creates a session (like it did in non-socket mode, with the addition of step 2) by:

1. Binding an implementation of the session factory interface and returning an object reference.
2. Creating a network socket to receive the video data. The network socket address is used to build the circuit.
3. Allocating a session with OVS. The **mzz::factory::AllocateSession()** method allocates the session and sets up the control circuit.
4. Adding a downstream circuit. The **mzz::ses::AddCircuit()** method allocates a downstream circuit for video and associates the circuit with the session. The **mzs::factory::alloc()** method allocates a stream for the downstream circuit.

After the **ovsdemo** sample application creates the session, it then queries the content service using the **mza::LgCtntMgmt::lstAtrByNm()** method to get the asset cookie and segment information corresponding to the specified video tag file.

Once the **ovsdemo** sample application obtains the asset cookie for the sample content and an object reference to the stream object, it calls the **mzs::stream::prepare()** method to prepare the stream for delivery. The sample application, which is now ready to stream the video to the client device, finishes its demonstration by:

1. Reading OGF packets from the network socket, striping off the header information, and decoding the video data back into a contiguous stream.
2. Playing the stream for 10 seconds. The stream service plays the clip as soon as the **prepare** operation is complete, based on the **playNow** flag passed into the **prepare()** method.
3. Rewinding and playing again for 5 more seconds.The **mzs::stream::play()** method plays the stream.
4. Stopping the video. The client is finished with the stream. The **mzs::stream::finish()** method "unprepares" the content.

Before the **ovsdemo** sample application exits, a certain amount of cleanup must occur, including deallocating the stream and disconnecting from the server. The **mzs::stream::dealloc()** method deallocates the stream. The **mzz::ses::Release()** method releases the session (shuts down all circuits, frees all resources, and destroys the session).

The following screen shows the results when running **ovsdemo** in socket mode; see *ovsdemo (non-socket mode)* on page G-7 for running in non-socket mode.

```
% ovsdemo -s /mds/video/ovs_mpg1_1536k.mpi ovs_mpg1_1536k
Opened port UDP:127.0.0.1:37689
allocating session (id demo, addr UDP:127.0.0.1:0) ... allocated.
adding downstream circuit ( addr UDP:127.0.0.1:37689, bitrate 2048000)
  ... added.
allocating stream ... allocated
querying content service (file ovs_mpg1_1536k) ... done.
  title:  <Unavailable>
  length:  31252 msecs
preparing stream ( playing immediately ) ... prepared.
reading data for 10 seconds
OGF: Rev 1  Data length: 8172  Data offset: 20  Type: Video
     FSeq: 1  CSeq: 1  CNum: 1
.
.
.
OGF: Rev 1  Data length: 8172  Data offset: 20  Type: Video
     FSeq: 23  CSeq: 1  CNum: 1
rewinding stream ... rewind command issued
reading more data for 5 seconds
OGF: Rev 1  Data length: 8172  Data offset: 20  Type: Video
     FSeq: 215  CSeq: 1  CNum: 1
.
.
.
Command response received.
OGF: Rev 1  Data length: 8172  Data offset: 20  Type: Video
     FSeq: 222  CSeq: 222  CNum: 2
finishing stream ... finished.
deallocating stream ... deallocated.
releasing session ... released.
OVS demo exiting successfully ...
```

ctntdemo

Syntax `ctntdemo`

Command `ctntdemo`

Interfaces Used **mza::Ctnt**, **mza::CtntFac**, **mza::CtntMgmt**, **mza::CtntPvdr**, **mza::CtntPvdrFac**, and **mza::CtntPvdrMgmt**

The **ctntdemo** sample application demonstrates how to list, update, create, and destroy content and content provider objects (see *Using Logical Content Interfaces* on page 3-10).

The Content Demo Main Menu—consisting of 10 choices, followed by choice number 99 to exit the demo—appears when you enter **ctntdemo**.

```
% ctntdemo
Content Demo Main Menu

1)  List Content Providers
2)  Get Content Provider By Name
3)  Update Content Provider By Name
4)  Create Content Provider
5)  Destroy Content Provider By Name
6)  List Content
7)  List Content By Name
8)  Update Content By Name
9)  Create Content
10) Destroy Content By Name

99) Exit
```

Choices 1 through 5 deal with content providers:

- Choice 1 displays the name and description of each content provider using the **mza::CtntPvdrMgmt::lstAtr()** method.
- Choice 2 displays the name and description of the specified content provider using the **mza::CtntPvdrMgmt::getAtrByNm()** method.
- Choice 3 updates the content provider name and description, then displays the name and description of the updated content provider using the **mza::CtntPvdrMgmt::getAtrByNm()** method.

- Choice 4 creates a new content provider using the **mza::CtntPvdrFac::create()** method, then displays the name and description of the new content provider using the **mza::CtntPvdrMgmt::getAtrByNm()** method.
- Choice 5 destroys the specified content provider using the **mza::CtntPvdr::destroy()** method.

Choices 6 through 10 deal with content:

- Choice 6 displays the name, description, and provider for each piece of content using the **mza::CtntMgmt::lstAtr()** method.
- Choice 7 displays the name, description, and provider for the specified content using the **mza::CtntMgmt::lstAtrByNm()** method.
- Choice 8 updates the name, description, or provider for the specified content, then displays the name, description, and provider for the updated content using the **mza::CtntMgmt::lstAtrByNm()** method.
- Choice 9 creates a new content object using the **mza::CtntFac::create()** method.
- Choice 10 destroys the specified content using the **mza::Ctnt::destroy()** method.

Choice 99 exits **ctntdemo**.

Caution Use care when destroying content and content provider objects. If you destroy them, they are permanently destroyed.

clipdemo

Syntax `clipdemo`

Command `clipdemo`

Interfaces Used **mza::LgCtnt**, **mza::LgCtntFac**, and **mza::LgCtntMgmt**

If you write applications that use logical content, you may want to build **LgCtnt** of your own (see *Using the Clip and Logical Content (LgCtnt) Interfaces* on page 3-19). The **clipdemo** sample application creates logical content from existing clips in the database. Logical-content functions can be used only with a database.

The **clipdemo** sample application demonstrates how to add a commercial clip and movie clip to a piece of logical content, based on the name of the customer. (Two customers, John and Mary, are used in **clipdemo**.)

The Clip Demo Main Menu—consisting of 6 choices, followed by choice number 99 to exit the demo—appears when you enter **clipdemo**.

```
% clipdemo
Clip Demo Main Menu

1) Build Some Logical Content for John
2) Print Logical Content Info for John
3) Destroy Logical Content Info for John
4) Build Some Logical Content for Mary
5) Print Logical Content Info for Mary
6) Destroy Logical Content Info for Mary

99) Exit
```

Choice 1 uses the **mza::LgCtntMgmt::lstAtrByNm()** method to search for existing content for the first sample customer, John. If logical content is found for John, the **mza::LgCtnt::destroy()** method destroys it. The **mza::LgCtntFac::create()** method creates a new logical content. The **mza::LgCtnt::addClip()** and **mza::LgCtnt::addClipByPos()** methods add the clips to the logical content.

```
Enter a Choice: 1
Looking for John's Logical Content to Destroy
Could not find Logical Content: <John's Logical Content>
Creating John's New Logical Content
Created John's New Logical Content
```

After the screen messages, **clipdemo** returns you to the Main Menu.

Choice 2 uses the **mza::LgCtnt::getAtr()** method to display information about the logical content created in Choice 1. The screen message includes the name, description, total time of the logical content, number of clips in the logical content, and information on the clips.

```
Enter a Choice: 2
John's Logical Content

Logical Content Name: John's Logical Content
Logical Content Description: Logical Content created by OVS clip demo
Logical Content Total Time: 0.3 minutes
Logical Content Number of Clips: 2
Clip Info
Clip 0 Name: Commercial Clip for John
Clip 1 Name: Movie Clip for John
```

Choice 3 destroys the logical content using **mza::LgCtnt::destroy()**.

```
Enter a Choice: 3
Destroyed John's Old Logical Content
```

Choices 4 through 6 build, print, and destroy logical content for the other sample customer, Mary. The logical-content interfaces and methods demonstrated are identical; only different commercial and movie clips are used.

Choice 99 exits **clipdemo**. Before exiting, **clipdemo** checks for any logical content that still exists. If, for example, you did not destroy John's logical content, **clipdemo** destroys it before exiting.

```
Enter a Choice: 99
Destroyed John's Old Logical Content
Could not find Logical Content: <Mary's Logical Content>
```

dcontsrv and contclnt

Syntax `dcontsrv [ -f fileName ]`

-f *fileName*	Alternate file for sample content. The default is **metafile1.dat**. The full path name must be specified or the file must exist in the directory the command is issued from.

Syntax `contclnt [ -b bitrate ] [ -c control-address ]`
`[ -C control-protocol ] [ -d data-address ] [ -D data-protocol ]`
`[ -i clientDeviceId ] contentName`

-b *bitrate*	Bits per second of content. The default is 2,048,000 bits per second.
-c *control-address*	Address used for the Media Net connection. The default is *local machine IP address*:0.
-C *control-protocol*	Protocol used for the Media Net connection. Values are UDP or TCP. The default is UDP.
-d *data-address*	Downstream address to send the video to. The default is *local machine IP address*:2856.
-D *data-protocol*	Downstream protocol used for video. Values are UDP or TCP. The default is UDP.
-i *clientDeviceId*	String used to uniquely identify the client to the session manager. The default is '**demo**'.
contentName	Name of the content played. The content name must exist in the specified *fileName*—the repository the content manager uses for description of the sequences to play—or the content manager will not be able to find the content to play.

Command
```
dcontsrv &
contclnt Content1
```

Interfaces Used **mtcr::resolve**, **mzs::factory**, **mzs::stream**, **mzz::factory**, and **mzz::ses**

Any time you implement your own content service, you will need to create a content resolver to match the logical-content information to the actual content in the MDS (see *Writing a Content Service* on page 5-5). The content server (**dcontsrv**) and client (**contclnt**) demonstrate how to write a new content service. The **dcontsrv** sample application is the server that the **contclnt** sample application queries for the content, as well as the resolver the stream service uses to resolve asset cookies. The **contclnt** sample application queries **dcontsrv** to obtain an asset cookie, and then invokes the stream service to allocate the stream and play each resolved segment. The stream and session interfaces and methods demonstrated in **contclnt** are identical to those demonstrated by **ovsdemo** on page G-7.

The **metafile1.dat** file contains the content metadata. It is used to represent the "database" that stores the metadata needed by the stream service to stream the content. If you use the **metafile1.dat** file as the default for **dcontsrv**, you must have an MDS volume called **video** that contains **ovs_mpg1_1536k.mpi** and **ovs_mpg1_2048k.mpi**.

The **dcontsrv** sample application runs in pair with the **contclnt** sample application—start **dcontsrv** in the background, then start **contclnt**.

The command **dcontsrv &** starts the server. A request to stream **Content1** starts the client. Screen messages echo the status.

```
% dcontsrv -f $ORACLE_HOME/vs30/demo/ctntdemo/metafile1.dat &
Content Demo Ready
% contclnt Content1
allocating session (id demo, addr UDP:127.0.0.1:0) ... allocated.
adding downstream circuit ( addr UDP:127.0.0.1:2856, bitrate 2048000)
  ... added.
allocating stream ... allocated
Resolved Segment 0
Resolved Segment 1
Resolved Segment 2
preparing stream ( playing immediately ) ...
(cookie=dcontsrv:Content1)prepared.
sleeping for 10 seconds
pausing stream ... paused
sleeping for 3 seconds
resuming stream ... resumed
sleeping for 5 seconds
finishing stream ... finished.
deallocating stream ... deallocated.
releasing session ... released.
Content demo exiting successfully ...
```

eclisten

Syntax `eclisten`

Command `eclisten`

Interfaces Used **mzs::ec**

The **eclisten** sample application is a sample listener for the stream event channel.

To listen to the stream event channel, **eclisten** establishes a dialogue between a supplier that pushes the events and a consumer that receives the events (see Chapter 6, Capturing Stream Events). When an event comes into the stream event channel, it is passed to the listener. The **eclisten** sample application

implements the **sendEvent()** method to accept and print the stream-related events as they appear, but a more sophisticated implementation could do a variety of things. Since its purpose is to listen for events from the stream service, **eclisten** must be run in conjunction with another application.

The following screen shows the results when running **ovsdemo (non-socket mode)** with **eclisten** to capture the events generated each time a request is made to the stream service. The shaded lines are provided by **eclisten**. In other words, if **ovsdemo (non-socket mode)** was run alone, this information would not be shown.

```
% eclisten &
Sample Listener: started...
% ovsdemo /mds/video/ovs_mpg1_1536k.mpi ovs_mpg1_1536k
allocating session (id demo, addr UDP:127.0.0.1:0) ... allocated.
adding downstream circuit ( addr UDP:127.0.0.1:2856, bitrate 2048000) ... added
initializing BLOB library ... initialized.
querying content service (file /mds/video/ovs_mpg1_1536k.mpi) ... done.
preparing BLOB ... prepared.
transferring BLOB ...
  transfer started ...
  blob alloc() called, offset = 0
  ... transfer complete.
Terminating BLOB library  .... terminated.
allocating stream ... allocated
Alloc: downstream=UDP:127.0.0.1:2856 capMask=20531 maxBitrate=2048000
querying content service (file ovs_mpg1_1536k) ... done.
  title:  <Unavailable>
  length:  43258 msecs
preparing stream ( playing immediately ) ... prepared.
sleeping for 10 seconds
Play: startPos=beginning endPos=end playRate=1000 bitRate=1536000
SeqPrepare: cookies=vscontsrv:ovs_mpg1_1536k
Prepare: cookie=vscontsrv:ovs_mpg1_1536k
pausing stream ... paused
sleeping for 3 seconds
Play: startPos=current endPos=current playRate=0 bitRate=1536000
resuming stream ... resumed
sleeping for 5 seconds
Play: startPos=current endPos=end playRate=1000 bitRate=1536000
finishing stream ... finished.
Finish: loopCancel=0
Dealloc: downstream=UDP:127.0.0.1:2856
deallocating stream ... deallocated.
releasing session ... released.
OVS demo exiting successfully ...
```

vesdemo

Syntax `vesdemo [-d duration] [-l length] [-v MDS_volume] input_video_file`

-d *duration*	The expected duration of the simulated feed, in seconds.
-l *length*	The length of content to buffer on the video server, in seconds.
-v *MDS_volume*	The location of the output feed, specified as MDS volume name (obtained by **mdsdir**). The default is **/mds/video**.
input_video_file	The location of the input video file, directory plus filename, such as **/tmp/oracle.osf**.

Command `vesdemo -d 600 -l 300 -v /mds/video /tmp/oracle.osf`

Interfaces Used **vesw — Real-time Feed Functions**

The **vesdemo** sample application demonstrates how to write a real-time encoder using the **vesw** interfaces (see Chapter 7, Extending Video Encoders for Real-time Feeds). It delivers content metadata and encoded video data to the real-time feed service, **vsfeedsrv**, so that the video is stored and streamed in real-time to OVS clients. To load content onto the server, the sample feed client:

1. Initializes the Media Net environment using the **veswInit()** method.
2. Creates a new feed session with the video server using the **veswNewFeed()** method.
3. Sends encoded content to the server using the **veswSendData()** method.
4. Sends information about tagged frames to the server using the **veswSendTags()** method.
5. Prevents the Media Net connection with the server from timing out using the **veswIdle()** method.
6. Shuts down the feed and releases the related resources using the **veswClose()** method.
7. Shuts down the Media Net environment and terminates the Media Net session using the **veswTerm()** method.

Ensure that the real-time feed service, **vsfeedsrv**, is running (using the **mnorbls** command). The following screen shows the results when running **vesdemo**.

```
% vesdemo -d 600 -l 300 -v /mds/video /tmp/oracle.osf
Queuing input file /tmp/oracle.osf
Creating session /mds/video/oracle for 300 seconds
Simulation started at Oct 31 19:51:38 1997 and will run for 600
seconds
[Oct 31 19:51:47] oracle -    7 tags and 528509 bytes sent
[Oct 31 19:51:58] oracle -   14 tags and 1064097 bytes sent
[Oct 31 19:52:08] oracle -   21 tags and 1576124 bytes sent
[Oct 31 19:52:18] oracle -   28 tags and 2112473 bytes sent
[Oct 31 19:52:28] oracle -   34 tags and 2630199 bytes sent
.
.
.
[Oct 31 19:53:09] oracle -   61 tags and 4697415 bytes sent
[Oct 31 19:53:19] oracle -   68 tags and 5211101 bytes sent
[Oct 31 19:53:29] oracle -   75 tags and 5723151 bytes sent
[Oct 31 19:53:39] oracle -   82 tags and 6241306 bytes sent
Closing feed : oracle
```

Notice above that **vesdemo** delays awhile to maintain the average bit rate, enabling the TCP/IP connection to push data through the network. A real program would yield to the other threads that are preparing the content and tag data.

The **vesdemo** sample application creates a feed session with basename **/mds/video/oracle**. This feed session runs for 10 minutes (600 seconds). The video server keeps the latest 5 minutes (300 seconds) on the MDS volume. After the feed session is successfully started, the output of the real-time feed can be accessed by the Oracle Video Client. See the *Oracle Video Client Developer's Guide* for more information.

The **vesdemo** sample application creates two types of files in the **video** MDS volume—an **.mpi** file (**oracle.mpi**) and an **.rk** file (**oracle*timestamp*.rk)**. There will be one **.mpi** file and, depending on how long the **vesdemo** sample application is allowed to continue, at least one **.rk** file. List these files using the **mdsdir** command.

```
% mdsdir
Volume /mds/video (rw): 3 matches
oracle.mpi       oracle345a36c.rk       oracle345a36cb.rk
```

Note You may have to stop **vsfeedsrv** manually (see *Extra Processes When Exiting OVS* on page G-5) with the **kill** command, or by using the Oracle Media Net ORB Administration Utility (see the *Oracle Media Net Developer's Guide*):

```
mnorbadm -n vsfeedsrv -s down
```

APPENDIX

H

Example Code Reference

This appendix lists the source code of the following OVS files.

File Name	Description	Listing on Page
ovsdemo.c	Client demonstration that streams video.	H-2
ctntdemo.c	OVS logical-content binary.	H-17
clipdemo.c	OVS clip binary.	H-31
dcontsrv.c	OVS content-resolver-service binary.	H-43
contclnt.c	OVS content-resolver-client binary.	H-47
eclisten.c	Sample listener for stream event channel.	H-53
vesdemo.c	Real-time feed service client.	H-64
cont.idl	Description of content-query interface.	H-75
contImpl.c	Implementation of content-query interface.	H-78
rslvImpl.c	Implementation of content-resolver interface (**mtcr.idl**).	H-84
metafile1.dat	File containing the content metadata.	H-87

ovsdemo.c

```
/* Copyright (c) 1996 by Oracle Corporation. All Rights Reserved.     *
 * ovsdemo.c - OVS Demo binary
 *
 */

#ifdef unix
#ifndef ORASYS_TYPES
#define ORASYS_TYPES
#include <sys/types.h>
#endif
#ifndef ORASYS_SOCKET
#define ORASYS_SOCKET
#include <sys/socket.h>
#endif
#ifndef ORANETINET_IN
#define ORANETINET_IN
#include <netinet/in.h>
#endif
#ifndef ORANETDB
#define ORANETDB
#include <netdb.h>
#endif
#endif

#ifndef SYSX_ORACLE
#include <sysx.h>                                          /* Oracle defs */
#endif
#ifndef SYSXCD_ORACLE
#include <sysxcd.h>                               /* unaligned data get/put */
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h>                                 /* filesystem defs */
#endif
#ifndef SYSB8_ORACLE
#include <sysb8.h>                                /* eight-byte types */
#endif
#ifndef YS_ORACLE
#include <ys.h>                                  /* System layer defs */
#endif
#ifndef YSR_ORACLE
#include <ysr.h>                                  /* resource db defs */
#endif
#ifndef MN_ORACLE
#include <mn.h>                                     /* Media Net defs */
#endif
#ifndef MTUX_ORACLE
#include <mtux.h>                                   /* Media Net Init */
#endif
#ifndef YO_ORACLE
#include <yo.h>                                          /* orb defs */
#endif
#ifndef MDSBLOB_ORACLE
#include <mdsblob.h>                                    /* BLOB defs */
#endif
#ifndef MZCCH_ORACLE
#include <mzcch.h>                                /* channel interface */
#endif
#ifndef MZC_ORACLE
#include <mzc.h>                                  /* circuit interface */
#endif
#ifndef MZZ_ORACLE
#include <mzz.h>                                  /* session interface */
#endif
#ifndef MKD_ORACLE
```

```
#include <mkd.h>                                          /* common datatypes */
#endif
#ifndef MKDC_ORACLE
#include <mkdc.h>                                       /* C constants for mkd */
#endif
#ifndef MZALGCTN_IDL
#include <mzalgctn.h>                                    /* content interface */
#endif
#ifndef MZS_ORACLE
#include <mzs.h>                                          /* stream interface */
#endif

/* FORWARD DEFINITIONS AND FUNCTION PROTOTYPES */

static struct ysargmap ovsdemoArgs[] =
{
  { 'b', "ovsdemo.bitrate", 1 },
  { 'c', "ovsdemo.control-address", 1 },
  { 'C', "ovsdemo.control-protocol", 1 },
  { 'd', "ovsdemo.data-address", 1 },
  { 'D', "ovsdemo.data-protocol", 1 },
  { 'i', "ovsdemo.clientDeviceId", 1 },
  { 's', "ovsdemo.socket-mode=true",0 },
  { YSARG_PARAM, (char *)"ovsdemo.blob-name", 1 },
  { YSARG_PARAM, (char *)"ovsdemo.video-tagfile", 1 },
  { 0, "", 0 }
};

/*
 * Per-blob transfer context
 */
struct blobcx
{
  size_t segSz;                                          /* blob segment size */
  ub4    total;                                          /* total blob length */
  int    numSegs;                                  /* number of blob segments */
  ub1    **segs;                                    /* array of blob segments */
};
typedef struct blobcx blobcx;

static void ovsdemo(const char *id, const char *cproto, const char *ctrl,
                    const char *dproto, const char *data, ub4 bitrate,
                    const char *blobname, const char *tagfile);

static void ovsdemoSession(yoenv *env, const char *id,
               const char *cproto, const char *ctrl,
               const char *dproto, const char *data,
               ub4 bitrate,
               mzz_session *ses, mzc_circuit *dcirc);

static boolean ovsdemoQuery( yoenv *env, const char *file,
               boolean tagfile, mkd_assetCookie *cookie,
               ub4 *bitrate);

static void ovsdemoBlob(  yoenv *env,
        const char *blobname );
static ub1 *ovsdemoBlobAlloc( dvoid *usrp, size_t segSz, ub4 off, ub4 total,
               size_t *actSz );

static void ovsdemoStream( mzc_circuit *circ, yoenv *env,
               const char *tagfile, ub4 bitrate);

#ifdef unix
static void ovsdemoSocket( const char *id, const char *cproto,
                           const char *ctrl, const char *dproto, ub4 bitrate,
                           const char *blobname, const char *tagfile );
static int ovsdemoSetupDataPort(const char *dproto,char *data,boolean *dgram);
static int ovsdemoConnectDataPort(int port);
```

```
static void ovsdemoReadDataPort(int port,boolean dgram,sysb8 *dur,ub2 *fs,
                                ub2 *cs,ub2 *cn);
#endif

void main(int argc, char **argv)
{
  ub1  ctxbuf[SYSX_OSDPTR_SIZE];             /* global context storage space */
  char base[SYSFP_MAX_PATHLEN];
  char  *arg;
  char  *id;
  char  *cproto;
  char  *ctrl;
  char  *dproto;
  char  *data;
  char  *blobname;
  char  *tagfile;
  ub4   bitrate;

  sword  err;

  /* get basename of program for ys */
  sysfpExtractBase( base, argv[0] );

  if (mtuxInit(ctxbuf, base, (mnLogger)0,
              (sword)(argc - 1), argv + 1, ovsdemoArgs) != mtuxLayerSuccess)
    exit( 1 );

  /* maximum bitrate that the client can support */
  arg = ysResGetLast( "ovsdemo.bitrate" );
  if (!arg)
    arg = (char *)"2048000";                                    /* 2 Mbps */
  bitrate = (ub4) atoi( arg );

  /* the client device id must be unique across all sessions in the system */
  id = ysResGetLast( "ovsdemo.clientDeviceId" );
  if (!id)
    id = (char *)"demo";

  /*
   * control protocol and address of the control (Media Net) circuit.
   * it isn't really used for anything.
   */
  cproto = ysResGetLast( "ovsdemo.control-protocol" );
  if (!cproto)
    cproto = (char *)"UDP";
  ctrl = ysResGetLast( "ovsdemo.control-address" );
  if (!ctrl)
    ctrl = (char *)"127.0.0.1:0";

  /* data protocol */
  dproto = ysResGetLast( "ovsdemo.data-protocol" );
  if (!dproto)
    dproto = (char *)"UDP";
  data = ysResGetLast( "ovsdemo.data-address" );
  if (!data)
    data = (char *)"127.0.0.1:2856";

  /* name of the BLOB to transfer */
  blobname  = ysResGetLast( "ovsdemo.blob-name" );

  /* name of the tag file in MDS to play */
  tagfile = ysResGetLast( "ovsdemo.video-tagfile" );

  yseTry
  {
#ifdef unix
    if(ysResGetBool("ovsdemo.socket-mode"))
      ovsdemoSocket(id, cproto, ctrl, dproto, bitrate, blobname, tagfile);
```

```
    else
#endif
      ovsdemo( id, cproto, ctrl, dproto, data, bitrate, blobname, tagfile );
  }
  yseCatchAll
  {
    yslPrint( "exception thrown: %s -- exiting\n", yseExid );
  }
  yseEnd;

  mtuxTerm(ctxbuf);
  exit(0);
}

/* ovsdemo - main routine */
static void ovsdemo( const char *id, const char *cproto, const char *ctrl,
           const char *dproto, const char *data,
           ub4 bitrate, const char *blobname, const char *tagfile )
{
  yoenv       env;                                  /* active ORB environment */
  mzz_session ses;                                          /* session object */
  mzc_circuit dcirc;                            /* downstream circuit object */

  /* initialize ORB environment */
  yoEnvInit( &env );


  /*
   * Estable session with Oracle Video Server and acquire a
   * high-speed data circuit for video
   */
  ovsdemoSession( &env, id, cproto, ctrl, dproto, data, bitrate, &ses, &dcirc);

  /*
   * Example BLOB transfer
   */
  ovsdemoBlob( &env, blobname );

  /*
   * Example Video Stream operations
   */
  ovsdemoStream( &dcirc, &env, tagfile, bitrate);

  /*
   * Clean up session resources and exit
   */
  yslPrint( "releasing session ... " );
  mzz_ses_Release( ses.or, &env );
  yslPrint( "released.\n" );


  mzz_session__free( &ses, yoFree );
  mzc_circuit__free( &dcirc, yoFree );

  yoEnvFree( &env );

  /* the end */
  yslPrint( "OVS demo exiting successfully ...\n" );
  return;
}

/*
 * ovsdemoSession - set up session and high-speed downstream circuit
 *
 * env      is the active environment (INOUT)
 * id       is the client device ID to use (IN)
 * cproto   is the control circuit network protocol (IN)
 * ctrl     is the control circuit network address (IN)
```

```
 * dproto   is the data circuit network protocol (IN)
 * data     is the data circuit network address (IN)
 * bitrate  is the bitrate for the data circuit (IN)
 * ses      is the returned session object (OUT)
 * dcirc    is the returned data circuit object (OUT)
 */
static void ovsdemoSession( yoenv *env, const char *id,
               const char *cproto, const char *ctrl,
               const char *dproto, const char *data,
               ub4 bitrate,
               mzz_session *ses, mzc_circuit *dcirc)
{
  mzz_factory zfac;
  mzc_clientDeviceId cid;
  mzc_cktspec  spec;
  mzc_commProperty props;

  /* bind to session factory */
  zfac = (mzz_factory) yoBind( mzz_factory__id, (char *)0, (yoRefData *)0,
              (char *)0 );

  /*
   * set client device id - must be unique across the system.
   * It is convenient, but not necessary (??) to make the device id be
   * a user-printable string
   */
  cid._length = cid._maximum = (ub4) (strlen(id)+1);
  cid._buffer = (ub1 *)id;

  /*
   * build control circuit for session - a :persistant" point-to-point
   * bidirectional channel (no bitrate because it isn't used for anything).
   */
  props = mzc_propDown | mzc_propUp | mzc_propPointcast | mzc_propControl |
    mzc_propPersistantConnect;
  spec._d = mzc_cktspecTypeRequest;
  spec._u.req._d = mzc_cktreqTypeSymmetric;
  spec._u.req._u.sym.props = props;
  spec._u.req._u.sym.chnl._d = mzc_chnlspecTypeRequest;
  spec._u.req._u.sym.chnl._u.req.props = props;
  spec._u.req._u.sym.chnl._u.req.protocol.name = (char *)cproto;
  spec._u.req._u.sym.chnl._u.req.protocol.info = (char *)ctrl;
  spec._u.req._u.sym.chnl._u.req.bitrate = 0;


  /* invoke call - nothing special about this session  */
  yslPrint( "allocating session (id %s, addr %s:%s) ... ", id, cproto, ctrl );
  *ses = mzz_factory_AllocateSession(zfac, env, mzz_sessNull, &cid, &spec );
  yslPrint( "allocated.\n" );

  /* release mzz factory object - no longer needed */
  yoRelease( (dvoid *)zfac );

  /*
   * build data circuit - asymmetric "persistant" point-to-point
   * downstream-only real time channel from the video pump to the client
   */
  props = mzc_propDown | mzc_propPointcast | mzc_propData |
    mzc_propIsochronousData | mzc_propPersistantConnect;
  spec._d = mzc_cktspecTypeRequest;
  spec._u.req._d = mzc_cktreqTypeAsymmetric;
  spec._u.req._u.asym.props = props;
  spec._u.req._u.asym.upchnl._d = mzc_chnlspecTypeNone;
  spec._u.req._u.asym.upchnl._u.none = 0;
  spec._u.req._u.asym.downchnl._d = mzc_chnlspecTypeRequest;
  spec._u.req._u.asym.downchnl._u.req.props = props;
  spec._u.req._u.asym.downchnl._u.req.protocol.name = (char *)dproto;
  spec._u.req._u.asym.downchnl._u.req.protocol.info = (char *)data;
```

```
  spec._u.req._u.asym.downchnl._u.req.bitrate = bitrate;

  /* invoke call */
  yslPrint( "adding downstream circuit ( addr %s:%s, bitrate %d) ... ",
       dproto, data, bitrate );
  *dcirc = mzz_ses_AddCircuit( ses->or, env, &spec );
  yslPrint( "added.\n" );
}


/*
 * ovsdemoQuery - query the content service to get an asset cookie
 *
 * env      is the active environment (INOUT)
 * file     is the name of the requested file (IN)
 * tagfile  is this for a tagfile or some other kind of content (IN)
 * cookie   is the returned asset cookie (OUT)
 * bitrate  is the bitrate of the returned cookie (if applicable) (OUT)
 *
 * return TRUE if query succeeded and matched one file; FALSE otherwise
 */
static boolean ovsdemoQuery( yoenv *env, const char *file,
               boolean tagfile, mkd_assetCookie *cookie,
               ub4 *bitrate )
{
  mza_LgCtntAtrLst listings;
  mza_Itr itr;
  boolean err = FALSE;
  int i;

  itr.Position = 0;
  itr.NumItems = 1;

  yslPrint( "querying content service (file %s) ... ", file );
  yseTry
  {
     if(tagfile)
     {
        /*
         * bind content service object
         */
        mza_LgCtntMgmt  fobj;           /* content service obj ref */
        fobj = (mza_LgCtntMgmt) yoBind( mza_LgCtntMgmt__id, (char *)0,
            (yoRefData *)0, (char *)0);

        listings = mza_LgCtntMgmt_lstAtrByNm( fobj, env, (char *)file, TRUE,
                        &itr );
        yoRelease((dvoid*)fobj);
     }
     else
     {
        /*
         * bind content service object
         */
        mza_BlobMgmt  fobj;           /* content service obj ref */
        fobj = (mza_BlobMgmt) yoBind( mza_BlobMgmt__id, (char *)0,
            (yoRefData *)0, (char *)0);

        listings = mza_BlobMgmt_lstAtrByNm( fobj, env, (char *)file,
                        &itr );
        yoRelease((dvoid*)fobj);
      }

  }
  yseCatchAll
  {
      yslPrint("Caught Exception from mza query: <%s>",
        ysidToStr(yseExid));
```

```
      err = TRUE;
  }
  yseEnd;

  /* check if here was an error since we can't return from a try block */
  if (err)
    return FALSE;
  yslPrint("done.\n");

  /* check to make sure we only retrieved one match */
  if (listings._length == 0)
  {
    yslPrint( "  ERROR: file not found -- exiting\n" );
    return FALSE;
  }
  /* This should never happen since we only asked for 1 */
  if (listings._length > 1)
  {
    yslPrint("  ERROR: multiple listings returned (unsupported) -- exiting\n");
    return FALSE;
  }
  /* This should never happen since we only asked for 1 */
  if (listings._buffer[0].numClips == 0 )
  {
    yslPrint("  ERROR: No clips for content\n");
    return FALSE;
  }

  if(tagfile)
  {
     yslPrint( "  title:  %s\n", listings._buffer[0].desc );
     yslPrint( "  length:  %d msecs\n",listings._buffer[0].msecs );
  }
  *cookie = (mkd_assetCookie) ysStrDup( listings._buffer[0].cookie );
  /*
   * Get the maximum bitrate
   */
  *bitrate = listings._buffer[0].maxRate;

  mza_LgCtntAtrLst__free( &listings, yoFree );
  return TRUE;
}

/* ovsdemoBlob - code to transfer a single blob */
static void ovsdemoBlob( yoenv *env,
    const char *blobname )
{
  mdsBlob        *blob;
  blobcx          bcx;
  ysevt          *sem;
  int             i;
  boolean         sts;
  mkd_assetCookie cookie;
  ub4             dummy;
  /* initialize BLOB library */
  yslPrint( "initializing BLOB library ... " );
  mdsBlobInit();
  yslPrint( "initialized.\n" );

  /* create semaphore that will be triggered when transfer completes */
  sem = ysSemCreate( (dvoid *)0 );

  /* clear BLOB transfer state */
  bcx.segSz = (size_t)0;
  bcx.total = (ub4)0;
  bcx.numSegs = 0;
  bcx.segs = (ub1 **)0;
```

```
  /* Query the blob interface of the content service to get an asset cookie */

  sts = ovsdemoQuery( env, blobname, FALSE, &cookie, &dummy );
  if (sts == FALSE)
    goto doneblob;

  /* prepare the BLOB */
  yslPrint( "preparing BLOB ... " );
  blob = mdsBlobPrepare( cookie, (CORBA_Object)0, (ub4)0, ovsdemoBlobAlloc,
            (dvoid *)&bcx, sem );
  yslPrint( "prepared.\n" );

  /* transfer the BLOB */
  yslPrint( "transferring BLOB ... \n" );
  mdsBlobTransfer( &blob, 1, (ub4)0);
  yslPrint( "  transfer started ... \n" );

  /* wait for operation to complete (exceptions will be raised synchronously */
  ysSemSynch( sem, (dvoid *)0 );
  yslPrint( "  ... transfer complete.\n" );

doneblob:
  /* terminate BLOB library */
  yslPrint( "Terminating BLOB libray  .... " );
  mdsBlobTerm();
  yslPrint( "terminated.\n" );

  /* free BLOB */
  for (i=0 ; i < bcx.numSegs ; i++)
    ysmGlbFree( (dvoid *)bcx.segs[i] );
}

/*
 * ovsdemoBlobAlloc - callback routine to allocate memory for
 * blob on an as-needed basis.
 *
 * This particular callback routine allocates the memory in 32Kbyte chunks
 * all at once (i.e., during the first invocation)
 */
static ub1 *ovsdemoBlobAlloc( dvoid *usrp, size_t segSz, ub4 off, ub4 total,
              size_t *actSz )
{
  blobcx *bcx = (blobcx *)usrp;
  sword   index;
  size_t  bufOff;

  yslPrint( "  blob alloc() called, offset = %u\n", off );

  if (!off)
  {
    /* Initialize blob state structure */
    bcx->segSz   = 32768;
    bcx->total   = total;
    bcx->numSegs = (total + bcx->segSz - 1) / bcx->segSz;
    bcx->segs    = (ub1 **)ysmGlbAlloc( sizeof(ub1 *) * bcx->numSegs,
                   "buffer array" );
    memset( (dvoid *)bcx->segs, 0, sizeof(ub1 *) * bcx->numSegs );
  }

  index = off / bcx->segSz;
  if (!bcx->segs[index])
    bcx->segs[index] = (ub1 *)ysmGlbAlloc( bcx->segSz, "buffer" );

  bufOff = off % bcx->segSz;
  *actSz = bcx->segSz - bufOff;
  return ( bcx->segs[index] + bufOff );
}
```

```
/*
 * ovsdemoStream - demonstrate stream service calls
 */
static void ovsdemoStream( mzc_circuit *circ, yoenv *env,
              const char *tagfile, ub4 bitrate)
{
  mkd_assetCookie cookie;
  ub4 asset_bitrate;
  boolean sts;

  mzs_factory sfac;
  mzs_stream strm;
  mzs_stream_instance inst;
  mzs_capMask caps;
  mkd_segInfoList status;

  sysb8 tm;

  /*
   * allocate a stream - specify the capabilities and bind w/ the circuit's
   * full bitrate.
   */

  /* bind stream factory object */
  sfac = (mzs_factory) yoBind( mzs_factory__id, (char *)0, (yoRefData *)0,
              (char *)0 );

  /* indicate that we can receive audio and MPEG-I video
   * and can pause and blindly seek but not scan (rate control) */
  caps = mzs_capAudio | mzs_capVideo | mzs_capMpeg1 | mzs_capMpeg2 |
    mzs_capSeek | mzs_capPause;

  yslPrint("allocating stream ... ");
  strm = mzs_factory_alloc( sfac, env, circ, caps, bitrate );
  yslPrint("allocated\n");

  /* release mzs factory object (no longer needed) */
  yoRelease((dvoid *)sfac);

  /*
   * query the content service to get the asset cookie and segment info
   * corresponding to the requested video tag file.
   */
  sts = ovsdemoQuery( env, tagfile, TRUE, &cookie, &asset_bitrate );
  if (sts == FALSE)
    goto donestream;

  /* prepare stream to play from beginning to end at the given bitrate.
   * playNow starts the movie immediately (no separate play cmd necessary) */
  yslPrint( "preparing stream ( playing immediately ) ... " );
  inst = mzs_stream_prepare( strm, env, cookie, (mkd_pos *)&mkdBeginning,
              (mkd_pos *)&mkdEnd, asset_bitrate,
               mzs_stream_playNow,
               &status,
               (dvoid *)0);
  yslPrint( "prepared.\n" );

  /* throw away status info */
  mkd_segInfoList__free( &status, yoFree );

  /* at this point we should be receiving video. Since this is a demo,
   * we don't do anything with it, but merely sleep for a while */
  sysb8ext( &tm,  10000000);
  yslPrint( "sleeping for 10 seconds\n" );
  ysTimer( &tm, (ysevt *)0 );

  /* here's how to pause a stream (if you're curious) */
```

```
  yslPrint( "pausing stream ... " );
  mzs_stream_play(strm, env, inst,
           (mkd_pos *)&mkdCurrent, (mkd_pos *) &mkdCurrent,
           (mkd_pos *)&mkdCurrent, mzs_stream_ratePause, asset_bitrate);
  yslPrint( "paused\n" );

  /* wait some more */
  yslPrint( "sleeping for 3 seconds\n" );
  sysb8ext( &tm, 3000000 );
  ysTimer( &tm, (ysevt *)0 );

  /* ... and how to resume it */
  yslPrint( "resuming stream ... " );
  mzs_stream_play( strm, env, inst, (mkd_pos *)&mkdCurrent,
           (mkd_pos *)&mkdCurrent, (mkd_pos *)&mkdEnd,
            mzs_stream_rate1x, bitrate );
  yslPrint( "resumed\n" );

  /* wait a little more */
  sysb8ext( &tm, 5000000 );
  yslPrint( "sleeping for 5 seconds\n" );
  ysTimer( &tm, (ysevt *)0 );


  /* stop the stream */
  yslPrint( "finishing stream ... " );
  mzs_stream_finish( strm, env, inst, mzs_stream_finishOne );
  yslPrint( "finished.\n" );

donestream:
  yslPrint( "deallocating stream ... " );
  mzs_stream_dealloc( strm, env );
  yslPrint( "deallocated.\n" );

  return;
}

#ifdef unix
static void ovsdemoSocket( const char *id, const char *cproto,
                           const char *ctrl, const char *dproto, ub4 bitrate,
                           const char *blobname, const char *tagfile )
{
  yoenv       env;                                  /* active ORB environment */
  mzz_session ses;                                          /* session object */
  mzc_circuit dcirc;                           /* downstream circuit object */

  mkd_assetCookie cookie;
  ub4 asset_bitrate;
  boolean sts;

  mzs_factory sfac;
  mzs_stream strm;
  mzs_stream_instance inst;
  mzs_capMask caps;
  mkd_segInfoList status;

  sysb8 tm;
  char data[256];                           /* string containing port address */
  int port;
  boolean dgram;
  ub2 fs, cs, cn;

  /*
   * Setup a port to recieve the video data
   */
  port = ovsdemoSetupDataPort(dproto,data,&dgram);
  if(port < 0)
  {
```

```
  yslPrint("Error setting up data port\n");
  return;
}

/* initialize ORB environment */
yoEnvInit( &env );

/*
 * Estable session with Oracle Video Server and acquire a
 * high-speed data circuit for video
 */
ovsdemoSession( &env, id, cproto, ctrl, dproto, data, bitrate, &ses, &dcirc);

if(!dgram)
  port = ovsdemoConnectDataPort(port);

/* bind stream factory object */
sfac = (mzs_factory) yoBind( mzs_factory__id, (char *)0, (yoRefData *)0,
            (char *)0 );

/* indicate that we can receive audio and MPEG-I video
 * and can pause and blindly seek but not scan (rate control) */
caps = mzs_capAudio | mzs_capVideo | mzs_capMpeg1 | mzs_capMpeg2 |
  mzs_capSeek | mzs_capPause;

yslPrint("allocating stream ... ");
strm = mzs_factory_alloc( sfac, &env, &dcirc, caps, bitrate );
yslPrint("allocated\n");

/* release mzs factory object (no longer needed) */
yoRelease((dvoid *)sfac);

/*
 * query the content service to get the asset cookie and segment info
 * corresponding to the requested video tag file.
 */
sts = ovsdemoQuery( &env, tagfile, TRUE, &cookie, &asset_bitrate );
if (sts == FALSE)
  goto donestream;

/* prepare stream to play from beginning to end at the given bitrate.
 * playNow starts the movie immediately (no separate play cmd necessary) */
yslPrint( "preparing stream ( playing immediately ) ... " );
inst = mzs_stream_prepare( strm, &env, cookie, (mkd_pos *)&mkdBeginning,
             (mkd_pos *)&mkdEnd, asset_bitrate,
              mzs_stream_playNow,
              &status,
              (dvoid *)0);
yslPrint( "prepared.\n" );

/* throw away status info */
mkd_segInfoList__free( &status, yoFree );

/* at this point we should be receiving video. Since this is a demo,
 * we don't do anything with it, but merely sleep for a while */
sysb8ext( &tm,  10000000);
yslPrint( "reading data for 10 seconds\n" );
fs = cs = cn = 1;
ovsdemoReadDataPort(port, dgram, &tm, &fs, &cs, &cn);

/* rewind to the beginning  */
yslPrint( "rewinding stream ... " );
mzs_stream_play_nw( strm, &env, inst, (mkd_pos *)&mkdCurrent,
         (mkd_pos *)&mkdBeginning, (mkd_pos *)&mkdEnd,
          mzs_stream_rate1x, bitrate, ysEvtDummy() );
yslPrint( "rewind command issued\n" );

/* wait a little more */
```

```
  sysb8ext( &tm, 5000000 );
  yslPrint( "reading more data for 5 seconds\n" );
  ovsdemoReadDataPort(port, dgram, &tm, &fs, &cs, &cn);

  /* stop the stream */
  yslPrint( "finishing stream ... " );
  mzs_stream_finish( strm, &env, inst, mzs_stream_finishOne );
  yslPrint( "finished.\n" );

donestream:
  yslPrint( "deallocating stream ... " );
  mzs_stream_dealloc( strm, &env );
  yslPrint( "deallocated.\n" );

  /*
   * Clean up session resources and exit
   */
  yslPrint( "releasing session ... " );
  mzz_ses_Release( ses.or, &env );
  yslPrint( "released.\n" );


  mzz_session__free( &ses, yoFree );
  mzc_circuit__free( &dcirc, yoFree );

  yoEnvFree( &env );

  /* the end */
  yslPrint( "OVS demo exiting successfully ...\n" );
  return;
}

/*
 * ovsdemoSetupDataPort - setup a port to receive OGF data
 *
 */
static int ovsdemoSetupDataPort(const char *dproto, char *data, boolean *dgram)
{
  int s;
  int sinlen;
  struct sockaddr_in sin;
  struct hostent *ent;

  if(strcmp(dproto,"UDP") == 0)
  {
    s = socket(PF_INET, SOCK_DGRAM, 0);
    *dgram = TRUE;
  }
  else if(strcmp(dproto,"TCP") == 0)
  {
    s = socket(PF_INET, SOCK_STREAM, 0);
    *dgram = FALSE;
  }
  else
    yslPrint( "don't know how to build a %s port\n", dproto);

  ent = gethostbyname(ysGetHostName());
  if(!ent)
  {
    /* unable to get host's address */
    return -1;
  }

  /* bind socket to any address */
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = ((struct in_addr *)ent->h_addr_list[0])->s_addr;
  sin.sin_port = 0;
  sinlen = sizeof(sin);
```

```
  if(bind(s, (struct sockaddr *) &sin, sinlen) == -1)
  {
    /* bind failed */
    return -1;
  }

  /* devine what we bound to */
  sinlen = sizeof(sin);
  if(getsockname(s, (struct sockaddr *) &sin, &sinlen) == -1)
  {
    /* unable to get socket name */
    return -1;
  }

  /* construct a port address for OVS */
  sprintf(data,"%s:%d",inet_ntoa(sin.sin_addr),ntohs(sin.sin_port));

  if(!*dgram)
  {
    if(listen(s,1) == -1)
    {
      /* listen failed */
      return -1;
    }
  }

  yslPrint("Opened port %s:%s\n",dproto,data);
  return s;
}

static int ovsdemoConnectDataPort(int port)
{
  int cs, sinlen;
  struct sockaddr_in sin;

  sinlen = sizeof(sin);
  cs = accept(port, (struct sockaddr *) &sin, &sinlen);
  if(cs == -1)
  {
    /* accept failed */
  }
  close(port);               /* stop listening for additional connections */

  return cs;
}

/*
 * ovsdemoReadDataPort - read an OGF packet from the downstream channel
 *
 */
static void ovsdemoReadDataPort(int port, boolean dgram, sysb8 *dur, ub2 *fs,
                                ub2 *cs, ub2 *cn)
{
  int  rlen, l, len, sinlen;
  struct sockaddr_in sin;
  ub1  buf[8216], *p, uc;
  int  buflen;
  ub4  magic, sz, type;
  ub2  fseq, cseq, cnum;
  ub1  rev, offset;
  ysevt *evt;

  evt = ysSemCreate((dvoid *)0);
  ysTimer(dur,evt);
  while(!ysSemTest(evt))
  {
    if(dgram)
    {
```

```
      sinlen = sizeof(sin);
      l = recvfrom(port,(char *)buf,sizeof(buf),0,(struct sockaddr *)&sin,
                   &sinlen);
      if(l == -1)
      {
        /* error */
        perror("recvfrom");
      }
      if(l < 20)
      {
        /* short packet */
      }
      if(sysxGetB4(buf) != 0xf0f0f003)
      {
        /* unknown packet */
      }
      sz = sysxGetB4(buf+4);
    }
    else
    {
      /* assemble OGF packet from byte stream */

      /* search for magic number in stream */
      for(p = buf, len = 4; len; p += l, len -= l)
      {
        l = read(port,p,len);
        if(l < 1)
        {
          /* error or end of stream */
        }
      }
      magic = sysxGetB4(buf);
      while(magic != 0xf0f0f003)
      {
        /* slow search for the magic number */
        if((l=read(port,&uc,1)) != 1)
        {
          /* error or end of stream */
        }
        magic = ((magic << 8) & 0xffffff00) | uc;
        sysxPutB4(buf,magic);
      }

      /* read the remainder of the header */
      for(p = buf+4, len = 16; len; p += l, len -= l)
      {
        l = read(port,p,len);
        if(l < 1)
        {
          /* error or end of stream */
        }
      }
      /* get size of payload and read it */
      sz = sysxGetB4(buf+4);
      for(len = sz; len; p += l, len -= l)
      {
        l = read(port,p,len);
        if(l < 1)
        {
          /* error or end of stream */
        }
      }
    }

    fseq = sysxGetB2(buf+8);
    rev  = buf[10];
    offset = buf[11];
    type = sysxGetB4(buf+12);
```

```
    cseq = sysxGetB2(buf+16);
    cnum = sysxGetB2(buf+18);

    /* detect a command response (good data follows) */
    if(cnum != *cn)
    {
      yslPrint("Command response received.\n");
      *cn = cnum;
    }

    /* see if anything was dropped */
    if(fseq != *fs)
    {
      yslPrint("Frame drop: expected %d received %d.\n",*fs,fseq);
      if(*fs < cseq + 5)
        yslPrint("Warning: Drop within first 5 frames of command.\n");
    }
    *fs = fseq + 1;

    /* display the OGF header */
    yslPrint("OGF: Rev %d  Data length: %d  Data offset: %d  Type:",
             rev, sz, offset);
    if(type & 0x0001)
      yslPrint(" Video");
    if(type & 0x0002)
      yslPrint(" Control");
    if(type & 0x0100)
      yslPrint(" Data");
    yslPrint("\n     FSeq: %d  CSeq: %d  CNum: %d\n\n", fseq, cseq, cnum);

    ysIdle();                                /* give YS some processing time */
  }
}
#endif
```

ctntdemo.c

```
/* Copyright (c) 1996 by Oracle Corporation.  All Rights Reserved.
 *
 * ctntdemo.c - OVS Logical Content Demo binary
 *
 * This demo details some other features of the Content and Content Provider
 * Interface. The Content object has a lot of attributes, (See CtntAtr typedef
 * in mzalgctn.idl). This demo only deals with a few of them, the Name,
 * Description and Content Provider. All other attributes are not displayed or
 * are set to defaults.
 *
 *
 */

#ifndef SYSX_ORACLE
#include <sysx.h> /* Oracle defs */
#endif
#ifndef ORASTDDEF
#include <stddef.h>
#define ORASTDDEF
#endif
#ifndef ORASTDLIB
#include <stdlib.h>
#define ORASTDLIB
#endif
#ifndef ORASTRING
#include <string.h>
#define ORASTRING
#endif
#ifndef ORACTYPE
#include <ctype.h>
#define ORACTYPE
#endif
#ifndef ORASTDIO
#include <stdio.h>
#define ORASTDIO
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h> /* filesystem defs */
#endif
#ifndef SYSB8_ORACLE
#include <sysb8.h> /* eight-byte types */
#endif
#ifndef YS_ORACLE
#include <ys.h> /* System layer defs */
#endif
#ifndef YSR_ORACLE
#include <ysr.h> /* resource db defs */
#endif
#ifndef MN_ORACLE
#include <mn.h> /* Media Net defs */
#endif
#ifndef MTUX_ORACLE
#include <mtux.h> /* mtux init */
#endif
#ifndef YO_ORACLE
#include <yo.h> /* orb defs */
#endif
#ifndef YOORB_ORACLE
#include <yoorb.h> /* orb defs */
#endif
#ifndef MKD_ORACLE
#include <mkd.h> /* common datatypes */
#endif
#ifndef MZALGCTN_ORACLE
```

```
#include <mzalgctn.h> /* logical content datatypes */
#endif


/* FORWARD DEFINITIONS AND FUNCTION PROTOTYPES */

static struct ysargmap ctntdemoArgs[] =
{
   {0, "", 0}
};


static void     ctntdemo(void);
static boolean  checkUsingDB(yoenv * env);
static void     print_content_provider_list(mza_CtntPvdrAtrLst * atrLst);
static void     print_content_provider(mza_CtntPvdrAtr * atr);
static void     print_content_list(mza_CtntAtrLst * atrLst);


sb4
main(int argc, char **argv)
{
   ub1             ctxbuf[SYSX_OSDPTR_SIZE];    /* global context storage
                         * space */
   char            base[SYSFP_MAX_PATHLEN];


   /*
    * get basename of program for ys
    */
   sysfpExtractBase(base, argv[0]);

   /*
    * Initialize the Media Net Services
    */
   if (mtuxInit((dvoid *) ctxbuf, base, (mnLogger)0,
                (sword)(argc - 1), argv + 1, ctntdemoArgs) != mtuxLayerSuccess)
      return (1);

   /*
    * Run the main routine
    */
   yseTry
   {
      ctntdemo();
   }
   yseCatchAll
   {
      yslPrint("Unexpected exception thrown by ctntdemo: %s -- exiting\n",
               yseExid);
   }
   yseEnd;

   /*
    * Terminate what we started
    */
   DISCARD mtuxTerm((dvoid *) ctxbuf);
   return (1);
}

/*
 * ctntdemo - main routine which does all the work
 *
 *
 *
 */
static void
ctntdemo(void)
```

```
{
   yoenv           env;        /* active ORB environment */
   char            ans[32];    /* Buffer to hold the choice in */
   int             res;
   boolean         sts;
   mza_CtntFac     ctntFacOR;
   mza_CtntMgmt    ctntMgmtOR;
   mza_CtntPvdrMgmt ctntPvdrMgmtOR;
   mza_CtntPvdrFac ctntPvdrFacOR;
   mza_CtntPvdrAtrLst cpAtrLst;
   mza_CtntPvdrAtr cpAtr;
   mza_CtntAtr     cAtr;
   mza_CtntAtrLst  cAtrLst;
   mza_Itr         itr;
   char            name[80];
   char            newname[80];
   char            newdesc[236];
   boolean         err;

   /*
    * initialize ORB environment
    */
   yoEnvInit(&env);

   /*
    * Make sure that the content manager is using the Database. If it is not,
    * then there is no way to create content and no point in running this
    * demo.
    */
   if (checkUsingDB(&env) == FALSE)
   {
      yoEnvFree(&env);        /* Freeup our environment */
      yslPrint("vscontsrv was started in standalone mode ...\n");
      yslPrint("No Content operation supported ...\n");
      return;
   }

   /*
    * Create some loosely bound Object References we will need later. This
    * will need to be released when we are done with them
    */
   ctntPvdrFacOR = (mza_CtntPvdrFac) yoBind(mza_CtntPvdrFac__id, (char *) 0,
                   (yoRefData *) 0, (char *) 0);
   ctntPvdrMgmtOR = (mza_CtntPvdrMgmt) yoBind(mza_CtntPvdrMgmt__id, (char *) 0,
                    (yoRefData *) 0, (char *) 0);
   ctntFacOR = (mza_CtntFac) yoBind(mza_CtntFac__id, (char *) 0,
                    (yoRefData *) 0, (char *) 0);
   ctntMgmtOR = (mza_CtntMgmt) yoBind(mza_CtntMgmt__id, (char *) 0,
                   (yoRefData *) 0, (char *) 0);

   /*
    * Enter the main select loop. This loop prints a menu out, asks the user
    * to enter a choice and then performs the chosen operation.
    */
   while (TRUE)
   {
      yslPrint("Content Demo Main Menu\n\n");
      yslPrint("1)  List Content Providers\n");
      yslPrint("2)  Get Content Provider By Name\n");
      yslPrint("3)  Update Content Provider By Name\n");
      yslPrint("4)  Create Content Provider\n");
      yslPrint("5)  Destroy Content Provider By Name\n");
      yslPrint("6)  List Content\n");
      yslPrint("7)  List Content By Name\n");
      yslPrint("8)  Update Content By Name\n");
      yslPrint("9)  Create Content\n");
      yslPrint("10) Destroy Content By Name\n\n");
      yslPrint("99) Exit\n");
```

```
 yslPrint("Enter a Choice: ");
 sts = yslGets(ans, sizeof(ans));
 res = (int) atol(ans);

 switch (res)
 {

 case 1:            /* List Content Providers */
/*
 * Initialize the iterator to return at most 10 content providers at
 * a time
 */
itr.Position = 0;
itr.NumItems = 10;

/*
 * Make sure to get them all. We know we have them all when the
 * iterator position is -1. Get a batch of content providers and
 * then print them out.
 */
while (itr.Position != -1)
{
   cpAtrLst = mza_CtntPvdrMgmt_lstAtr(ctntPvdrMgmtOR, &env, &itr);
   print_content_provider_list(&cpAtrLst);
   /*
    * Make sure to free our results so we don't leak any memory
    */
   mza_CtntPvdrAtrLst__free(&cpAtrLst, yoFree);
}
break;

 case 2:            /* Get Content Provider By Name */
/*
 * Ask for a content proivder name. We will then retrive the content
 * provider by this name.
 */
yslPrint("Enter a Content Provider Name: ");
sts = yslGets(name, sizeof(name));

/*
 * The 'err' boolean is used here because we cannot return from
 * within a yseTry block
 */
yseTry
{
   err = FALSE;
   mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
   name, &cpAtr);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntPvdrMgmt_getAtrByNm: %s\n",
   yseExid);
   err = TRUE;
}
yseEnd;
if (err)
   break;

/*
 * Print this content providers info
 */
print_content_provider(&cpAtr);

/*
 * Make sure to free our results
 */
mza_CtntPvdrAtr__free(&cpAtr, yoFree);
```

```
break;
 case 3:            /* Update Content Provider By Name */
/*
 * Before we update a content provider we need to get an object
 * reference to it. Use the getAtrByNm mathod to do this. First ask
 * for a name to get the OR, then ask for the new name and desc,
 * then update them.
 */

yslPrint("Enter a Content Provider Name for Update: ");
sts = yslGets(name, sizeof(name));

/*
 * get the Content Provider OR
 */
yseTry
{
   err = FALSE;
   mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
               name, &cpAtr);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntPvdrMgmt_getAtrByNm: %s\n",
   yseExid);
   err = TRUE;
}
yseEnd;
if (err)
   break;

/*
 * Print out the one we got
 */
print_content_provider(&cpAtr);

/*
 * Enter the new name and description
 */
yslPrint("Enter a New Content Provider Name: ");
sts = yslGets(newname, sizeof(newname));
yslPrint("Enter a New Content Provider Description: ");
sts = yslGets(newdesc, sizeof(newdesc));

/*
 * Update the name and description. There is no setAtr method like
 * the content object has so we have to set each attribute
 * individually.
 */
yseTry
{
   err = FALSE;
   mza_CtntPvdr__set_name(cpAtr.ctntPvdrOR, &env, newname);
   mza_CtntPvdr__set_desc(cpAtr.ctntPvdrOR, &env, newdesc);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntPvdr__set_name/desc: %s\n",
   yseExid);
   err = TRUE;
}
yseEnd;

/*
 * Make sure to free our results
 */
mza_CtntPvdrAtr__free(&cpAtr, yoFree);
```

```
/*
 * If there was en error don't continue
 */
if (err)
   break;

/*
 * Check the results to make sure it worked.
 */
yseTry
{
   err = FALSE;
   mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
               newname, &cpAtr);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntPvdrMgmt_getAtrByNm: %s\n",
        yseExid);
   err = TRUE;
}
yseEnd;
if (err)
   break;

print_content_provider(&cpAtr);

mza_CtntPvdrAtr__free(&cpAtr, yoFree);


break;
 case 4:            /* Create Content Provider */
/*
 * To create a new content provider, we need a name and description
 * for it. Ask for the new name and description and then call the
 * method to create a new one.
 */
yslPrint("Enter a New Content Provider Name: ");
sts = yslGets(newname, sizeof(newname));
yslPrint("Enter a New Content Provider Description: ");
sts = yslGets(newdesc, sizeof(newdesc));

/*
 * Ignore the result from the create method since we don't need it
 * for anything
 */
yseTry
{
   err = FALSE;
   DISCARD mza_CtntPvdrFac_create(ctntPvdrFacOR, &env,
                                  newname, newdesc);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntPvdrFac_create: %s\n",
   yseExid);
   err = TRUE;
}
yseEnd;
if (err)
   break;

/*
 * Check the results to make sure it was created properly
 */
yseTry
{
```

```
         err = FALSE;
         mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
                     newname, &cpAtr);
      }
      yseCatchAll
      {
         yslPrint("exception thrown by mza_CtntPvdrMgmt_getAtrByNm: %s\n",
         yseExid);
         err = TRUE;
      }
      yseEnd;

      if (err)
         break;

      print_content_provider(&cpAtr);

      mza_CtntPvdrAtr__free(&cpAtr, yoFree);

      break;
       case 5:            /* Destroy Content Provider By Name */
      /*
       * To destroy a content provider, we need an object reference to
       * one. use getAtrByNm to get it, then call the destroy method on
       * it.
       */
      yslPrint("Enter a Content Provider Name to Destroy: ");
      sts = yslGets(name, sizeof(name));

      /*
       * get the Content Provider OR
       */
      yseTry
      {
         mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
                     name, &cpAtr);
         mza_CtntPvdr_destroy(cpAtr.ctntPvdrOR, &env);
         mza_CtntPvdrAtr__free(&cpAtr, yoFree);
      }
      yseCatchAll
      {
         yslPrint("exception thrown by mza_CtntPvdrMgmt_getAtrByNm \
or mza_CtntPvdr_destroy: %s\n",
         yseExid);
      }
      yseEnd;

      break;
       case 6:            /* List Content */
      /*
       * Lst all the content objects out. Call the lstAtr method until we
       * retireve them all. Set the iterator so we get 10 at a time.
       */

      itr.Position = 0;
      itr.NumItems = 10;

      /*
       * Make sure to get them all. The iterator Position will be -1 when
       * there are no more.
       */
      while (itr.Position != -1)
      {
         cAtrLst = mza_CtntMgmt_lstAtr(ctntMgmtOR, &env, &itr);

         print_content_list(&cAtrLst);

         mza_CtntAtrLst__free(&cAtrLst, yoFree);
```

```
}

break;
 case 7:            /* List Content By Name */
/*
 * Get some content based on a name. Wildcards are allowed. See the
 * IDL for description of what wildcards will work. First enter a
 * name to query on, then make the query until there are no more
 * which match the given name.
 */
yslPrint("Enter a Content Name (wildcards OK): ");
sts = yslGets(name, sizeof(name));

/*
 * Get 10 at a time
 */
itr.Position = 0;
itr.NumItems = 10;

/*
 * Make sure to get them all
 */
while (itr.Position != -1)
{
   cAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, &env, name, &itr);
   print_content_list(&cAtrLst);

   mza_CtntAtrLst__free(&cAtrLst, yoFree);
}

break;
 case 8:            /* Update Content By Name */
/*
 * To update content, you need to get the existing attribute
 * structure for the content and then modify those parameters you
 * would like to change and then call setAtr on the content object.
 * This code allow you to modify the name, the description or the
 * content provider only.
 */
yslPrint("Enter a Content Name (No wildcards): ");
sts = yslGets(name, sizeof(name));

/*
 * We set the number of items to one becuase I only want to get back
 * one item to update. I sugest up above not to enter wildcards,
 * since this may cause more than one content object to be selected.
 * By setting NumItems to one, I will get back at most only 1 item.
 * If 0 items are returned, we print an error message and break
 */
itr.Position = 0;
itr.NumItems = 1;

cAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, &env, name, &itr);

/*
 * If we didn't get exactly one we are in trouble
 */
if (itr.NumItems != 1)
{
   yslError("Returned %d content items. Can only update one\n",
        itr.NumItems);
   mza_CtntAtrLst__free(&cAtrLst, yoFree);

   break;
}
/*
 * print out what we got
 */
```

```
print_content_list(&cAtrLst);

/*
 * We decided to only allow one of the three allowable attributes to
 * be updated at a time in this example. Ask the user which one they
 * would like to update here and then perform the necessary actions
 * to update it. To update an attribute, the corresponding attribute
 * in the CtntAtrLst returned above is modified and then passed in
 * to the setAtr method.
 */
yslPrint("What would you like to update?\n");
yslPrint("1) Content Name\n");
yslPrint("2) Content Description\n");
yslPrint("3) Content Provider\n");
yslPrint("4) Nothing\n");
yslPrint("Enter a Choice: ");
sts = yslGets(ans, sizeof(ans));
res = (int) atol(ans);

switch (res)
{
case 1:        /* Update Content Name */
   /*
    * Ask for a new content name
    */

   yslPrint("Enter a New Content Name: ");
   sts = yslGets(newname, sizeof(newname));
   /*
    * Free the existing memory in the attribute structure , allocate
    * new memory to hold the new name, and then copy in the new
    * name.
    */
   yoFree((dvoid *) cAtrLst._buffer[0].name);
   cAtrLst._buffer[0].name = (char *) yoAlloc(strlen(newname) + 1);
   DISCARD strcpy(cAtrLst._buffer[0].name, newname);
   break;
case 2:        /* Update Content Description */
   /*
    * Ask for a new content description
    */
   yslPrint("Enter a New Content Description: ");
   sts = yslGets(newdesc, sizeof(newdesc));
   /*
    * Free the existing memory in the attribute structure , allocate
    * new memory to hold the new description, and then copy in the
    * new description.
    */
   yoFree((dvoid *) cAtrLst._buffer[0].desc);
   cAtrLst._buffer[0].desc = (char *) yoAlloc(strlen(newdesc) + 1);
   DISCARD strcpy(cAtrLst._buffer[0].desc, newdesc);
   break;
case 3:        /* Update Content Provider */
   /*
    * To update the content provider, we need to get an object
    * reference to one first. Ask for a new content provider name.
    * To set the content provider to NULL, enter an empty string.
    * First though, free the object reference in the original
    * content attribute structure so we can change it later.
    */
   if (cAtrLst._buffer[0].ctntPvdrOR != (mza_CtntPvdr) CORBA_OBJECT_NIL)
   {
      yoRelease((dvoid *) cAtrLst._buffer[0].ctntPvdrOR);
   }

   /*
    * Ask for a content provider name and then get the OR for it.
    */
```

```
        yslPrint("Enter a New Content Provider Name: ");
        sts = yslGets(newname, sizeof(newname));
        if (strlen(newname) > 0)
        {
           yseTry
           {
          err = FALSE;
          mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
                    newname, &cpAtr);
           }
           yseCatchAll
           {
          yslPrint("exception thrown by \
mza_CtntPvdrMgmt_getAtrByNm: %s\n",
               yseExid);
          err = TRUE;
           }
           yseEnd;
           if (err)
          break;

           cAtrLst._buffer[0].ctntPvdrOR = (mza_CtntPvdr)
          yoDuplicate((const dvoid *) cpAtr.ctntPvdrOR);
           mza_CtntPvdrAtr__free(&cpAtr, yoFree);
        }
        else
           cAtrLst._buffer[0].ctntPvdrOR = (mza_CtntPvdr) CORBA_OBJECT_NIL;

        break;
     case 4:
        /*
         * Don't update anything
         */
        break;
     default:
        yslPrint("Sorry, Try again\n");
        break;
     }
     /*
      * Update the new parameter for the content object. Pass in the
      * content attribute structure which has one parameter modified.
      */
     yseTry
     {
        mza_Ctnt_setAtr(cAtrLst._buffer[0].ctntOR, &env,
           &cAtrLst._buffer[0]);
     }
     yseCatchAll
     {
        yslPrint("exception thrown by mza_Ctnt_setAtr: %s\n",
             yseExid);
     }
     yseEnd;

     /*
      * No we can free the content attribute structure
      */
     mza_CtntAtrLst__free(&cAtrLst, yoFree);

     /*
      * Check the results to make sure the update worked
      */
     itr.Position = 0;
     itr.NumItems = 1;

     cAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, &env,
            cAtrLst._buffer[0].name, &itr);
```

```
     print_content_list(&cAtrLst);


     break;
      case 9:            /* Create Content */
     /*
      * We only allow a few parameters to be specified for the new
      * content object. The rest of them are set to some arbitrary
      * numbers or values. First get the new content attributes, allocate
      * space in the content attribute structure we will use to create
      * the content, and copy the values in.
      */
     yslPrint("Enter a New Content Name: ");
     sts = yslGets(newname, sizeof(newname));
     cAtr.name = (char *) yoAlloc(strlen(newname) + 1);
     DISCARD strcpy(cAtr.name, newname);

     yslPrint("Enter a New Content Description: ");
     sts = yslGets(newdesc, sizeof(newdesc));
     cAtr.desc = (char *) yoAlloc(strlen(newdesc) + 1);
     DISCARD strcpy(cAtr.desc, newdesc);

     yslPrint("Enter a New Content Provider Name: ");
     sts = yslGets(newname, sizeof(newname));
     if (strlen(newname))
     {
        yseTry
        {
           err = FALSE;
           mza_CtntPvdrMgmt_getAtrByNm(ctntPvdrMgmtOR, &env,
                       newname, &cpAtr);
        }
        yseCatchAll
        {
           yslPrint("exception thrown by \
mza_CtntPvdrMgmt_getAtrByNm: %s\n",
            yseExid);
           err = TRUE;
        }
        yseEnd;
        if (err)
           break;

        cAtr.ctntPvdrOR = (mza_CtntPvdr)
           yoDuplicate((const dvoid *) cpAtr.ctntPvdrOR);

        mza_CtntPvdrAtr__free(&cpAtr, yoFree);
     }
     else
        cAtr.ctntPvdrOR = (mza_CtntPvdr) CORBA_OBJECT_NIL;

     /*
      * Fill in the rest of the structure with defaults
      */
     cAtr.ctntOR = (mza_Ctnt) CORBA_OBJECT_NIL;    /* Not used in create */
         cAtr.createDate.mkd_wallNano = 0;
         cAtr.createDate.mkd_wallSec = 0;
         cAtr.createDate.mkd_wallMin = 0;
         cAtr.createDate.mkd_wallHour = 0;
         cAtr.createDate.mkd_wallDay = 1;
         cAtr.createDate.mkd_wallMonth = 1;
         cAtr.createDate.mkd_wallYear = 1997;
     cAtr.filename = (char *)(dvoid*) "/mds/video/sample.mpi";
     sysb8set(&cAtr.len,sysb8zero);
     cAtr.msecs = 0;
     cAtr.rate = 1000;
     sysb8set(&cAtr.firstTime,sysb8zero);
     sysb8set(&cAtr.lastTime,sysb8zero);
```

```
    cAtr.format.mkd_contFormatVendor = (char*)0;
    cAtr.format.mkd_contFormatFmt = mkd_compFormatMpeg1;
    cAtr.format.mkd_contFormatVid._length = 0;
    cAtr.format.mkd_contFormatVid._maximum = 0;
    cAtr.format.mkd_contFormatVid._buffer = (ub1*)0;
    cAtr.format.mkd_contFormatAud._length = 0;
    cAtr.format.mkd_contFormatAud._maximum = 0;
    cAtr.format.mkd_contFormatAud._buffer = (ub1*)0;
    cAtr.format.mkd_contFormatHeightInPixels = 320;
    cAtr.format.mkd_contFormatWidthInPixels = 240;
    cAtr.format.mkd_contFormatPelAspectRatio = 2;
    cAtr.format.mkd_contFormatFrameRate = 30000;
cAtr.prohibFlags = (mkd_prohib) 0;
cAtr.tagsFlag = TRUE;
cAtr.multiRateFlag = FALSE;
cAtr.reliableFlag = FALSE;
cAtr.volLocation = (char *) NULL;
cAtr.contStatus = mkd_contStatusDisk;
cAtr.assigned = FALSE;
cAtr.sugBufSz = mza_BufSzUnknown;

/*
 * Create the new content
 */
yseTry
{
   DISCARD mza_CtntFac_create(ctntFacOR, &env, &cAtr);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_CtntFac_create: %s\n",
        yseExid);
}
yseEnd;

break;
 case 10:            /* Destroy Content By Name */
/*
 * Get the name of the content to destroy, query to get the
 * attribute structure, use the object reference to make the destroy
 * call
 */
yslPrint("Enter a Content Name (No wildcards): ");
sts = yslGets(name, sizeof(name));

/*
 * We only want one content object back
 */
itr.Position = 0;
itr.NumItems = 1;

cAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, &env, name, &itr);

if (itr.NumItems != 1)
{
   yslError("Returned %d content items. Can only destroy one\n",
        itr.NumItems);
   break;
}

yseTry
{
   mza_Ctnt_destroy(cAtrLst._buffer[0].ctntOR, &env, FALSE, FALSE);
}
yseCatchAll
{
   yslPrint("exception thrown by mza_Ctnt_destroy: %s\n",
        yseExid);
```

```
     }
     yseEnd;

     break;
      case 99:
     /*
      * Release the loosely bound Object References
      */
     yoRelease((dvoid *) ctntPvdrMgmtOR);
     yoRelease((dvoid *) ctntPvdrFacOR);
     yoRelease((dvoid *) ctntMgmtOR);
     yoRelease((dvoid *) ctntFacOR);
     return;
      default:
     yslPrint("Invalid selection entered. Try Again\n");
         break;
      }
   }
}
/*
 * Function: checkUsingDB
 *
 * This function determines if the content service is using the
 * database or not.  It returns TRUE if it is and FALSE is it is not
 */
static          boolean
checkUsingDB(yoenv * env)
{
   mza_LgCtntMgmt  lgCtntMgmtOR;
   boolean         usingDB;

   lgCtntMgmtOR = (mza_LgCtntMgmt) yoBind(mza_LgCtntMgmt__id, (char *) 0,
                     (yoRefData *) 0, (char *) 0);
   /*
    * Search for the Logical Content for John
    */
   usingDB = mza_LgCtntMgmt_usingDB(lgCtntMgmtOR, env);

   yoRelease((dvoid *) lgCtntMgmtOR);

   return usingDB;
}
/*
 * Function: print_content_list
 *
 * This function prints out some info about a list of content
 */
static void
print_content_list(mza_CtntAtrLst * atrLst)
{

   int             i;
   yoenv           env;

   yoEnvInit(&env);

   yslPrint("Name                                Description\
                    Provider\n");
   yslPrint("------------------------------------------\
------------------------------\n");
   /*
    * Loop through the content list and print some info about each one
    */
   for (i = 0; i < atrLst->_length; ++i)
   {
      char           *name;

      /*
```

```
       * For each content, get the content provider if there is one
       */
      if (atrLst->_buffer[i].ctntPvdrOR != (mza_CtntPvdr) CORBA_OBJECT_NIL)
     name = mza_CtntPvdr__get_name(atrLst->_buffer[i].ctntPvdrOR, &env);
      else
     name = (char *) NULL;

      yslPrint("%-32s  %-32s %-s\n",
           atrLst->_buffer[i].name,
           atrLst->_buffer[i].desc,
           name ? name : "NULL");
      if (name)
     yoFree((dvoid *) name);
   }
   yslPrint("\n\n");
   yoEnvFree(&env);

}
/*
 * Function: print_content_provider_list
 *
 * This function prints out some info about a list of content providers
 */
static void
print_content_provider_list(mza_CtntPvdrAtrLst * atrLst)
{
   int             i;

   yslPrint("Name                         Description\n");
   yslPrint("-----------------------------------------\n");
   /*
    * Loop through the content provider list  and print some info about each
    * one
    */
   for (i = 0; i < atrLst->_length; ++i)
      yslPrint("%-26s  %-s\n",
           atrLst->_buffer[i].name,
           atrLst->_buffer[i].desc);
   yslPrint("\n\n");
}
/*
 * Function: print_content_provider
 *
 * This function prints out some info about a single content provider
 */
static void
print_content_provider(mza_CtntPvdrAtr * atr)
{

   yslPrint("Name                         Description\n");
   yslPrint("-----------------------------------------\n");
   yslPrint("%-26s  %-s\n",
        atr->name,
        atr->desc);
   yslPrint("\n\n");
}
```

clipdemo.c

```
/* Copyright (c) 1996 by Oracle Corporation.  All Rights Reserved.
 *
 * clipdemo.c - OVS Clip Demo binary
 *
 * This demo shows how a Logical Content may be created and clips
 * may be added to it dynamically. It also destroys clips, ctnt and
 * logical content when it is done.
 *
 *
 */

#ifndef SYSX_ORACLE
#include <sysx.h>/* Oracle defs */
#endif
#ifndef ORASTDDEF
#include <stddef.h>
#define ORASTDDEF
#endif
#ifndef ORASTDLIB
#include <stdlib.h>
#define ORASTDLIB
#endif
#ifndef ORASTRING
#include <string.h>
#define ORASTRING
#endif
#ifndef ORACTYPE
#include <ctype.h>
#define ORACTYPE
#endif
#ifndef ORASTDIO
#include <stdio.h>
#define ORASTDIO
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h>/* filesystem defs */
#endif
#ifndef SYSB8_ORACLE
#include <sysb8.h>/* eight-byte types */
#endif
#ifndef YS_ORACLE
#include <ys.h>/* System layer defs */
#endif
#ifndef YSR_ORACLE
#include <ysr.h>/* resource db defs */
#endif
#ifndef MN_ORACLE
#include <mn.h>/* Media Net defs */
#endif
#ifndef MTUX_ORACLE
#include <mtux.h>/* mtux Init */
#endif
#ifndef YO_ORACLE
#include <yo.h>/* orb defs */
#endif
#ifndef YOORB_ORACLE
#include <yoorb.h>/* orb defs */
#endif
#ifndef MKD_ORACLE
#include <mkd.h>/* common datatypes */
#endif
#ifndef MZALGCTN_ORACLE
#include <mzalgctn.h>/* common datatypes */
#endif
```

```
#ifndef MKDC_ORACLE
#include <mkdc.h>/* for mkdBeginning and mkdEnd */
#endif


/* FORWARD DEFINITIONS AND FUNCTION PROTOTYPES */

static struct ysargmap clipdemoArgs[] =
{
   {0, "", 0}
};

/*
 * Context structure to hold the Ctnt and Clip Object
 * References which get created for use in the new
 * Logical content
 */
struct clipDemoCtx {
    mza_Ctnt johnMovieCtnt;
    mza_Ctnt johnCommCtnt;
    mza_Ctnt maryMovieCtnt;
    mza_Ctnt maryCommCtnt;
    mza_Clip johnMovieClip;
    mza_Clip johnCommClip;
    mza_Clip maryMovieClip;
    mza_Clip maryCommClip;
};
typedef struct clipDemoCtx  clipDemoCtx;


static void     clipdemo(clipDemoCtx *demoCtx);
static mza_LgCtnt find_logical_content(const char *name, yoenv * env);
static mza_Clip find_clip(const char *name, yoenv * env);
static mza_Ctnt find_ctnt(const char *name, yoenv * env);
static boolean  checkUsingDB(yoenv * env);
static void     print_logical_content(mza_LgCtntAtr * lcAtr);
static booleancreate_clipdemo_objects(clipDemoCtx *demoCtx);
static voiddestroy_clipdemo_objects();

#define JOHNS_MOVIE_CTNT_NAME "Movie Ctnt for John"
#define JOHNS_COMM_CTNT_NAME  "Commercial Ctnt for John"
#define MARYS_MOVIE_CTNT_NAME "Movie Ctnt for Mary"
#define MARYS_COMM_CTNT_NAME  "Commercial Ctnt for Mary"
#define JOHNS_MOVIE_CLIP_NAME "Movie Clip for John"
#define JOHNS_COMM_CLIP_NAME  "Commercial Clip for John"
#define MARYS_MOVIE_CLIP_NAME "Movie Clip for Mary"
#define MARYS_COMM_CLIP_NAME  "Commercial Clip for Mary"
#define JOHNS_LOGICAL_CONTENT_NAME "John's Logical Content"
#define MARYS_LOGICAL_CONTENT_NAME "Mary's Logical Content"


sb4
main(int argc, char **argv)
{
   ub1             ctxbuf[SYSX_OSDPTR_SIZE];/* global context storage
            * space */
   char            base[SYSFP_MAX_PATHLEN];
   clipDemoCtx demoCtx;

   /* get basename of program for ys */
   sysfpExtractBase(base, argv[0]);

   /*
    * Initialize the Media Net Services
    */
   if (mtuxInit((dvoid*)ctxbuf, base, (mnLogger)0,
                (sword)(argc - 1), argv + 1, clipdemoArgs) != mtuxLayerSuccess)
     return 1;
```

```
   if(create_clipdemo_objects(&demoCtx) == FALSE)
   {
      yslError("Could not create required objects for the clip demo\n");
      destroy_clipdemo_objects();
      DISCARD mtuxTerm((dvoid*)ctxbuf);
      return(1);
   }

   yseTry
   {
      clipdemo(&demoCtx);
   }
   yseCatchAll
   {
      yslPrint("exception thrown by clipdemo: %s -- exiting\n", yseExid);
   }
   yseEnd;

   destroy_clipdemo_objects();

   /*
    * Terminate what we started
    */
   DISCARD mtuxTerm((dvoid*)ctxbuf);
   return(1);
}

/*
 * clipdemo - main routine which does all the work
 *
 * I don't add any exception handling in this routine. Typically each ORB
 * call should handle exceptions and react accordingly.
 *
 */
static void
clipdemo(clipDemoCtx *demoCtx)
{
   yoenv           env;/* active ORB environment */
   char            ans[32];/* Buffer to hold the choice in */
   int             res;
   sb4    pos;
   mza_LgCtntFac   lgCtntFacOR;
   mza_LgCtntAtr   lcAtr;
   mza_LgCtnt      lgCtntOR;
   boolean         sts;

   /* initialize ORB environment */
   yoEnvInit(&env);


   /*
    * Create some loosely boud Object References we will need later
    */
   lgCtntFacOR = (mza_LgCtntFac) yoBind(mza_LgCtntFac__id, (char *) 0,
          (yoRefData *) 0, (char *) 0);
   lgCtntOR = (mza_LgCtnt) CORBA_OBJECT_NIL;



   /*
    * Enter the select loop
    */
   while (TRUE)
   {
      yslPrint("Clip Demo Main Menu\n\n");
      yslPrint("1) Build Some Logical Content for John\n");
      yslPrint("2) Print Logical Content Info for John\n");
```

```
 yslPrint("3) Destroy Logical Content Info for John\n");
 yslPrint("4) Build Some Logical Content for Mary\n");
 yslPrint("5) Print Logical Content Info for Mary\n");
 yslPrint("6) Destroy Logical Content Info for Mary\n\n");
 yslPrint("99) Exit\n");
 yslPrint("Enter a Choice: ");
 sts = yslGets(ans, sizeof(ans));
 res = (int) atol(ans);

 switch (res)
 {

 case 1:/* Build Some Logical Content For John */
/*
 * We can't create another logical content without destroying the
 * old one first.
 */
yslPrint("Looking for John's Logical Content to Destroy\n");
lgCtntOR = find_logical_content(JOHNS_LOGICAL_CONTENT_NAME, &env);

if (lgCtntOR != (mza_LgCtnt) CORBA_OBJECT_NIL)
{
   /*
    * If John already has one, destroy it first.
    */
       mza_LgCtnt_destroy(lgCtntOR, &env);
   yslPrint("Destroyed John's Old Logical Content\n");
   yoRelease((dvoid *) lgCtntOR);
}

/*
 * Create a new Logical Content for John
 */
yslPrint("Creating John's New Logical Content\n");
lgCtntOR = mza_LgCtntFac_create(lgCtntFacOR, &env,
   (char*)(dvoid*)JOHNS_LOGICAL_CONTENT_NAME,
   (char*)(dvoid*)"Logical Content created by OVS clip demo");
yslPrint("Created John's New Logical Content\n");

/*
 * Imagine a case where, in order to reduce the price of a movie,
 * the viewer will be shown an advertisement first. In this case
 * a manufacturer of trucks has requested that their
 * commercial be shown to anyone who orders a certain movie.
 * We need to go get the commercial clip and the
 * movie  clip  from the content manager so we can add them to the
 * logical content. It turns out we have references to
 * the clips when we created them and stored them in the
 * structure 'demoCtx'. Use this structure to find the clips we want
 * and add them to the logical content. I add these in reverse
    * order to to show that it can be done.
 */

pos = mza_LgCtnt_addClip(lgCtntOR, &env, demoCtx->johnMovieClip);
mza_LgCtnt_addClipByPos(lgCtntOR, &env, demoCtx->johnCommClip,
      (sb4)1);

/*
 * Free the Object References we used
 */
yoRelease((dvoid *) lgCtntOR);


break;
 case 2:/* Print John's Logical Content Info */

lgCtntOR = find_logical_content(JOHNS_LOGICAL_CONTENT_NAME, &env);
if (lgCtntOR == (mza_LgCtnt) CORBA_OBJECT_NIL)
```

```
   break;

/*
 * Get the info, using longFmt set to TRUE so we can get the clip
 * names too.
 */
mza_LgCtnt_getAtr(lgCtntOR, &env, TRUE, &lcAtr);
yslPrint("John's Logical Content \n\n");
print_logical_content(&lcAtr);
yoRelease((dvoid *) lgCtntOR);

break;
 case 3:/* Destroy the Logical Content For John */
lgCtntOR = find_logical_content(JOHNS_LOGICAL_CONTENT_NAME, &env);
if (lgCtntOR == (mza_LgCtnt) CORBA_OBJECT_NIL)
   break;

    mza_LgCtnt_destroy(lgCtntOR, &env);
yslPrint("Destroyed John's Old Logical Content\n");
yoRelease((dvoid *) lgCtntOR);

break;
 case 4:/* Build Some Logical Content For Mary */
/*
 * We can't create another logical content without destroying the
 * old one first.
 */
yslPrint("Looking for Mary's Logical Content to Destroy\n");
lgCtntOR = find_logical_content(MARYS_LOGICAL_CONTENT_NAME, &env);

if (lgCtntOR != (mza_LgCtnt) CORBA_OBJECT_NIL)
{
   /*
    * If Mary already has one, destroy it first.
    */
       mza_LgCtnt_destroy(lgCtntOR, &env);
   yslPrint("Destroyed mary's Old Logical Content\n");
   yoRelease((dvoid *) lgCtntOR);
}

/*
 * Create a new Logical Content for mary
 */
yslPrint("Creating mary's New Logical Content\n");
lgCtntOR = mza_LgCtntFac_create(lgCtntFacOR, &env,
   (char*)(dvoid*)MARYS_LOGICAL_CONTENT_NAME,
   (char*)(dvoid*)"Logical Content created by OVS clip demo");
yslPrint("Created mary's New Logical Content\n");

/*
 * Do the same thing as for john, but look for Mary's movie and clip.
 */
pos = mza_LgCtnt_addClip(lgCtntOR, &env, demoCtx->maryMovieClip);
mza_LgCtnt_addClipByPos(lgCtntOR, &env, demoCtx->maryCommClip,
      (sb4)1);

/*
 * Free the Object References we used
 */
yoRelease((dvoid *) lgCtntOR);

break;
 case 5:/* Print Mary's Logical Content Info */
lgCtntOR = find_logical_content(MARYS_LOGICAL_CONTENT_NAME, &env);
if (lgCtntOR == (mza_LgCtnt) CORBA_OBJECT_NIL)
   break;

/*
```

```
      * Get the info, using longFmt set to TRUE so we can get the clip
      * names too.
      */
     mza_LgCtnt_getAtr(lgCtntOR, &env, TRUE, &lcAtr);
     yslPrint("Mary's Logical Content \n\n");
     print_logical_content(&lcAtr);
     yoRelease((dvoid *) lgCtntOR);

     break;
      case 6:/* Destroy the Logical Content For Mary */
     lgCtntOR = find_logical_content(MARYS_LOGICAL_CONTENT_NAME, &env);
     if (lgCtntOR == (mza_LgCtnt) CORBA_OBJECT_NIL)
        break;

         mza_LgCtnt_destroy(lgCtntOR, &env);
     yslPrint("Destroyed Mary's Old Logical Content\n");
     yoRelease((dvoid *) lgCtntOR);

     break;
      case 99:

     yoEnvFree(&env);
     yoRelease((dvoid *) lgCtntFacOR);
     return;
      default:
     yslPrint("Invalid selection entered. Try Again\n");
         break;
      }
   }
}
/*
 * find_logical_content
 *
 * Given a LgCtnt Name, find a LgCtnt OR.
 */

static          mza_LgCtnt
find_logical_content(const char *name, yoenv * env)
{
   mza_LgCtntMgmt  lgCtntMgmtOR;
   mza_LgCtnt      lgCtntOR;
   mza_Itr         itr;
   mza_LgCtntAtrLst lcAtrLst;

   lgCtntMgmtOR = (mza_LgCtntMgmt) yoBind(mza_LgCtntMgmt__id, (char *) 0,
             (yoRefData *) 0, (char *) 0);

   /*
    * Search for the Logical Content
    */
   itr.Position = 0;
   itr.NumItems = 1;
   lcAtrLst = mza_LgCtntMgmt_lstAtrByNm(lgCtntMgmtOR, env,
       (char*)(dvoid*)name, FALSE, &itr);
   if (lcAtrLst._length != 1)
   {
      yslPrint("Could not find Logical Content: <%s>\n",name);
      lgCtntOR = (mza_LgCtnt) CORBA_OBJECT_NIL;
   }
   else
   {
      lgCtntOR = (mza_LgCtnt) yoDuplicate((dvoid *)
        lcAtrLst._buffer[0].lgCtntOR);
   }
   yoRelease((dvoid *) lgCtntMgmtOR);

   mza_LgCtntAtrLst__free(&lcAtrLst, yoFree);
```

```
   return lgCtntOR;
}
/*
 * find_clip
 *
 * Given a Clip Name, find a Clip OR.
 */
static          mza_Clip
find_clip(const char *name, yoenv * env)
{
   mza_ClipMgmt     clipMgmtOR;
   mza_Clip         clipOR;
   mza_Itr          itr;
   mza_ClipAtrLst clAtrLst;

   clipMgmtOR = (mza_ClipMgmt) yoBind(mza_ClipMgmt__id, (char *) 0,
             (yoRefData *) 0, (char *) 0);

   /*
    * Search for the Clip
    */
   itr.Position = 0;
   itr.NumItems = 1;
   clAtrLst = mza_ClipMgmt_lstAtrByNm(clipMgmtOR, env,
        (char*)(dvoid*)name, &itr);
   if (clAtrLst._length != 1)
   {
      yslPrint("Could not find Clip: <%s>\n",name);
      clipOR = (mza_Clip) CORBA_OBJECT_NIL;
   }
   else
   {
      clipOR = (mza_Clip) yoDuplicate((dvoid *)
        clAtrLst._buffer[0].clipOR);
   }
   yoRelease((dvoid *) clipMgmtOR);

   mza_ClipAtrLst__free(&clAtrLst, yoFree);

   return clipOR;
}
/*
 * find_ctnt
 *
 * Given a Ctnt Name, find a Ctnt OR.
 */
static          mza_Ctnt
find_ctnt(const char *name, yoenv * env)
{
   mza_CtntMgmt     ctntMgmtOR;
   mza_Ctnt         ctntOR;
   mza_Itr          itr;
   mza_CtntAtrLst ctAtrLst;

   ctntMgmtOR = (mza_CtntMgmt) yoBind(mza_CtntMgmt__id, (char *) 0,
             (yoRefData *) 0, (char *) 0);

   /*
    * Search for the Ctnt
    */
   itr.Position = 0;
   itr.NumItems = 1;
   ctAtrLst = mza_CtntMgmt_lstAtrByNm(ctntMgmtOR, env,
        (char*)(dvoid*)name, &itr);
   if (ctAtrLst._length != 1)
   {
      yslPrint("Could not find Ctnt <%s>\n",name);
      ctntOR = (mza_Ctnt) CORBA_OBJECT_NIL;
```

```
   }
   else
   {
      ctntOR = (mza_Ctnt) yoDuplicate((dvoid *)
        ctAtrLst._buffer[0].ctntOR);
   }
   yoRelease((dvoid *) ctntMgmtOR);

   mza_CtntAtrLst__free(&ctAtrLst, yoFree);

   return ctntOR;
}
/*
 * checkUsingDB
 *
 * Check to see if we are using a DB enable vscontsrv
 */
static          boolean
checkUsingDB(yoenv * env)
{
   mza_LgCtntMgmt  lgCtntMgmtOR;
   boolean         usingDB;

   lgCtntMgmtOR = (mza_LgCtntMgmt) yoBind(mza_LgCtntMgmt__id, (char *) 0,
             (yoRefData *) 0, (char *) 0);
   /*
    * Search for the Logical Content for John
    */
   usingDB = mza_LgCtntMgmt_usingDB(lgCtntMgmtOR, env);

   yoRelease((dvoid *) lgCtntMgmtOR);

   return usingDB;
}
static void
print_logical_content(mza_LgCtntAtr * lcAtr)
{
   int i;

   yslPrint("Logical Content Name: %s\n", lcAtr->name);
   yslPrint("Logical Content Description: %s\n", lcAtr->desc);
   yslPrint("Logical Content Total Time: %0.1f minutes\n", (double)
        lcAtr->msecs / 6000.0);
   yslPrint("Logical Content Number of Clips: %d\n", lcAtr->numClips);
   for (i = 0; i < lcAtr->numClips; ++i)
   {
      if (i == 0)
     yslPrint("Clip Info\n");
      yslPrint("Clip %d Name: %s\n", i,
           lcAtr->clipAtrLst._buffer[i].name);
   }
}
/*
 * create_clipdemo_objects - Create the Ctnt and Clip objects
 * we need for the rest of the demo.
 *
 * This function creates 4 Ctnt objects and 4 Clip Objects,
 * a movie and commercial for John, and a movie and commercial
 * for Mary.
 */
static          boolean
create_clipdemo_objects(clipDemoCtx *demoCtx)
{
   yoenv           env;
   mza_CtntFac     ctntFacOR = (mza_CtntFac)CORBA_OBJECT_NIL;
   mza_ClipFac     clipFacOR = (mza_ClipFac)CORBA_OBJECT_NIL;
   mza_CtntAtr     cAtr;
   boolean         result = TRUE;
```

```
/*
 * Make sure the context structure is cleared
 */
DISCARD memset((void*)demoCtx, (int)0, sizeof(clipDemoCtx));

yoEnvInit(&env);

/*
 * Make sure that the content manager is using the Database. If it is not,
 * then there is no way to create clips and no pint in running this demo.
 */
if (checkUsingDB(&env) == FALSE)
{
   yoEnvFree(&env);
   yslPrint("vscontsrv was started in Standalone mode ...\n");
   yslPrint("No Clip operation supported ...\n");
   return FALSE;
}


yseTry
{
    /*
 * Get the loosely bound Object References to create the Ctnt and Clip
 * Objects we will need
 */
    ctntFacOR = (mza_CtntFac) yoBind(mza_CtntFac__id, (char *) 0,
        (yoRefData *) 0, (char *) 0);
    clipFacOR = (mza_ClipFac) yoBind(mza_ClipFac__id, (char *) 0,
        (yoRefData *) 0, (char *) 0);

    /*
 * Fill in the CtntAtr structure with some interesting data. It doesn't
 * really matter what we put here since it will never be played. Normally
 * these objects will have been created using the tagger or VSM.
 */

    cAtr.ctntOR = (mza_Ctnt) CORBA_OBJECT_NIL;/* Not used in create */
    cAtr.ctntPvdrOR = (mza_CtntPvdr) CORBA_OBJECT_NIL;
    cAtr.desc = (char *)(dvoid*) "Sample Content for OVS Clip Demo";
    cAtr.createDate.mkd_wallNano = 0;
    cAtr.createDate.mkd_wallSec = 0;
    cAtr.createDate.mkd_wallMin = 0;
    cAtr.createDate.mkd_wallHour = 0;
    cAtr.createDate.mkd_wallDay = 1;
    cAtr.createDate.mkd_wallMonth = 1;
    cAtr.createDate.mkd_wallYear = 1997;
    cAtr.filename = (char *)(dvoid*) "/mds/video/oracle1.mpi";
    cAtr.len = 0;
    cAtr.msecs = 1000;
    cAtr.rate = 1024000;
    cAtr.firstTime = (sysb8) 0;
    cAtr.lastTime = (sysb8) 1000;
    cAtr.format.mkd_contFormatVendor = (char*)0;
    cAtr.format.mkd_contFormatFmt = mkd_compFormatMpeg1;
    cAtr.format.mkd_contFormatVid._length = 0;
    cAtr.format.mkd_contFormatVid._maximum = 0;
    cAtr.format.mkd_contFormatVid._buffer = (ub1*)0;
    cAtr.format.mkd_contFormatAud._length = 0;
    cAtr.format.mkd_contFormatAud._maximum = 0;
    cAtr.format.mkd_contFormatAud._buffer = (ub1*)0;
    cAtr.format.mkd_contFormatHeightInPixels = 320;
    cAtr.format.mkd_contFormatWidthInPixels = 240;
    cAtr.format.mkd_contFormatPelAspectRatio = 2;
    cAtr.format.mkd_contFormatFrameRate = 30000;
    cAtr.prohibFlags = (mkd_prohib) 0;
    cAtr.tagsFlag = TRUE;
```

```
        cAtr.multiRateFlag = FALSE;
        cAtr.reliableFlag = FALSE;
        cAtr.volLocation = (char *) NULL;
        cAtr.contStatus = mkd_contStatusDisk;
        cAtr.assigned = FALSE;
        cAtr.sugBufSz = mza_BufSzUnknown;

        /*
     * First create the 4 Ctnt objects making sure the name changes for each
     * one. Use the returned Ctnt OR for each to create a clip.
     */
        cAtr.name = (char*)(dvoid*)JOHNS_MOVIE_CTNT_NAME;
        demoCtx->johnMovieCtnt = mza_CtntFac_create(ctntFacOR, &env, &cAtr);
        cAtr.name = (char*)(dvoid*)JOHNS_COMM_CTNT_NAME;
        demoCtx->johnCommCtnt = mza_CtntFac_create(ctntFacOR, &env, &cAtr);
        cAtr.name = (char*)(dvoid*)MARYS_MOVIE_CTNT_NAME;
        demoCtx->maryMovieCtnt = mza_CtntFac_create(ctntFacOR, &env, &cAtr);
        cAtr.name = (char*)(dvoid*)MARYS_COMM_CTNT_NAME;
        demoCtx->maryCommCtnt = mza_CtntFac_create(ctntFacOR, &env, &cAtr);


        /*
     * Now create the 4 Clip Objects
     */

        demoCtx->johnMovieClip = mza_ClipFac_create(clipFacOR, &env,
         demoCtx->johnMovieCtnt,
         (char*)(dvoid*) JOHNS_MOVIE_CLIP_NAME,
         (char *)(dvoid*) "Sample Clip for OVS Clip Demo",
         (mkd_pos*)(dvoid*)&mkdBeginning,
         (mkd_pos*)(dvoid*)&mkdEnd);
        demoCtx->johnCommClip = mza_ClipFac_create(clipFacOR, &env,
         demoCtx->johnCommCtnt,
         (char*)(dvoid*) JOHNS_COMM_CLIP_NAME,
         (char *)(dvoid*) "Sample Clip for OVS Clip Demo",
         (mkd_pos*)(dvoid*)&mkdBeginning,
         (mkd_pos*)(dvoid*)&mkdEnd);
        demoCtx->maryMovieClip = mza_ClipFac_create(clipFacOR, &env,
         demoCtx->maryMovieCtnt,
         (char*)(dvoid*) MARYS_MOVIE_CLIP_NAME,
         (char *)(dvoid*) "Sample Clip for OVS Clip Demo",
         (mkd_pos*)(dvoid*)&mkdBeginning,
         (mkd_pos*)(dvoid*)&mkdEnd);
        demoCtx->maryCommClip = mza_ClipFac_create(clipFacOR, &env,
         demoCtx->maryCommCtnt,
         (char*)(dvoid*) MARYS_COMM_CLIP_NAME,
         (char *)(dvoid*) "Sample Clip for OVS Clip Demo",
         (mkd_pos*)(dvoid*)&mkdBeginning,
         (mkd_pos*)(dvoid*)&mkdEnd);

    }
    yseCatchAll
    {
       yslError("Caught Exception while creating demo objects: <%s>\n",yseExid);
       result = FALSE;
    }
    yseEnd;


    if(ctntFacOR)
     yoRelease((dvoid *) ctntFacOR);
    if(clipFacOR)
     yoRelease((dvoid *) clipFacOR);

    yoEnvFree(&env);
    return result;

}
```

```
/*
 * destroy_clipdemo_objects - destroy the objects we created earlier
 *
 */
static void
destroy_clipdemo_objects()
{
   yoenv           env;
   mza_LgCtnt      lgCtntOR;
   mza_Clip        clipOR;
   mza_Ctnt        ctntOR;

   yoEnvInit(&env);

   yseTry
   {
       /*
    * make sure that John and Mary's logical content is destroyed
    */
       lgCtntOR = find_logical_content(JOHNS_LOGICAL_CONTENT_NAME, &env);

       if (lgCtntOR != (mza_LgCtnt) CORBA_OBJECT_NIL)
       {
            mza_LgCtnt_destroy(lgCtntOR, &env);
        yslPrint("Destroyed John's Old Logical Content\n");
        yoRelease((dvoid *) lgCtntOR);
       }

       lgCtntOR = find_logical_content(MARYS_LOGICAL_CONTENT_NAME, &env);

       if (lgCtntOR != (mza_LgCtnt) CORBA_OBJECT_NIL)
       {
            mza_LgCtnt_destroy(lgCtntOR, &env);
        yslPrint("Destroyed Mary's Old Logical Content\n");
        yoRelease((dvoid *) lgCtntOR);
       }

       /*
    * The clips should have been removed from the logical content so we
    * should be able to destroy the clips and content we created earlier.
        * Search for them by name, since if we failed once before, the
        * create will have failed the next time and we want to make sure they
        * get deleted.
    */

       if((clipOR = find_clip(JOHNS_MOVIE_CLIP_NAME, &env)) !=
        (mza_Clip) CORBA_OBJECT_NIL)
       {
        mza_Clip_destroy(clipOR, &env);
            yoRelease((dvoid *) clipOR);
       }
       if((clipOR = find_clip(JOHNS_COMM_CLIP_NAME, &env)) !=
        (mza_Clip) CORBA_OBJECT_NIL)
       {
        mza_Clip_destroy(clipOR, &env);
            yoRelease((dvoid *) clipOR);
       }
       if((clipOR = find_clip(MARYS_MOVIE_CLIP_NAME, &env)) !=
        (mza_Clip) CORBA_OBJECT_NIL)
       {
        mza_Clip_destroy(clipOR, &env);
            yoRelease((dvoid *) clipOR);
       }
       if((clipOR = find_clip(MARYS_COMM_CLIP_NAME, &env)) !=
        (mza_Clip) CORBA_OBJECT_NIL)
       {
        mza_Clip_destroy(clipOR, &env);
            yoRelease((dvoid *) clipOR);
```

```
        }

        if((ctntOR = find_ctnt(JOHNS_MOVIE_CTNT_NAME, &env)) !=
         (mza_Ctnt) CORBA_OBJECT_NIL)
        {
         mza_Ctnt_destroy(ctntOR, &env, FALSE, FALSE);
             yoRelease((dvoid *) ctntOR);
        }
        if((ctntOR = find_ctnt(JOHNS_COMM_CTNT_NAME, &env)) !=
         (mza_Ctnt) CORBA_OBJECT_NIL)
        {
         mza_Ctnt_destroy(ctntOR, &env, FALSE, FALSE);
             yoRelease((dvoid *) ctntOR);
        }
        if((ctntOR = find_ctnt(MARYS_MOVIE_CTNT_NAME, &env)) !=
         (mza_Ctnt) CORBA_OBJECT_NIL)
        {
         mza_Ctnt_destroy(ctntOR, &env, FALSE, FALSE);
             yoRelease((dvoid *) ctntOR);
        }
        if((ctntOR = find_ctnt(MARYS_COMM_CTNT_NAME, &env)) !=
         (mza_Ctnt) CORBA_OBJECT_NIL)
        {
         mza_Ctnt_destroy(ctntOR, &env, FALSE, FALSE);
             yoRelease((dvoid *) ctntOR);
        }

    }
    yseCatchAll
    {
       yslError("Caught Exception while destroying demo objects: <%s>\n",
            yseExid);
    }
    yseEnd;

    yoEnvFree(&env);
    return;


}
```

dcontsrv.c

```
/* Copyright (c) 1996 by Oracle Corporation.  All Rights Reserved.
 *
 * dcontsrv.c - OVS Content Resolver Demo main routine
 *
 * This is an example of how to write a content service and
 * resolver for OVS 3.0.
 *
 * The content service provides the capability to create and query for content.
 * Some means of storing and retrieving infomation about available content
 * must be provided. The methods in the cont interface in cont.idl
 * provide the means to create new content and retrieve them at a later date.
 * This demo just uses a simple ascii file format to store the content data. In
 * actual implementations, either MDS or a database may be used to provide more
 * capability. The file format is described below. The primary function for
 * the content service is to provide asset cookies to a client. The two
 * query functions return a structure(s) which contain an asset cookie(s)
 * which may be used to call mzs_stream_prepare().
 *
 *
 *
 * A content resolver's job is to resolve a content asset cookie for the
 * stream service. The stream service is passed the asset cookie when the
 * stream is prepared. The stream service then calls the content resolver
 * to get an mkd_segmentList back. An mkd_segmentList is a list of structures
 * that describe what MDS files to play, and the information about them.
 * The content resolver's job is to create
 * this mkd_segmentList from the name it is passed. An asset cookie has a
 * specific format. The format is <ImplId>:<AssetCookieName>. The ImplId part
 * is the Impl Id of the content resolver. In this case it is equal to
 * "contDemo__id" and is stored in the global myImplId. The ImplId of the
 * resolver gets set in the call to yoSetImpl(). The AssetCookieName part
 * is opaque to everyone but the content resolver. It knows how to convert it
 * to an mkd_segmentList.
 * In this example, the AssetCookieName will just be set to a content name.
 * The ascii file containing all of the content data is scanned for the content
 * name and a segment list is created from the data in the file. The data in
 * this file contains the entries needed to create the mkd_segmentList.
 * This data would normally come from a database table, or be read directly
 * from a tagfile.
 *
 * The format of the files for this demo are ascii files, and contain one
 * line for each piece of content. Each line contains the content name followed
 * by a colon, followed by segment definitions.
 *
 *  content name:segment data 1: segment data 2: ... : segment data n
 *
 *
 * Segment data contains 6 fields defined as follows:
 * 1    char *MDS filename
 * 2    int Start Position in seconds
 * 3    int Stop Position in seconds
 * 4    charRW Flag (Y|N)
 * 5    charFF Flag (Y|N)
 * 6    charPause Flag (Y|N)
 *
 * Example:
 * Content1: /mds/video/oracle1.mpi 0 300 N N N:/mds/video/oracle2.mpi 0 300 N N N
 * Content2: /mds/video/oracle1.mpi 600 900 N N N
 *
 * Restrictions:
 *
 * This example is limited to 5 segments per piece of content.
 *
 * The name of the content file to be used is set on the command line. If no
```

```
 * filename is specified there, the default filename of 'metafile1.dat' is used.
 * If this file is not found, a FileError Exception is thrown and the program
 * will exit.
 *
 *
 */
#ifndef SYSX_ORACLE
#include <sysx.h>/* Oracle defs */
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h>/* filesystem defs */
#endif
#ifndef YS_ORACLE
#include <ys.h>/* System layer defs */
#endif
#ifndef MN_ORACLE
#include <mn.h>/* Media Net defs */
#endif
#ifndef MTUX_ORACLE
#include <mtux.h>/* mtux init */
#endif
#ifndef YO_ORACLE
#include <yo.h>/* orb defs */
#endif
#ifndef YOCOA_ORACLE
#include <yocoa.h>/* orb defs */
#endif
#ifndef MTCR_IDL
#include <mtcr.h>
#endif
#ifndef CONT_IDL
#include <cont.h>
#endif


static struct ysargmap dcontsrvArgs[] =
{
   {'f', "dcontsrv.fileName", 1},
   {0, "", 0}
};

static void     dcontsrv(void);
static void     registerIntf(void);
static void     unregisterIntf(void);


externref CONST_W_PTR struct mtcr_resolve__tyimpl mtcr_resolve__impl;
externref CONST_W_PTR struct demo_cont__tyimpl demo_cont__impl;

/* Define some implementation Id to be used by this server */
externdef char  *myImplId = "dcontsrv";

void
main(int argc, char **argv)
{
   ub1             ctxbuf[SYSX_OSDPTR_SIZE];/* global context storage
           * space */
   char            base[SYSFP_MAX_PATHLEN];

   /* get basename of program for ys */
   sysfpExtractBase(base, argv[0]);

   if (mtuxInit((dvoid *)ctxbuf, base, (mnLogger)0,
               (sword)(argc - 1), argv + 1, dcontsrvArgs) != mtuxLayerSuccess)
      exit(1);

   yseTry
   {
```

```
      dcontsrv();
   }
   yseCatchAll
   {
      yslPrint("exception thrown: %s -- exiting\n", yseExid);
   }
   yseEnd;

   mtuxTerm(ctxbuf);
   exit(0);
}
static void
dcontsrv()
{
    char *arg;
    char fileName[256];

    /*
     * Get the content file name resource from the command line
     * and store it in a Global context under the name
     * 'fileName'
     */
    if (arg = ysResGetLast("dcontsrv.fileName"))
     strcpy(fileName,arg);
    else
     strcpy(fileName,"metafile1.dat");

    yscSetKeyed(YSC_BYNAME, (dvoid *) "fileName",
        strlen("fileName"), (dvoid *) fileName);

   /*
    * Register the content interfaces
    */
   yseTry
   {
      registerIntf();
   }
   yseCatchAll
   {
      yslError("Error registering server implementations");
      return;
   }
   yseEnd;

   yslError("Content Demo Ready\n");

   /*
    * Start the service Que
    */
   yoService((ysque *) 0);

   /*
    * Unregister the content interfaces
    */
   yseTry
   {
      unregisterIntf();
   }
   yseCatchAll
   {
      yslError("Error unregistering server implementations");
      return;
   }
   yseEnd;


}
```

```
static void
registerIntf()
{

   /*
    * Notice the use of myImplId to set the impl Id the stream service will
    * formward the resolve request to and the address of the Impl struct.
    */
   yoSetImpl(demo_cont__id, myImplId,
         (yostub *) demo_cont__stubs,
         (dvoid *) & demo_cont__impl, (yoload) 0, TRUE, (dvoid *) 0);

   yoImplReady(demo_cont__id, myImplId, (ysque *) 0);

   yoSetImpl(mtcr_resolve__id, myImplId,
         (yostub *) mtcr_resolve__stubs,
         (dvoid *) & mtcr_resolve__impl, (yoload) 0, TRUE, (dvoid *) 0);

   yoImplReady(mtcr_resolve__id, myImplId, (ysque *) 0);

   return;

}

static void
unregisterIntf()
{

   yoImplDeactivate(demo_cont__id, (char *) myImplId);
   yoImplDeactivate(mtcr_resolve__id, (char *) myImplId);

   return;

}
```

contclnt.c

```
/* Copyright (c) 1996 by Oracle Corporation.  All Rights Reserved.
 *
 * contclnt.c - OVS Demo binary
 *
 * Uses sample content resolver to resolve an asset cookie
 *
 */

#ifndef SYSX_ORACLE
#include <sysx.h>                                          /* Oracle defs */
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h>                                       /* filesystem defs */
#endif
#ifndef SYSB8_ORACLE
#include <sysb8.h>                                      /* eight-byte types */
#endif
#ifndef YS_ORACLE
#include <ys.h>                                        /* System layer defs */
#endif
#ifndef YSR_ORACLE
#include <ysr.h>                                        /* resource db defs */
#endif
#ifndef MN_ORACLE
#include <mn.h>                                           /* Media Net defs */
#endif
#ifndef MTUX_ORACLE
#include <mtux.h>                                         /* Media Net init */
#endif
#ifndef YO_ORACLE
#include <yo.h>                                 /* orb defs */
#endif
#ifndef MZCCH_IDL
#include <mzcch.h>                              /* channel interface */
#endif
#ifndef MZC_IDL
#include <mzc.h>                                /* circuit interface */
#endif
#ifndef MZZ_IDL
#include <mzz.h>                                /* session interface */
#endif
#ifndef MKD_IDL
#include <mkd.h>                                /* common datatypes */
#endif
#ifndef MKDC_ORACLE
#include <mkdc.h>                               /* C constants for mkd */
#endif
#ifndef MZS_IDL
#include <mzs.h>                                /* stream interface */
#endif
#ifndef CONT_IDL
#include <cont.h>                             /* demo content interface */
#endif

/* FORWARD DEFINITIONS AND FUNCTION PROTOTYPES */

static struct ysargmap contclntArgs[] =
{
  { 'b', "contclnt.bitrate", 1 },
  { 'c', "contclnt.control-address", 1 },
  { 'C', "contclnt.control-protocol", 1 },
  { 'd', "contclnt.data-address", 1 },
  { 'D', "contclnt.data-protocol", 1 },
  { 'i', "contclnt.clientDeviceId", 1 },
```

```
  { YSARG_PARAM, (char *)"contclnt.contentName", 1 },
  { 0, "", 0 }
};


static void contclnt(const char *id, const char *cproto, const char *ctrl,
            const char *dproto, const char *data,
            ub4 bitrate, const char *tagfile);

static void contclntSession(yoenv *env, const char *id,
              const char *cproto, const char *ctrl,
              const char *dproto, const char *data,
              ub4 bitrate,
              mzz_session *ses, mzc_circuit *dcirc );

static void contclntStream( mzc_circuit *circ, yoenv *env,
              const char *tagfile, ub4 bitrate );

void main(int argc, char **argv)
{
  ub1  ctxbuf[SYSX_OSDPTR_SIZE];             /* global context storage space */
  char base[SYSFP_MAX_PATHLEN];
  char  *arg;
  char  *id;
  char  *cproto;
  char  *ctrl;
  char  *dproto;
  char  *data;
  char  *contentName;
  ub4   bitrate;

  /* get basename of program for ys */
  sysfpExtractBase( base, argv[0] );

  if (mtuxInit((dvoid *)ctxbuf, base, (mnLogger)0,
               (sword)(argc - 1), argv + 1, contclntArgs) != mtuxLayerSuccess)
    exit( 1 );

  /* maximum bitrate that the client can support */
  arg = ysResGetLast( "contclnt.bitrate" );
  if (!arg)
    arg = (char *)"2048000";                                    /* 2 Mbps */
  bitrate = (ub4) atoi( arg );

  /* the client device id must be unique across all sessions in the system */
  id = ysResGetLast( "contclnt.clientDeviceId" );
  if (!id)
    id = (char *)"demo";

  /*
   * control protocol and address of the control (Media Net) circuit.
   * it isn't really used for anything.
   */
  cproto = ysResGetLast( "contclnt.control-protocol" );
  if (!cproto)
    cproto = (char *)"UDP";
  ctrl = ysResGetLast( "contclnt.control-address" );
  if (!ctrl)
    ctrl = (char *)"127.0.0.1:0";

  /* data protocol and address of that the video pump will send data to */
  dproto = ysResGetLast( "contclnt.data-protocol" );
  if (!dproto)
    dproto = (char *)"UDP";
  data = ysResGetLast( "contclnt.data-address" );
  if (!data)
    data = (char *)"127.0.0.1:2856";
```

```
  /*
   * name of the content to play. This will be retrieved
   * from the content service
   */
  contentName = ysResGetLast( "contclnt.contentName" );

  yseTry
  {
    contclnt( id, cproto, ctrl, dproto, data, bitrate, contentName );
  }
  yseCatchAll
  {
    yslPrint( "exception thrown: %s -- exiting\n", yseExid );
  }
  yseEnd;

  mtuxTerm((dvoid *)ctxbuf);
  exit(0);
}

/* contclnt - main routine */
static void contclnt( const char *id, const char *cproto, const char *ctrl,
            const char *dproto, const char *data,
            ub4 bitrate, const char *contentName )
{
  yoenv       env;                                /* active ORB environment */
  mzz_session ses;                                        /* session object */
  mzc_circuit dcirc;                            /* downstream circuit object */

  /* initialize ORB environment */
  yoEnvInit( &env );

  /*
   * Estable session with Oracle Video Server and acquire a
   * high-speed data circuit for video
   */
  contclntSession( &env, id, cproto, ctrl, dproto, data, bitrate,
       &ses, &dcirc);

  /*
   * Example Video Stream operations
   */
  contclntStream( &dcirc, &env, contentName, bitrate );

  /*
   * Clean up session resources and exit
   */
  yslPrint( "releasing session ... " );
  mzz_ses_Release( ses.or, &env );
  yslPrint( "released.\n" );

  mzz_session__free( &ses, yoFree );
  mzc_circuit__free( &dcirc, yoFree );

  yoEnvFree( &env );

  /* the end */
  yslPrint( "Content demo exiting successfully ...\n" );
  return;
}

/*
 * contclntSession - set up session and high-speed downstream circuit
 *
 * env      is the active environment (INOUT)
 * id       is the client device ID to use (IN)
 * cproto   is the control circuit network protocol (IN)
```

```
 * ctrl     is the control circuit network address (IN)
 * dproto   is the data circuit network protocol (IN)
 * data     is the data circuit network address (IN)
 * bitrate  is the bitrate for the data circuit (IN)
 * ses      is the returned session object (OUT)
 * dcirc    is the returned data circuit object (OUT)
 */
static void contclntSession( yoenv *env, const char *id,
               const char *cproto, const char *ctrl,
               const char *dproto, const char *data,
               ub4 bitrate,
               mzz_session *ses, mzc_circuit *dcirc )
{
  mzz_factory zfac;
  mzc_clientDeviceId cid;
  mzc_cktspec  spec;
  mzc_commProperty props;

  /* bind to session factory */
  zfac = (mzz_factory) yoBind( mzz_factory__id, (char *)0, (yoRefData *)0,
                (char *)0 );

  /*
   * set client device id - must be unique across the system.
   * It is convenient, but not necessary (??) to make the device id be
   * a user-printable string
   */
  cid._length = cid._maximum = (ub4) (strlen(id)+1);
  cid._buffer = (ub1 *)id;

  /*
   * build control circuit for session - a :persistant" point-to-point
   * bidirectional channel (no bitrate because it isn't used for anything).
   */
  props = mzc_propDown | mzc_propUp | mzc_propPointcast | mzc_propControl |
    mzc_propPersistantConnect;
  spec._d = mzc_cktspecTypeRequest;
  spec._u.req._d = mzc_cktreqTypeSymmetric;
  spec._u.req._u.sym.props = props;
  spec._u.req._u.sym.chnl._d = mzc_chnlspecTypeRequest;
  spec._u.req._u.sym.chnl._u.req.props = props;
  spec._u.req._u.sym.chnl._u.req.protocol.name = (char *)cproto;
  spec._u.req._u.sym.chnl._u.req.protocol.info = (char *)ctrl;
  spec._u.req._u.sym.chnl._u.req.bitrate = 0;


  /* invoke call - nothing special about this session  */
  yslPrint( "allocating session (id %s, addr %s:%s) ... ", id, cproto, ctrl );
  *ses = mzz_factory_AllocateSession(zfac, env, mzz_sessNull, &cid, &spec );
  yslPrint( "allocated.\n" );

  /* release mzz factory object - no longer needed */
  yoRelease( (dvoid *)zfac );

  /*
   * build data circuit - asymmetric "persistant" point-to-point
   * downstream-only real time channel from the video pump to the client
   */
  props = mzc_propDown | mzc_propPointcast | mzc_propData |
    mzc_propIsochronousData | mzc_propPersistantConnect;
  spec._d = mzc_cktspecTypeRequest;
  spec._u.req._d = mzc_cktreqTypeAsymmetric;
  spec._u.req._u.asym.props = props;
  spec._u.req._u.asym.upchnl._d = mzc_chnlspecTypeNone;
  spec._u.req._u.asym.upchnl._u.none = 0;
  spec._u.req._u.asym.downchnl._d = mzc_chnlspecTypeRequest;
  spec._u.req._u.asym.downchnl._u.req.props = props;
  spec._u.req._u.asym.downchnl._u.req.protocol.name = (char *)dproto;
```

```
  spec._u.req._u.asym.downchnl._u.req.protocol.info = (char *)data;
  spec._u.req._u.asym.downchnl._u.req.bitrate = bitrate;

  /* invoke call */
  yslPrint( "adding downstream circuit ( addr %s:%s, bitrate %d) ... ",
       dproto, data, bitrate );
  *dcirc = mzz_ses_AddCircuit( ses->or, env, &spec );
  yslPrint( "added.\n" );
}



/*
 * contclntStream - demonstrate stream service calls
 */
static void contclntStream( mzc_circuit *circ, yoenv *env,
              const char *contentName, ub4 bitrate )
{

  mzs_factory sfac;
  mzs_stream strm;
  mzs_stream_instance inst;
  mzs_capMask caps;
  mkd_segInfoList status;
  demo_cont   demoOR;
  demo_contAtr contAtr;

  sysb8 tm;

  /*
   * allocate a stream - specify the capabilities and bind w/ the circuit's
   * full bitrate.
   */

  /* bind stream factory object */
  sfac = (mzs_factory) yoBind( mzs_factory__id, (char *)0, (yoRefData *)0,
                 (char *)0 );

  /* indicate that we can receive audio and MPEG-II video
   * and can pause and blindly seek but not scan (rate control) */
  caps = mzs_capAudio | mzs_capVideo | mzs_capMpeg1 | mzs_capMpeg2 | mzs_capSeek |
    mzs_capPause;

  yslPrint("allocating stream ... ");
  strm = mzs_factory_alloc( sfac, env, circ, caps, bitrate );
  yslPrint("allocated\n");

  /* release mzs factory object (no longer needed) */
  yoRelease((dvoid *)sfac);



  /*
   * Call the content service to get an asset cookie for the named content
   */
   demoOR = (demo_cont) yoBind( demo_cont__id, (char *)0, (yoRefData *)0,
                  (char *)0 );
   contAtr = demo_cont_queryByNm(demoOR, env, (char*)contentName, FALSE);

   yoRelease((dvoid*)demoOR);

  /* prepare stream to play from beginning to end at the given bitrate.
   * playNow starts the movie immediately (no separate play cmd necessary) */
  yslPrint( "preparing stream ( playing immediately ) ... (cookie=%s)",
           contAtr.cookie );
  inst = mzs_stream_prepare( strm, env, contAtr.cookie,
                (mkd_pos *)&mkdBeginning,
```

```
                (mkd_pos *)&mkdEnd, bitrate,
                mzs_stream_playNow,
                &status,
                (dvoid *)0);
  yslPrint( "prepared.\n" );

  /* throw away status info */
  mkd_segInfoList__free( &status, yoFree );

  /* at this point we should be receiving video. Since this is a demo,
   * we don't do anything with it, but merely sleep for a while */
  yslPrint( "sleeping for 10 seconds\n" );
  sysb8ext( &tm, 10000000 );
  ysTimer( &tm, (ysevt *)0 );

  /* here's how to pause a stream (if you're curious) */
  yslPrint( "pausing stream ... " );
  mzs_stream_play(strm, env, inst,
         (mkd_pos *)&mkdCurrent, (mkd_pos *) &mkdCurrent,
         (mkd_pos *)&mkdCurrent, mzs_stream_ratePause, bitrate);
  yslPrint( "paused\n" );

  /* wait some more */
  yslPrint( "sleeping for 3 seconds\n" );
  sysb8ext( &tm, 3000000 );
  ysTimer( &tm, (ysevt *)0 );

  /* ... and how to resume it */
  yslPrint( "resuming stream ... " );
  mzs_stream_play( strm, env, inst, (mkd_pos *)&mkdCurrent,
          (mkd_pos *)&mkdCurrent, (mkd_pos *)&mkdEnd,
          mzs_stream_rate1x, bitrate );
  yslPrint( "resumed\n" );

  /* wait a little more */
  yslPrint( "sleeping for 5 seconds\n" );
  sysb8ext( &tm, 3000000 );
  ysTimer( &tm, (ysevt *)0 );

  /* stop the stream */
  yslPrint( "finishing stream ... " );
  mzs_stream_finish( strm, env, inst, mzs_stream_finishOne );
  yslPrint( "finished.\n" );

  yslPrint( "deallocating stream ... " );
  mzs_stream_dealloc( strm, env );
  yslPrint( "deallocated.\n" );

  return;
}
```

eclisten.c

```
/*------------------------------ eclisten.c --------------------------------*/
/* File: eclisten.c - Sample listener for the stream event channel          */
/*                                                                          */
/* Description:                                                             */
/*   This implements a sample listener for the stream event channel.        */
/*   This implementation prints the events as they appear, but a real-world */
/*   application could use this information for monitoring, billing, etc.   */
/*                                                                          */
/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/
/* Oracle Corporation     */
/* Oracle Media Server (TM)     */
/* All Rights Reserved                                                      */
/* Copyright (C) 1993-1997     */
/*--------------------------------------------------------------------------*/

/*------------------------------ Includes ----------------------------------*/

#ifndef SYSX_ORACLE
# include <sysx.h>              /* Generally useful defines */
#endif

#ifndef YO_ORACLE
#include <yo.h>                 /* The object layer for networking */
#endif

#ifndef MTUX_ORACLE
#include <mtux.h>               /* Media Net initialization/teardown */
#endif

#ifndef YOCOA_ORACLE
#include <yocoa.h>              /* Common Object Adapter for networking */
#endif

#ifndef YECEVCH_ORACLE
#include <yecevch.h>            /* Event channel specific information */
#endif /* !YECEVCH_ORACLE */

#ifndef YDNMIDL_ORACLE
#include <ydnmidl.h>            /* CORBA naming service */
#endif /* !YDNMIDL_ORACLE */

#ifndef MZSEC_ORACLE
#include <mzsec.h>              /* Header describing the mzs events */
#endif /* !MZSEC_ORACLE */

#ifndef ORASTDIO
#define ORASTDIO
#include <stdio.h>              /* For providing output */
#endif /* !ORASTDIO */

/*--------------------------------------------------------------------------*/
/* This is the name of the mzs event channel.                               */
/*--------------------------------------------------------------------------*/
#define MZS_EVENT_CHANNEL     "MZS_EventChannel"

/*------------------------------ Exceptions --------------------------------*/

extern ysidDecl(TMZSECL_EX_FAILURE) = "cli_EventChannel::failure";
externdef ysidDecl(mzs_ec_implid) = "MZS_EventChannel";

/*-------------------------------- Types -----------------------------------*/

/* This event channel context is used to hold interesting pointers and */
```

```
/* values while the listener is running. */
typedef struct eventChannel
{
  yeceCa_ProxyPushSupp   orTPS;
  mzs_ec                 orCUE;
  ysque                 *ImplCliQueEC;
  boolean                valid;
} eventChannel;

/*----------------------- Private Function Prototypes -----------------------*/

/* Establish ourselves as an implementor of the event channel consumer */
/* interface. */
static void SetupEventChannel(void);

/* Attach to the stream event channel. */
static void AttachToChannel(eventChannel  *ecP);

/* Detach from the stream event channel. */
static void DetachFromChannel(eventChannel  *ecP);

/* Perform operations at idle time - this is where we actually process */
/* incoming events. */
static void IdleFunc(void *usrp, const ysid *exid, void *arg,
                     size_t argsz);

/* Attach our consumer to a supplier who will provide events. */
static boolean AttachConsumer(eventChannel  *ecP);

/* Detach our consumer from the supplier at the other end. */
static void DetachConsumer(eventChannel  *ecP);

/* Find the named typed event channel */
static yeceTeca_TypedEventChannel GetEventChannel(void);

/* The function that actually processes incoming events. */
static void SendEvent_i(mzs_ec or, yoenv* ev, mzs_event *ptev);

/* A utility function to output a circuit in a user-readable manner. */
static void CircuitToString(mzc_circuit *ckt, char *buf);

/* We declare an implementation with this one function in it */
extern const struct mzs_ec__tyimpl mzs_ec__impl =
{
  SendEvent_i
};

/*---------------------------------- main ----------------------------------*/
/* This is the main routine for the event channel listener.                 */
/*--------------------------------------------------------------------------*/
int main(int argc, char *argv[])
{
  ub1           osdp[SYSX_OSDPTR_SIZE] = {0};
  eventChannel  ecV;

  if (mtuxSimpleInit(osdp, "Event Channel Listener", (mnLogger)0) !=
      mtuxLayerSuccess)
    return 1;

  /* Setup the Event Channel */
  SetupEventChannel();

  yseTry
  {
    (void)printf("Sample Listener: started...\n");

    /* Attach to the event channel as a client */
    AttachToChannel(&ecV);
```

```
    while(1)
    {
       ysYield();       /* Just wait for things to happen */
    }
  }
  yseCatchAll
  {
    (void)fprintf(stderr, "Exception received when listening: %s\n",
                  ysidToStr(yseExid));
  }
  yseEnd;

  mtuxTerm(osdp);

  return 0;
}

/*-------------------------------------------------------------------------
// Name: SetupEventChannel
// Function:
//   Our goal is to implement the interface that an event supplier calls as
//   events occur.  This function does the object layer operations necessary
//   to prepare our implementation of the event channel listener.
// Input:
//   None
// Output:
//   None
// Returns:
//   Nothing
// Raises:
//   Nothing
//-------------------------------------------------------------------------*/
static void SetupEventChannel()
{
  /* Tell the object layer about this implementation of the event channel */
  /* listener.                                                            */
  yoSetImpl(mzs_ec__id, mzs_ec_implid,
            mzs_ec__stubs,
            (dvoid*) &mzs_ec__impl,
            (yoload)0, FALSE, (dvoid *) 0);
}

/*-------------------------------------------------------------------------
// Name: AttachToChannel
// Function:
//   This function has several goals.  First, it creates a queue where our
//   incoming calls can be staged.  Next, it makes a call to try to attach
//   ourselves as a consumer of the stream event channel.  If this is
//   successful, we make sure we will get called on a regular basis to try
//   to process incoming events.
// Input:
//   A pointer to our event channel context
// Output:
//   Fields in our context are set for later use
// Returns:
//   Nothing
// Raises:
//   Errors that are thrown from below this function are reraised to the
//   caller.
//-------------------------------------------------------------------------*/
static void AttachToChannel(eventChannel  *ecP)
{
  yseTry
  {
    /* Create a queue for handling incoming events */
    ecP->ImplCliQueEC = yoQueCreate("CliEC");
```

```
    /* Get ourselves attached to the stream event channel. */
    AttachConsumer(ecP);

    /* Tell the network layer that we occasionally want our IdleFunc to */
    /* be called and passed a pointer to our event channel. */
    ysSetIdler("IdleFunc()", IdleFunc, (void *)ecP);
  }
  yseCatchAll
  {
    /* Something has gone wrong... */
    /* Clean up and reraise the error */
    DetachFromChannel(ecP);

    yseRethrow;
  }
  yseEnd;
}

/*-----------------------------------------------------------------------------
// Name: AttachConsumer
// Function:
//   To successfully listen on the event channel, we need to establish a
//   dialogue between a supplier that will push the events and a consumer of
//   our own making.  This takes several steps, as described below.  If this
//   function succeeds, we will have created our consumer and told a supplier
//   to hand it events as they occur.
// Input:
//   A pointer to our event channel context.
// Output:
//   Several object references will be saved in the context for use later.
// Returns:
//   TRUE if all went well.
// Raises:
//   Errors that occur in layers beneath us are reraised.
//---------------------------------------------------------------------------*/
static boolean AttachConsumer(eventChannel  *ecP)
{
  yeceCa_ProxyPushSupp          orTPS = NULL;
  yeceTeca_TypedConsAdm         orTCA = NULL;
  yeceTeca_TypedEventChannel    orTEC = NULL;
  mzs_ec                        orCUE = NULL;
  yoenv                         evEC;

  yoEnvInit(&evEC);

  yseTry
  {
    /* We try to find an object reference to the named typed event channel */
    /* where the stream events get published. */
    orTEC = GetEventChannel();

    /* We express particular interest in this channel as someone who will */
    /* be consuming events. */
    orTCA = (yeceTeca_TypedConsAdm)
      yeceTeca_TypedEventChannel_for_consumers(orTEC, &evEC);

    /* We now make sure there is a supplier who is willing to push events */
    /* into our implementation of an event channel listener. */
    orTPS = (yeceCa_ProxyPushSupp)
      yeceTeca_TypedConsAdm_obtain_typed_push_supp(orTCA, &evEC,
                                                   (char *)mzs_ec__id);

    /* We stash a pointer to this supplier - we'll need it later */
    ecP->orTPS = orTPS;

    /* We're about ready, so tell the object layer we're ready to take */
    /* requests. */
    yoImplReady(mzs_ec__id, mzs_ec_implid, (void*)0);
```

```
    /* Instantiate our listener */
    orCUE = (mzs_ec) yoCreate(mzs_ec__id,
                              mzs_ec_implid,
                              (yoRefData*)0, (char*)0,
                              NULL);

    /* Actually listen using it */
    yoObjListen(orCUE, ecP->ImplCliQueEC);

    /* Stash this pointer - we'll need that later too... */
    ecP->orCUE = orCUE;

    /* We finish the loop - we want our supplier to push to our listening */
    /* implementation... */
    yeceCa_ProxyPushSupp_connect_push_cons(orTPS, &evEC,
                                           (struct YCyecec_PushCons *)orCUE);

    /* We note that all is well at this point */
    ecP->valid = TRUE;
  }
  yseCatchAll
  {
    /* Bummer.  We were able to successfully establish a dialogue between */
    /* a supplier who will push us events and our listener.  We need to   */
    /* clean up anything we managed to do before the problem arose.       */

    /* If we have a typed push supplier, clean it up. */
    if(orTPS != NULL)
    {
      yoRelease(orTPS);
      orTPS = ecP->orTPS = NULL;
    }

    /* If we have a typed consumer admin object, clean it up. */
    if(orTCA != NULL)
    {
      yoRelease(orTCA);
      orTCA = NULL;
    }

    /* If we have a reference to the typed event channel, clean it up. */
    if(orTEC != NULL)
    {
      yoRelease(orTEC);
      orTEC = NULL;
    }

    /* If we instantiated our listener, clean that up. */
    if(orCUE != NULL)
    {
      yoRelease(orCUE);
      orCUE = ecP->orCUE = NULL;
    }

    /* Free our environment that we used for calls */
    yoEnvFree(&evEC);

    yseRethrow;
  }
  yseEnd;

  /* If we got this far, all is well.  Clean up the things we used */
  /* in the process of setting up this connection. */
  yoRelease(orTEC);
  yoRelease(orTCA);

  yoEnvFree(&evEC);     /* Free the environment - we're done with it. */
```

```
  return TRUE;
}

/*-----------------------------------------------------------------------------
// Name: GetEventChannel
// Function:
//   This function is used to query the name service in search of the typed
//   event channel where stream events are published.  When a stream service
//   starts, it checks to see if another stream service has previously
//   created such a channel.  If not, it does so itself.  Whichever stream
//   service creates this typed event channel publishes it in the CORBA name
//   server under a well known name.
// Input:
//   None
// Output:
//   None
// Returns:
//   An object reference to the stream typed event channel.
// Raises:
//   TMZSECL_EX_FAILURE This indicates that noone has created the event
//                      channel yet.
//   Other exceptions could be thrown from below, and are reraised to the
//   caller.
//-----------------------------------------------------------------------------*/
static yeceTeca_TypedEventChannel GetEventChannel(void)
{
  ydnmInitialNamingContext    orINC = NULL;
  ydnmNamingContext           orNC = NULL;
  yeceTeca_Factory            orTECF = NULL;
  yeceTeca_TypedEventChannel  orTEC = NULL;
  ydnmName                    name;
  ydnmNameComponent           name_comp[2];
  yoenv                       evNM;

  yoEnvInit(&evNM);

  /* Find the naming service.  We'll need this to find the named */
  /* event channel where the events appear. */
  orINC = (ydnmInitialNamingContext)
    yoBind(ydnmInitialNamingContext__id,
           0,
           (yoRefData*)0, (char*)0);

  /* Now that we found one, establish a session with it */
  orNC = ydnmInitialNamingContext_get(orINC, &evNM);

  /* Setup the name - we are looking for a specific named event channel */
  name_comp[0].kind = name_comp[1].kind = (char *)0;
  name_comp[0].id   = MZS_EVENT_CHANNEL;

  name._length = name._maximum = 1;
  name._buffer = name_comp;

  yseTry
  {
    /* Ask the name service to give us a reference to the correct typed */
    /* event channel. */
    orTEC = (yeceTeca_TypedEventChannel)
      ydnmNamingContext_resolve(orNC, &evNM, &name);
  }
  yseCatchObj(EX_YDNMNOTFOUND, ydnmNotFound, nfi)
  {
    /* The event channel is not found in the naming service */
    orTEC = NULL;

    /* If we got a naming context, clean it up. */
    if ( orNC != NULL )
```

```
    {
      yoRelease(orNC); orNC = NULL;
    }

    /* If we have a loosely bound reference to the naming service, */
    /* clean that up. */
    if(orINC != NULL)
    {
      yoRelease(orINC);
      orINC = NULL;
    }

    yoEnvFree(&evNM);

    yseThrow(TMZSECL_EX_FAILURE);
  }
  yseCatchAll
  {
    /* Something unknown and unfortunate happened */

    /* If we got a naming context, clean it up. */
    if(orNC != NULL)
    {
      yoRelease(orNC);
      orNC = NULL;
    }

    /* If we have a loosely bound reference to the naming service, */
    /* clean that up. */
    if(orINC != NULL)
    {
      yoRelease(orINC);
      orINC = NULL;
    }

    yoEnvFree(&evNM);

    yseRethrow;
  }
  yseEnd;

  /* If we got to this point, we created a bunch of objects.  Clean */
  /* up after ourselves... */
  yoRelease(orNC);
  yoRelease(orINC);

  yoEnvFree(&evNM);

  /* Return the typed event channel */
  return orTEC;
}

/*-----------------------------------------------------------------------------
// Name: DetachFromChannel
// Function:
//   This is a cleanup function to be called when we are done processing
//   events.  It removes us from the idle queue so we aren't run any longer,
//   it cleans up the queue we created for incoming calls, and it makes a
//   call to detach our consumer from the supplier at the other end.
// Input:
//   A pointer to our event channel context
// Output:
//   Fields in the context are updated
// Returns:
//   Nothing
// Raises:
//   Nothing
//-----------------------------------------------------------------------------*/
```

```
static void DetachFromChannel(eventChannel *ecP)
{
  /* We no longer need to run */
  ysSetIdler("IdleFunc()", (ysHndlr) 0, (dvoid *) 0);

  /* If we created a queue for handling events, clean it up */
  if(ecP->ImplCliQueEC != NULL)
  {
    yoQueDestroy(ecP->ImplCliQueEC);
    ecP->ImplCliQueEC = NULL;
  }

  /* Detach our consumer from the supplier */
  DetachConsumer(ecP);
}

/*---------------------------------------------------------------------------
// Name: DetachConsumer
// Function:
//   This is a cleanup function to be called when we no longer wish to be
//   supplied with events from the event channel.  It basically undoes what
//   we did in AttachConsumer().  It disconnects us from the supplier at the
//   other end, deactivates our implementation of the consumer interface, and
//   cleans up some object references we no longer need.
// Input:
//   A pointer to our event channel context
// Output:
//   Fields in the context are updated
// Returns:
//   Nothing
// Raises:
//   If errors are raised from below, this function will reraise them.
//---------------------------------------------------------------------------*/
static void DetachConsumer(eventChannel *ecP)
{
  yoenv  evEC;

  /* We were never set up to begin with. */
  if(ecP->valid == FALSE)
    return;

  /* Set up our environment for making calls */
  yoEnvInit(&evEC);

  yseTry
  {
    /* Tell the supplier we're going away */
    yeceCa_ProxyPushSupp_disconnect_push_supp(ecP->orTPS, &evEC);

    /* Stop our implementation of the consumer interface */
    yoImplDeactivate(mzs_ec__id, mzs_ec_implid);


    /* Release our reference to the typed push supplier */
    if(ecP->orTPS != NULL)
    {
      yoRelease(ecP->orTPS);
      ecP->orTPS = NULL;
    }

    /* Release our instantiation of our consumer */
    if(ecP->orCUE != NULL)
    {
      yoRelease(ecP->orCUE);
      ecP->orCUE = NULL;
    }
  }
  yseCatchAll
```

```
  {
    /* If something goes wrong, not much to do.  Clean up and rethrow. */
    yoEnvFree(&evEC);
    yseRethrow;
  }
  yseEnd;

  /* Free up our environment */
  yoEnvFree(&evEC);
}


/*------------------------------------------------------------------------
// Name: IdleFunc
// Function:
//   This is the real workhorse of the listener.  At startup, we install this
//   function as a ys idler.  This ensures that as long as the process gets
//   idled, which it must to work, we will regularly get a chance to process
//   incoming events.  The main body of the program does nothing but idle, so
//   we get ample opportunity to work.
// Input:
//   usrp       A pointer to our event channel context.
//   exid
//   arg
//   argsz
// Output:
//   Fields in the context could be updated.
// Returns:
//   Nothing
// Raises:
//   Errors that occur in layers beneath us are reraised.
//------------------------------------------------------------------------*/
static void IdleFunc(void *usrp, const ysid *exid, void *arg,
                     size_t argsz)
{
  eventChannel  *ecP = (eventChannel  *)usrp;

  if(exid)
  {
    /* We should deal with shutdown here, but we don't... */
    return;
  }

  /* If events have come in to our implementation of the consumer, process */
  /* them! */
  if(ysSvcPending(ecP->ImplCliQueEC))
    ysSvcAll(ecP->ImplCliQueEC);
}

/*------------------------------------------------------------------------
// Name: SendEvent_i
// Function:
//   At long last, this is the function that actually receives published
//   events.  When such an event comes in, it will get passed to this function
//   in a timely manner.  This function just prints the events in a somewhat
//   structured manner, but a more sophisticated implementation could do a
//   wide variety of things.
// Input:
//   or         A pointer to the event channel object (not used)
//   ev         The execution environment (not used)
//   ptev       A pointer to the received event
// Output:
//   None
// Returns:
//   None
// Raises:
//   None
//------------------------------------------------------------------------*/
```

```
static void SendEvent_i(/* ARGUSED */mzs_ec or, /* ARGUSED */yoenv* ev,
                        mzs_event* ptev)
{

  /* We figure out what kind of event happened, and print accordingly */
  switch (ptev->_d)
  {
  case mzs_evTypeAlloc:
  {
    char            dsBuf[128];
    mzs_evAlloc    *event_allocP = &(ptev->_u.event_alloc);

    CircuitToString(&event_allocP->cliCircuit, dsBuf);
    (void)printf("Alloc: downstream=%s capMask=%d maxBitrate=%d\n",
                 dsBuf,
                 event_allocP->capabilities,
                 event_allocP->maxBitrate);
  }
    break;

  case mzs_evTypePrepare:
  {
    mzs_evPrepare *event_prepareP = &(ptev->_u.event_prepare);

    (void)printf("Prepare: cookie=%s\n",
                 event_prepareP->cookie);
  }
    break;

  case mzs_evTypeSeqPrepare:
  {
    mzs_evSeqPrepare *event_seqPrepareP = &(ptev->_u.event_seqPrepare);
    uword counter;

    (void)printf("SeqPrepare: cookies=");
    for(counter = 0; counter < event_seqPrepareP->cookies._length;
        ++counter)
    {
      mkd_assetCookie *this =
        (mkd_assetCookie *)(event_seqPrepareP->cookies._buffer[counter]);
      (void)printf("%s%s", counter ? " " : "", this);
    }
    (void)putchar((int)'\n');
  }
    break;

  case mzs_evTypePlay:
  {
    char startBuf[64];
    char endBuf[64];
    size_t ourLen = 64;
    mzs_evPlay *event_playP = &(ptev->_u.event_play);

    if(! (mkduPos2Str(startBuf, &ourLen, &event_playP->startPos)))
      DISCARD strcpy(startBuf, "???");
    ourLen = 64;
    if(! (mkduPos2Str(endBuf, &ourLen, &event_playP->endPos)))
      DISCARD strcpy(endBuf, "???");

    (void)printf("Play: startPos=%s endPos=%s playRate=%d bitRate=%d\n",
             startBuf, endBuf,
             event_playP->playRate,
             event_playP->bitRate);
  }
    break;

  case mzs_evTypeFinish:
  {
```

```
    mzs_evFinish *event_finishP = &(ptev->_u.event_finish);
    (void)printf("Finish: loopCancel=%d\n",
             event_finishP->loopCancel);
  }
    break;

  case mzs_evTypeDealloc:
  {
    char dsBuf[128];
    mzs_evDealloc *event_deallocP = &(ptev->_u.event_dealloc);

    CircuitToString(&event_deallocP->cliCircuit, dsBuf);
    (void)printf("Dealloc: downstream=%s\n", dsBuf);
  }
    break;

  case mzs_evTypeDenial:
  {
    mzs_evDenial *event_denialP = &(ptev->_u.event_denial);
    uword counter;

    (void)printf("Denial: cookies=");
    for(counter = 0; counter < event_denialP->cookies._length;
        ++counter)
    {
      mkd_assetCookie *this =
        (mkd_assetCookie *)(event_denialP->cookies._buffer[counter]);
      (void)printf("%s%s", counter ? " " : "", this);
    }
    (void)putchar((int)'\n');
  }
    break;
  }

  /* We flush stuff to stdout just to make sure we don't lose anything */
  /* if we are killed at an inopportune time and stdout was redirected */
  /* to a file. */
  fflush(stdout);
}

/*-----------------------------------------------------------------------------
// Name: CircuitToString
// Function:
//   This is a utility function used when trying to print events in a useful
//   manner.  Some of the events include mzc circuits.  This function just
//   walks down the structure in an effort to print the downstream netproto
//   in a user-friendly way.
// Input:
//   ckt        The circuit to be printed
//   buf        The buffer to hold the output string
// Output:
//   buf        A printable version of the downstream netproto
// Returns:
//   Nothing
// Raises:
//   Nothing
//-----------------------------------------------------------------------------*/

static void CircuitToString(mzc_circuit *ckt, char *buf)
{
  mzc_cktInfo  *ourInfo;
  mzc_channel  *down;
  mzc_chnlInfo *cInfo;
  mzc_link     *dLink;
  mzc_netproto *ourProto;

  ourInfo = &ckt->info;
  down = &ourInfo->downstream;
```

```
  cInfo = &down->info;
  dLink = &cInfo->comm;
  ourProto = &dLink->protocol;

  (void)sprintf(buf, "%s:%s", ourProto->name, ourProto->info);
}
```

vesdemo.c

```
/*
 * Copyright (C)1996-1997 Oracle Corporation
 *
 * File: vesdemo.c
 *
 * Function:
 *      This is a sample program to illustrate the VESAPI usages.
 * It will create a feed session for each input osf files.  This program
 * parses the demo OSF files for tag information and delivers the original
 * content and the tag information to the feed server (vsfeedsrv).
 *
 * Usage: vesdemo [-d duration] [-l length] [-v MDS_volume] input_video_files
 *      [duration] specifies the duration of the demo program in seconds.
 *                 The demo will loop the input file if necessary.
 *      [length] specifies the length of the content kept on the video
 *               server.  This is specified in seconds.
 *      [MDS_volume] location of the output feed. default is /mds/video
 *      input_video_file is the input file to the demo program.
 *
 * i.e.  vesdemo -d 600 -l 300 -v /mds/movie /tmp/oracle.osf
 * This command line would create a feed session with basename
 * /mds/movie/oracle.  This feed session would run for 10 minutes
 * (600 seconds).  The video server would keep the latest 5 minutes (-l 300)
 * on the MDS volume.
 *
 * After the session is successfully started, the output of this feed
 * can be accessed by the Oracle Video Client.
 */

#ifndef SYSX_ORACLE
# include <sysx.h>      /* Oracle Types */
#endif
#ifndef YO_ORACLE
# include <yo.h>        /* Oracle Media Net Object Layer Interface */
#endif  /* !YO_ORACLE */
#ifndef VESW_ORACLE
# include <vesw.h>      /* Video Encoding Standard API */
#endif  /* !VESW_ORACLE */
#ifndef SYSFP_ORACLE
# include <sysfp.h>     /* Host File System I/O */
#endif
#ifndef YSLST_ORACLE
# include <yslst.h>     /* link list */
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* default volume name */
#define DEFAULT_VOLUME  "/mds/video"

/*
 * default duration and length of the feed.  3-minute is minimum
 */
```

```
#define DEMO_DEFDURATION   (3 * 60)

/*
 * maximum number of feed sessions allowed to simulate.  Note that this
 * is in here just to illustrate the way to create multiple feed sessions
 * with single veswInit & veswTerm.  This number is limited by the server
 * configuration and the host system processing speed.
 */
#define DEMO_MAXFEEDSESS 2

/* DEMO_PROCESSTIME is in units of micro-second.  This parameter defines
 * the amount of data for each batch to be processed.  The default value
 * is one second worth of content.
 */
#define DEMO_PROCESSTIME    1000000

/* DEMO_REPORTTIME specified the time between status reporting.  Default
 * is 10 seconds */
#define DEMO_REPORTTIME   10000000

/*
 * argument map
 */
static struct ysargmap vesdemoArgMap[] =
{
  { 'd', "vesdemo.duration", 1},
  { 'l', "vesdemo.length",   1},
  { 'v', "vesdemo.MDS-volume", 1},
  { YSARG_PARAM, "vesdemo.video-files", YSARG_MANY},
  { 0, (char *)0, 0}
};


/* ============================================================================
 *
 * The following vesp* functions and definitions are specific for parsing
 * of OSF demo content files.  They are not intended for general use and
 * is not supported by Oracle.  The purpose of these functions is to provide
 * a testbed for the realtime feed feature of the Oracle Video Server.
 */
/* vesp context */
typedef struct vespCtx vespCtx;
/*
 * vespInit initializes stream parsing functions.  It returns a context
 * if the operation is successful otherwise it returns NULL
 */
vespCtx *vespInit();
/*
 * vespTerm releases all the resources allocated for stream parsing
 */
void vespTerm(vespCtx *cx);
/*
 * vespGetAttributes retrieves attributes about the content.
 * It returns FALSE if the operation is successful; otherwise it returns
 * TRUE.  vespGetAttributes will fail if vespInit is not called or the
 * stream pointed to by buf is not OSF format.
 * Inputs:      cx - vesp context
 *              buf - byte stream to be parsed for the attributes
 *              buflen - length of the buffer buf
 * Outputs:     height, width - picture dimension in pixels
 *              aspectRatio - aspectRatio of a pixel * 10000
 *              framePerKiloSec - frame per seconds * 1000
 *              bitrate - bitrate of the media content
 *              initData - data required by the decoder
 *              initDataLen - length of initData in byte
 */
boolean vespGetAttributes(vespCtx *cx, ub1 *buf, size_t buflen,
                          ub2 *height, ub2 *width,
```

```
                        sb4 *aspectRatio, ub4 *framePerKiloSec,
                        ub4 *bitrate, ub1 **initData, size_t *initDataLen);
/*
 * vespGetTags attempts to retrieve tags from the byte stream pointed to by
 * buf.  This function returns TRUE if this operation failed, otherwise it
 * returns FALSE.
 * Inputs:      cx - vesp context
 *              buf - byte stream to be parsed for tags
 *              buflen - length of buffer buf
 * Output:      tags - an array of tags in ves_tag format
 *              num - number of tags in the array
 */
boolean vespGetTags(vespCtx *cx, ub1 *buf, size_t buflen,
                    ves_tag **tags, ub4 *num);
/* ========================================================================= */


/*
 * main context
 */
typedef struct demoCtx
{
  sysfp    *fp_demoCtx;          /* file access pointer */
  vespCtx  *spcx_demoCtx;        /* stream parser context */
  sysb8     bytep_demoCtx;       /* bytes already processed */
  ub1      *buf_demoCtx;         /* buffer of the content */
  size_t    bflen_demoCtx;       /* length of buf pointed to by buf_demoCtx */
  /* basename for both content and tag files */
  char      baseName_demoCtx[SYSFP_MAX_PATHLEN];
  veswCtx  *veswCtx_demoCtx;     /* context obtained from veswNewFeed */
  ub4       bitrate_demoCtx;     /* bitrate of this content */
  ub4       totalTags_demoCtx;   /* total number of tags */
} demoCtx;

/*
 * Context storing the commands
 */
typedef struct demoCmds
{
  ub4      nsess_demoCmds;     /* number of requested sessions */
  ub4      asess_demoCmds;     /* number of active sessions */
  ub4      seconds_demoCmds;   /* Demo will stop when this many seconds
                                * is past. [-d] */
  ub4      length_demoCmds;    /* Length of the feed in seconds kept on
                                * the video server. [-l] */
  char    *inFiles_demoCmds[DEMO_MAXFEEDSESS]; /* input content filenames */
  char    *volName_demoCmds;   /* MDS volume for both tag and content [-v] */
  demoCtx  ctx_demoCmds[DEMO_MAXFEEDSESS];     /* individual session ctx */
  ub1      ysCtx_demoCmds[SYSX_OSDPTR_SIZE];   /* ys Context */
  boolean  intr_demoCmds;       /* interrupt signal */
} demoCmds;


/*
 * Name: demoGetCtntInfo()
 * Func: This function retrieves content information from the file specified
 *       by filename using the vespGetAttributes call.  The return information
 *       is used to setup a feed session.  This call is necessary since
 *       we don't currently known anything about the encoded content.
 *       During encoding, call to this function is not necessary.
 */
static boolean demoGetCtntInfo(demoCtx *dcx, char *filename,
                               ub2 *height, ub2 *width, sb4 *aspect,
                               ub4 *frpks, ub4 *bitrate,
                               ub1 **initData, size_t *initLen)
{
  boolean        failed=FALSE;
  CONST char    *log;
```

```
#define DEMO_BLKSZ      1024
  ub1            buf[DEMO_BLKSZ];
  size_t         len;

  /* open the file of interest */
  dcx->fp_demoCtx = sysfpOpen(filename, "r", SYSFPKIND_BINARY, &log);
  if (!dcx->fp_demoCtx)
  {
    yslError("Fail to open %s - %s.\n", filename, log);
    return TRUE;
  }

  /* read a buffer from the file - only look at the first
   * DEMO_BLKSZ to determine format type and extract attributes */
  len = sysfpRead(dcx->fp_demoCtx, (dvoid *)buf, (size_t)DEMO_BLKSZ);
#undef DEMO_BLKSZ
  /* Initialize stream parser so we can parse the content for meta
   * data information.  This is not necessary when all these information
   * are known during the encoding process.
   */
  if (!(dcx->spcx_demoCtx = vespInit()))
  {
    yslPrint("vespInit Failed\n");
    return TRUE;
  }

  /* parse the buffer for content attributes */
  if ( vespGetAttributes(dcx->spcx_demoCtx, buf, len,
                         height, width, aspect, frpks, bitrate,
                         initData, initLen) )
  {
    failed = TRUE;
    yslPrint("Error: %s is not an OSF file\n", filename);
  }

  /* seek back to beginning */
  sysfpSeek(dcx->fp_demoCtx, sysb8zero);

  return failed;
}

/*
 * Name: demoMakeVesHdr
 *      This function illustrate how to create a ves_hdr with information
 * about the content.
 */
static void demoMakeVesHdr(ves_hdr *vhdr, ub2 h, ub2 w, sb4 ar, ub4 frpks,
                           ub1 *initData, size_t initLen)
{
  /* clear struct */
  memset((dvoid *)vhdr, 0x0, sizeof(ves_hdr));

  /* vend: string storing vendor specific info */
  vhdr->vend = "Feed Demo v1.1";

  /* this is an important field where it tells the video server how
   * to interpret the tags */
  vhdr->fmt = ves_formatRKF; /* OSF is one type of raw-key format */

  /* we are going to arbitrary set both vidCmp and audCmp since this
   * information is not directly reference by the video server */
#define DEMO_VIDCMP     "video codec"
#define DEMO_AUDCMP     "audio codec"
  vhdr->vid._maximum = vhdr->vid._length = (ub4)strlen(DEMO_VIDCMP);
  vhdr->vid._buffer =
    (ub1 *)ysmGlbAlloc((size_t)vhdr->vid._maximum, "vidCmp");
  memcpy((void *)vhdr->vid._buffer, DEMO_VIDCMP,
         (size_t)vhdr->vid._maximum);
```

```
  vhdr->aud._maximum = vhdr->aud._length = (ub4)strlen(DEMO_AUDCMP);
  vhdr->aud._buffer =
    (ub1 *)ysmGlbAlloc((size_t)vhdr->aud._maximum, "audCmp");
  memcpy((void *)vhdr->aud._buffer, DEMO_AUDCMP,
         (size_t)vhdr->aud._maximum);

  vhdr->heightInPixels = h;
  vhdr->widthInPixels = w;
  vhdr->pelAspectRatio = ar;
  vhdr->frameRate = frpks;

  /* This init data gets prepended to the stream any time stream is played
   * from a position other that mkd_beginning. The decoder is expected to
   * be able to recognize this data and use it to decode the stream. */
  vhdr->compData._d = ves_formatRKF;
  vhdr->compData._u.rkf.initData._maximum =
    vhdr->compData._u.rkf.initData._length = (ub4)initLen;
  vhdr->compData._u.rkf.initData._buffer =
    (ub1 *)ysmGlbAlloc(initLen, "initData");
  memcpy((void *)vhdr->compData._u.rkf.initData._buffer,
         (void *)initData, initLen);

  /* we are done with the initdata */
  ysmGlbFree((dvoid *)initData);

  return;
}

/*
 * Name: demoPrepare()
 * Func: prepare content. Fills context for each requested session
 */
static boolean demoPrepare(demoCmds *cmd)
{
  boolean failed=FALSE;
  sword   i;

  for (i=0; i < cmd->nsess_demoCmds; i++)
  { /* loop thru all input files */
    ub2         height, width;
    sb4         aspect;
    ub4         frpks;
    ub1        *initData;       /* initdata */
    size_t      initLen;        /* length of initdata */
    ves_hdr     vhdr;           /* ves header */
    ves_time    duration;       /* content duration to be kept */
    demoCtx    *dctx = &(cmd->ctx_demoCmds[cmd->asess_demoCmds]);

    /* get content specific information.  Note, this is called since
     * we do not know anything about the content.  When encoding, these
     * information should be known and therefore this call is not
     * necessary. */
    failed = demoGetCtntInfo(dctx, cmd->inFiles_demoCmds[i],
                             &height, &width, &aspect,
                             &frpks, &dctx->bitrate_demoCtx,
                             &initData, &initLen);

    if (!failed)
    { /* everything is normal about the input file */
      ub4      tmp;
      char    *lgName;  /* name of the feed */
      size_t   len;
      ub4      ipm;

      /* construct the ves header from above information */
      demoMakeVesHdr(&vhdr, height, width, aspect, frpks,
                     initData, initLen);
```

```
    /* construct a name for this feed.
     * sysfpExtractBase() extracts the basename from a full filename,
     * stripping both a path (if any) and suffix (if any).
     */
    sysfpExtractBase(dctx->baseName_demoCtx, cmd->inFiles_demoCmds[i]);
    /* concat volume name with basename to form a logical name */
    len = (cmd->volName_demoCmds) ? strlen(cmd->volName_demoCmds) :
      strlen(DEFAULT_VOLUME);
    len +=  (strlen(dctx->baseName_demoCtx) + 2);
    lgName = (char *)ysmGlbAlloc(len, "lgName");
    if (cmd->volName_demoCmds)
      sprintf(lgName, "%s/%s", cmd->volName_demoCmds,
              dctx->baseName_demoCtx);
    else
      sprintf(lgName, "%s/%s", DEFAULT_VOLUME,
              dctx->baseName_demoCtx);

    /* determine the duration we want to keep on the video server */
    if (cmd->length_demoCmds < DEMO_DEFDURATION)
      cmd->length_demoCmds = DEMO_DEFDURATION;  /* 3-minute is minimum */
    /* convert from seconds to ves_time */
    tmp = cmd->length_demoCmds;
    duration._d = ves_timeTypeSMPTE;
    duration._u.ves_timeSMPTE.hour = (tmp / 3600);
    tmp %= 3600;
    duration._u.ves_timeSMPTE.minute = (ub1)(tmp / 60);
    tmp %= 60;
    duration._u.ves_timeSMPTE.second = (ub1)(tmp);
    duration._u.ves_timeSMPTE.frame = 0;
    /* or alternatively we can also do
     * duration._d = ves_timeTypeMillisecs;
     * duration._u.ves_timeMs = cmd->length_demoCmds * 1000;
     */

    /* in osf files, we are only sending i-frame or keyframe tags
     * to the video server, therefore, p-frame & b-frame arguments are
     * set to zeros.  In particular, at this time we don't known the
     * keyframe frequency.  Therefore we over estimate the keyframe
     * rate by assuming every frame is a keyframe.  Note, the conversion
     * below is from frame per kiloseconds to frame per minute.
     */
    ipm = frpks * 60 / 1000;
    /* Create a feed session */
    yslPrint("Creating session %s for %d seconds\n",
             lgName, cmd->length_demoCmds);
    failed = veswNewFeed(&dctx->veswCtx_demoCtx, lgName, TRUE,
                         dctx->bitrate_demoCtx,
                         &duration, ipm, 0, 0, &vhdr);

    if (!failed)
    { /* good! feed session is created */
      cmd->asess_demoCmds++;
    }
    else
    {
      yslPrint("Fail to create new feed: %s\n",
               veswLastError(dctx->veswCtx_demoCtx));
      veswClose(dctx->veswCtx_demoCtx);
      dctx->veswCtx_demoCtx = (veswCtx *)0;
    }
    if (vhdr.aud._buffer)
      ysmGlbFree((dvoid *)vhdr.aud._buffer);
    if (vhdr.vid._buffer)
      ysmGlbFree((dvoid *)vhdr.vid._buffer);
    if (vhdr.compData._u.rkf.initData._buffer)
      ysmGlbFree((dvoid *)vhdr.compData._u.rkf.initData._buffer);
    ysmGlbFree((void *)lgName);
  }
```

```
    if (failed)
    { /* clean up */
      if (dctx->fp_demoCtx)
      {
        sysfpClose(dctx->fp_demoCtx);
        dctx->fp_demoCtx = (sysfp *)0;
      }
      if (dctx->spcx_demoCtx)
      {
        vespTerm(dctx->spcx_demoCtx);
        dctx->spcx_demoCtx = (vespCtx *)0;
      }
    }
  }

  if (cmd->asess_demoCmds == 0)
    failed = TRUE;
  return failed;
}

/*
 * release allocated resources
 */
static boolean demoUnprepare(demoCmds *cmd)
{
  boolean failed=FALSE;
  sword   i;

  /* loop thru all input files */
  for (i=0; i < cmd->asess_demoCmds; i++)
  {
    demoCtx *ctx = &(cmd->ctx_demoCmds[i]);

    if (ctx->veswCtx_demoCtx)
    { /* closing this feed session */
      yslPrint("Closing feed : %s\n", ctx->baseName_demoCtx);
      failed = veswClose(ctx->veswCtx_demoCtx);
      if (failed == TRUE)
        yslPrint("problem during close\n");
    }

    if (ctx->fp_demoCtx)
      sysfpClose(ctx->fp_demoCtx);      /* closing input file pointer */
    if (ctx->spcx_demoCtx)
      vespTerm(ctx->spcx_demoCtx);      /* release parser resources */
    if (ctx->buf_demoCtx)
      ysmGlbFree((dvoid *)ctx->buf_demoCtx);
  }
  return failed;
}


/*
 * Do actual work here - sending data and corresponging tags to the server
 */
static void demoDoWork(demoCmds *cmd)
{
  sysb8   tmp1, tmp2, elapseTime, endTime, baseTime; /* all in micro-seconds */
  ystm    sttm;                 /* ys time structure */
  char    str[YSTM_BUFLEN];     /* time string */
  sysb8   cst;                  /* conversion constant */
  ves_tag *tagBuf;              /* holding an array of tags */
  ub4      tagnum;              /* number of tags in tagBuf */
  sword    report=0, jay, ticker=0;
  boolean  failed=FALSE;

  /* setup reporting frequency, everything 10 seconds */
```

```
report = (DEMO_REPORTTIME / DEMO_PROCESSTIME);
/* a bit/s to byte/us conversion factor */
sysb8addb4(&cst, sysb8zero, 8000000);
sysb8set(&elapseTime, sysb8zero);
/* run for at least the duration being kept on the server */
if (cmd->seconds_demoCmds < cmd->length_demoCmds)
  cmd->seconds_demoCmds = cmd->length_demoCmds;
sysb8addb4(&endTime, sysb8zero, cmd->seconds_demoCmds);
/* Convert from seconds to micro seconds */
sysb8mulb4(&endTime, &endTime, 1000000);

/* define and display the baseTime */
ysClock(&baseTime);
ysConvClock(&baseTime, &sttm);
yslPrint("Simulation started at %s and will run for %d seconds\n",
         ysStrClock(str, &sttm, TRUE, 0), cmd->seconds_demoCmds);

while (sysb8cmp(&elapseTime, <, &endTime))
{
  size_t  len, alen;
  sysb8  targetTime;

  ticker++;   /* report ticker */
  for (jay=0; jay < cmd->asess_demoCmds; jay++)
  { /* loop thru active sessions */
    demoCtx *cx = &cmd->ctx_demoCmds[jay];

    /* determine the byte location of this batch */
    sysb8addb4(&targetTime, &elapseTime, DEMO_PROCESSTIME);
    sysb8mulb4(&tmp2, &targetTime, cx->bitrate_demoCtx);
    sysb8addb4(&tmp2, &tmp2, (8000000 - 1));  /* make it round up */
    sysb8div(&tmp1, &tmp2, &cst);
    /* determine the length of this batch */
    sysb8sub(&tmp2, &tmp1, &cx->bytep_demoCtx);
    len = (size_t)sysb8tosb8(&tmp2);
    if (len > cx->bflen_demoCtx)
    {
      if (cx->buf_demoCtx)
        ysmGlbFree((dvoid *)cx->buf_demoCtx);
      cx->buf_demoCtx = (ub1 *)ysmGlbAlloc(len, "buffer");
    }
    /* read in a buffer for parsing */
    alen = sysfpRead(cx->fp_demoCtx, (dvoid *)cx->buf_demoCtx, len);
    if (alen != len)
    { /* we are at the end of the file, seek back to the beginning */
      sysfpSeek(cx->fp_demoCtx, sysb8zero);
    }

    /* vespGetTags parses the buf_demoCtx buffer for tag/meta information.
     * The input buffer is unaltered. The resulting meta information is
     * then shipped to the server by the veswSendTags function.
     * The tags are stored in the ves_tag structures.  Since OSF files
     * are handled as Raw Key Frame files on the server.  The entries of
     * raw-kay tag structure may look like the following.
     *   // The following assumes tag is an instance of ves_tag (ves_tag tag)
     *   // frameType is the type of the frame this tag is describing
     *   // This one is Key-Frame and could also be ves_frameRawBais for
     *   // delta frame.
     *   tag.frameType = ves_frameRawKey;
     *   // timestamp is the presentation time of the tagged frame
     *   tag.timestamp._d = ves_timeTypeMillisecs;
     *   tag.timestamp._u.ves_timeMs = 10000;  // be displayed at 10 sec.
     *   // byteOffset is the location of the start of this frame in the
     *   // byte stream
     *   tag.byteOffset = 526474;
     *   // videoByteLength is the length of this tagged frame
     *   tag.videoByteLength = 2080;
     *   // compData contains compression specific information about this
```

```
     *   // tagged frame.  For RKF (raw-kay-frame), this field is empty.
     *   tag.compData._d = ves_formatRKF;
     *   tag.compData._u.rkf.nothing = NULL;
     *   // However, there are required fields for mpeg1 and mpeg2.
     *   // For example in MPEG1 system stream, we need the video stream id:
     *   // tag.compData._d = ves_formatMpeg1SS;
     *   // tag.compData._u.m1s.videoStreamCode = 224;
     *   // For example in mpeg2 transport stream
     *   // tag.compData._d = ves_formatMpeg2Trans;
     *   // tag.compData._u.m2t.continuityCounter = 0xC;
     *   // tag.compData._u.m2t.leadingZeros = 148;
     *   // tag.compData._u.m2t.trailingZeros = 99;
     *   // tag.compData._u.m2t.headerPesLength = 588;
     *   // tag.compData._u.m2t.trailerPesLength = 1356;
     *   // tag.compData._u.m2t.nonVideoPackets = 10;
     *   // tag.compData._u.m2t.nullPackets = 0;
     */
    if ( vespGetTags(cx->spcx_demoCtx, cx->buf_demoCtx, alen,
                     &tagBuf, &tagnum) )
    {
      yslPrint("Error - vespParse failed\n");
      failed = TRUE;
    }

    if ((alen > 0) && (!failed))
    {
      /* SendData is a non-blocking call that pushes data into network
         buffers. */
      if (veswSendData(cx->veswCtx_demoCtx, (dvoid *)cx->buf_demoCtx, alen))
      {
        yslError("Something bad happened in SendData - %s\n",
                 veswLastError(cx->veswCtx_demoCtx));
        failed = TRUE;
      }
    }

    if ((tagnum > 0) && (!failed))
    {
      /* send meta data */
      if (veswSendTags(cx->veswCtx_demoCtx, tagBuf, tagnum))
      {
        yslError("Something bad happened in SendTags - %s\n",
                 veswLastError(cx->veswCtx_demoCtx));
        failed = TRUE;
      }
      ysmGlbFree((dvoid *)tagBuf);
    }
    sysb8addb4(&(cx->bytep_demoCtx), &(cx->bytep_demoCtx), (sb4)alen);
    cx->totalTags_demoCtx += tagnum;

    if ( (ticker % report) == 0 )
    { /* report activity */
      ysClock(&tmp1);
      ysConvClock(&tmp1, &sttm);
      yslPrint("[%s] %s - % 4d tags and %d bytes sent\n",
               ysStrClock(str, &sttm, FALSE, 0), cx->baseName_demoCtx,
               cx->totalTags_demoCtx, (ub4)sysb8tosb8(&(cx->bytep_demoCtx)));
    }
  }

  if ((failed) || (cmd->intr_demoCmds))
    break;

  /* keep the server connection active */
  veswIdle(cmd->ysCtx_demoCmds);

  ysClock(&tmp1);
  sysb8sub(&elapseTime, &tmp1, &baseTime);
```

```
    if (sysb8cmp(&elapseTime, <, &targetTime))
    {
      /* Here we delay a while to maintain our average bitrate. In a
         real program, we'd be yielding to the other threads that are
         preparing the content and tag data. Calling veswIdle is
         important because it gives Media Net a chance to push data
         through the network. */
      do
      {
        veswIdle(cmd->ysCtx_demoCmds);  /* keep the server connection active */
        ysClock(&tmp1);
        sysb8sub(&elapseTime, &tmp1, &baseTime);
      } while (sysb8cmp(&elapseTime, <, &targetTime));
    }
    else
      yslPrint("Warning: falling behind, video playback may be glitchy\n");
  }

  return;
}


/*
 * Name: demoParseArgs
 * Func: parse the input arguments to setup the feed
 * Return:
 *      TRUE is parsing fails
 *      FALSE otherwise
 */
static boolean
demoParseArgs(demoCmds *cmd, sword argc, char **args)
{
  boolean       failed = TRUE;
  sword         yssts;
  char         *arg;
  yslst        *files;  /* link list containing input files */

  yssts = ysArgParse((sword)(argc-1), (args+1), vesdemoArgMap);
  switch (yssts)
  {
  case YSARG_NORMAL:    /* okay, everything is normal */
    if (arg = ysResGetLast( "vesdemo.duration" ))
      cmd->seconds_demoCmds = (ub4)atoi(arg);

    if (arg = ysResGetLast( "vesdemo.length"))
      cmd->length_demoCmds = (ub4)atoi(arg);

    if (arg = ysResGetLast( "vesdemo.MDS-volume" ))
      cmd->volName_demoCmds = arg;

    /* get the list of the input files */
    if (files = ysResGet( "vesdemo.video-files" ))
    {
      ysle *elm;
      for (elm=ysLstHead(files); elm; elm=elm->next)
      {
        if (cmd->nsess_demoCmds < DEMO_MAXFEEDSESS)
        {
          cmd->inFiles_demoCmds[cmd->nsess_demoCmds] = (char *)ysLstVal(elm);
          yslPrint("Queuing input file %s\n",
                   cmd->inFiles_demoCmds[cmd->nsess_demoCmds]);
          cmd->nsess_demoCmds++;
        }
        else
          yslPrint("Exceed maximum feed sessions: ignore %s\n",
                   (char *)ysLstVal(elm));
      }
    }
```

```
    break;
  case YSARG_VERSION:   /* -V */
    yslPrint("vesdemo V 1.1\n");
    break;
  default:
    break;
  }

  if (cmd->nsess_demoCmds > 0)
    failed = FALSE;

  return failed;
}

/*
 * Handle control-c for clean exits
 */
static void demoIntrHdlr(dvoid *usrp, CONST ysid *exid, dvoid *arg, size_t sz)
{
  demoCmds *cmd=(demoCmds *)usrp;

  cmd->intr_demoCmds = TRUE;
}

/*
 * Main program
 */
int main(int argc, char *argv[])
{
  boolean     failed = FALSE;
  demoCmds    cmd;
  veswLayer   result;

  /* clear context */
  memset((void *)&cmd, 0x0, sizeof(demoCmds));

  /* initialize ves and ys (which allows yslPrint to be called */
  result = veswInit(cmd.ysCtx_demoCmds, argv[0]);
  if (result != veswLayerSuccess)
  {
    yslPrint("veswInit failed\n");
    failed = TRUE;
  }
  else
  {
    /* set up interrupt handler so we can exit cleanly on control-c */
    ysSetIntr(demoIntrHdlr, (dvoid *)&cmd);

    /* Parse command argument */
    failed = demoParseArgs(&cmd, (sword)argc, argv);

    if (failed == FALSE)
    { /* got the command successfully */
      failed = demoPrepare(&cmd);     /* prepare feed */
      if (!failed)
        demoDoWork(&cmd);             /* do actual work */
      demoUnprepare(&cmd);    /* unprepare feed */
    }
  }

  /* terminate the feed session */
  result = veswTerm(cmd.ysCtx_demoCmds);
  if (result != veswLayerSuccess)
  {
    failed = TRUE;
    yslPrint("irregularity during termination\n");
  }
```

```
  return (failed == TRUE) ? 1 : 0;
}
```

cont.idl

```
//--------------------------------------------------------------------------
// File: cont.idl - Interface Definitions for Content Resolver demo
//
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
// Oracle Corporation
// Oracle Media Server (TM)
// All Rights Reserved
// Copyright (C) 1993-1996
//--------------------------------------------------------------------------

#ifndef CONTDEMO_ORACLE
#define CONTDEMO_ORACLE

#include <mzacom.idl>
#include <mkd.idl>
//--------------------------------- Includes ---------------------------------

//--------------------------------------------------------------------------
// Module: demo
// Function:
//   This is the module that can be used for demo applications.
//--------------------------------------------------------------------------
module demo
{

   //------------------------------------------------------------------------
   // Exceptions
   //------------------------------------------------------------------------

   //------------------------------------------------------------------------
   // Exception: NoContent
   // Function:
   //
   //------------------------------------------------------------------------
   exception NoContent
   {
     string     contName;
     string     errMsg;
   };
   //------------------------------------------------------------------------
   // Exception: DuplicateContent
   // Function:
   //
   //------------------------------------------------------------------------
   exception DuplicateContent
   {
     string     contName;
     string     errMsg;
   };
   //------------------------------------------------------------------------
   // Exception: FileError
   // Function:
   //
   //------------------------------------------------------------------------
   exception FileError
   {
     string     fileName;
     string     errMsg;
```

```
};


//-----------------------------------------------------------------------
// Typedefs
//-----------------------------------------------------------------------

//---------------------------------------------------------------------
// Typedef: segData
// Function: Information about Content Segments
//---------------------------------------------------------------------
struct segData {
 string fileName;// The Tag File Name
 unsigned long startPos; // The Start Position
     unsigned long stopPos;  // The Stop Position
     charrwFlag;// Rewind Allowed? (Y/N)
     char    ffFlag;// Fast Forward Allowed? (Y/N)
     char    pauseFlag;// Pause Allowed? (Y/N)
};

//---------------------------------------------------------------------
// Typedef: segDataLst
// Function: A list of segData structures
//---------------------------------------------------------------------
typedef sequence <segData> segDataLst;

struct contAtr {
 string            contName;// The name of the content
     mkd::assetCookie   cookie;// The asset cookie for this content
 segDataLst        data;// List of the Segments for this
        // content. Only filled in if longFmt
        // is set to TRUE.
};

//---------------------------------------------------------------------
// Typedef: contAtrLst
// Function: A list of contAtr structures
//---------------------------------------------------------------------
typedef sequence <contAtr> contAtrLst;

//-----------------------------------------------------------------------
// Interface: cont
// Function:
//
//-----------------------------------------------------------------------
interface cont
{
  //-----------------------------------------------------------------------
  // Name: demo::cont::create
  // Function:
  //
  // Input:
  //
  // Output:
  //
  // Returns:
  //
  // Raises:
  //   Nothing
  //-----------------------------------------------------------------------
  void create(in string name, in  segData data)
  raises (DuplicateContent, FileError);

  //-----------------------------------------------------------------------
  // Name: demo::cont::addSegData
  // Function:
  //
  // Input:
```

```
    //
    // Output:
    //
    // Returns:
    //
    // Raises:
    //   Nothing
    //------------------------------------------------------------------------
    void addSegData(in string name, in  segData data)
    raises (NoContent, FileError);

    //------------------------------------------------------------------------
    // Name: demo::cont::query
    // Function:
    //
    // Input:
    //   Nothing
    // Output:
    //   Nothing
    // Returns:
    //
    // Raises:
    //   Nothing
    //------------------------------------------------------------------------
    contAtrLst query(in boolean longFmt,
        inout mza::Itr iterator)
    raises (FileError);

    //------------------------------------------------------------------------
    // Name: demo::cont::queryByNm
    // Function:
    //
    // Input:
    //   Nothing
    // Output:
    //   Nothing
    // Returns:
    //
    // Raises:
    //   Nothing
    //------------------------------------------------------------------------
    contAtr queryByNm(in string name, in boolean longFmt)
    raises (NoContent, FileError);

  };

};

#endif // MFSI_ORACLE
```

contImpl.c

```
#ifndef SYSX_ORACLE
#include <sysx.h>/* Oracle defs */
#endif
#ifndef YSFMT_ORACLE
#include <ysfmt.h>
#endif
#ifndef YO_ORACLE
#include <yo.h>
#endif
#ifndef YOCOA_ORACLE
#include <yocoa.h>
#endif
#ifndef CONTI_ORACLE
#include <contI.h>
#endif
#ifndef ORASTDIO
#include <stdio.h>
#define ORASTDIO
#endif
#ifndef ORASTRING
#include <string.h>
#define ORASTRING
#endif

#define MAX_LINE_LENGTH 1024

/*
 * Suggested definition only. It need not be
 * const or static, or even defined at compile time.
 */
externdef CONST_W_PTR struct demo_cont__tyimpl demo_cont__impl =
 {
  demo_cont_create_i,
  demo_cont_addSegData_i,
  demo_cont_query_i,
  demo_cont_queryByNm_i
 };

static FILE * open_file(const char *mode);
static void check_content(demo_cont or, yoenv* ev, char *name);

externref char           *myImplId;
/*
 * METHOD:
 *  demo_cont_create_i
 *
 * DESCRIPTION:
 *  This method appends a new content to the end of the content file.
 *  It first checks to see if content with the given name already exists
 *  and if it does, if throws a DuplcateContent Exception
 *
 */
void demo_cont_create_i( demo_cont or, yoenv* ev, char* name, demo_segData*
  data)
{
   FILE *f1;

   /*
    * Verify the content does not already exists
    */
   check_content(or, ev, name);

   /*
    * Open the file to append the new content
```

```
    */
   f1 = open_file("a");

   /*
    * Add the new content to the end of the file
    */
   DISCARD fprintf(f1, "%s:%s %d %d %c %c %c\n",name,
    data->fileName, data->startPos, data->stopPos,
    data->rwFlag, data->ffFlag, data->pauseFlag);

   /*
    * Close the file
    */
   DISCARD fclose(f1);

   return;
}

/*
 * METHOD:
 *  demo_cont_addSegData_i
 *
 * DESCRIPTION:
 *  This method append some nre segment data to the named content. To
 *  keep it simple, it does this by copying the existing file one line
 *  at a time to a temporary file, appending the new segdata at the
 *  appropriate point. It then moves the temporary file over onto the
 *  content file, thus replacing it with the modified version. A check
 *  is made to verify the conent actually exists first and an exception
 *  is thrown if it does not.
 *
 */
void demo_cont_addSegData_i( demo_cont or, yoenv* ev, char* name,
  demo_segData* data)
{
   FILE *f1;
   FILE *f2;
   char line[MAX_LINE_LENGTH+1];
   char *tmpName;
   demo_contAtr atr;

   /*
    * Verify the content exists. Throws NoContent if it doesn't exist
    */
   atr =  demo_cont_queryByNm_i(or, ev, name, FALSE);

   f1 = open_file("r");
   /*
    * Create a temporary file name
    */
   tmpName = tmpnam((char*)NULL);
   if((f2 = fopen(tmpName,"w")) == (FILE *)NULL)
   {
    demo_FileError ex;
    ex.fileName = tmpName;
    ex.errMsg = (char*)(dvoid*)"Could not open temporary file";
    yseThrowObj(DEMO_EX_FILEERROR, ex);
   }

   while (fgets(line, MAX_LINE_LENGTH, f1) != (char *) NULL)
   {
    /*
     * Write the line to the temporary file
     */
        /*
         * If this is the content we want, write the new segment data
         * onto the end
     */
```

```
    if(strncmp(line,name,strlen(name)) == 0)
    {
       char new[MAX_LINE_LENGTH];
       /*
        * Remove the newline character from the end of the line
        * and then append the new segment data
        */
       if(line[strlen(line)-1] == '\n')
        line[strlen(line)-1] = '\0';
       ysFmtStr(new,"%s:%s %d %d %c %c %c\n",
            line,
            data->fileName, data->startPos, data->stopPos,
            data->rwFlag, data->ffFlag, data->pauseFlag);
       DISCARD fputs(new, f2);
    }
    else
       DISCARD fputs(line, f2);

   }

   DISCARD fclose(f1);
   DISCARD fclose(f2);

   /*
    * Now move the temporary file onto the original file using the
    * brute force method
    */
   f1 = open_file("w");
   if((f2 = fopen(tmpName,"r")) == NULL)
   {
    demo_FileError ex;
    ex.fileName = tmpName;
    ex.errMsg = "Could not open temporary file";
    yseThrowObj(DEMO_EX_FILEERROR, ex);
   }
   while (fgets(line, MAX_LINE_LENGTH, f2) != (char *) NULL)
   {
    /*
     * Write the line to the content file
     */
    DISCARD fputs(line, f1);
    fprintf(f2,"\n");

   }

   DISCARD fclose(f1);
   DISCARD fclose(f2);


   return;


}

demo_contAtrLst demo_cont_query_i( demo_cont or, yoenv* ev, boolean longFmt,
   mza_Itr* iterator)
{
   FILE *f1;
   int wanted, rowCounter, fetched;
   char *tok;
   demo_contAtrLst atrLst;
   char line[MAX_LINE_LENGTH];

   atrLst._length = 0;
   atrLst._maximum = iterator->NumItems;
   atrLst._buffer = (demo_contAtr *)yoAlloc(sizeof(demo_contAtr)*
           iterator->NumItems);
   wanted = iterator->NumItems;
```

```
   rowCounter = 0;
   fetched = 0;

   f1 = open_file("r");

   while (fgets(line, MAX_LINE_LENGTH, f1) != (char *) NULL &&
    fetched < wanted)
   {
    if(rowCounter++ < iterator->Position)
        continue;

    /*
     * Parse the line
     */
    tok = strtok(line,":");
    atrLst._buffer[fetched].contName = (char*)yoAlloc(strlen(tok)+1);
    DISCARD strcpy(atrLst._buffer[fetched].contName,tok);
        atrLst._buffer[fetched].cookie = (mkd_assetCookie )
        yoAlloc((mkd_assetCookieMaxlen + 1) * sizeof(char));
        ysFmtStr((char *)atrLst._buffer[fetched].cookie, "%s:%s",
         myImplId, tok);

        if(longFmt)
        {
           int             maxSegments = 0;
           char *arg;
       demo_segData *data;
           int seg;

           arg = ysResGetLast("contsrv.maxSegments");
           if (!arg)
              maxSegments = 5;
           else
              maxSegments = atoi(arg);

       atrLst._buffer[fetched].data._length = 0;
       atrLst._buffer[fetched].data._maximum = maxSegments;
       atrLst._buffer[fetched].data._buffer = (demo_segData*)yoAlloc(
            maxSegments*sizeof(demo_segData));

           seg = 0;
       while((tok = strtok(NULL,":")) != NULL &&
        seg < maxSegments) /* Get the segments */
       {
        data = &atrLst._buffer[fetched].data._buffer[seg];
             data->fileName = (char *)yoAlloc(sizeof(char)*256);
        sscanf(tok,"%s %d %d %c %c %c",
            data->fileName, &data->startPos, &data->stopPos,
            &data->rwFlag, &data->ffFlag, &data->pauseFlag);
        ++seg;
        ++atrLst._buffer[fetched].data._length;

       }
        }
    else
    {
       atrLst._buffer[fetched].data._length = 0;
       atrLst._buffer[fetched].data._maximum = 0;
       atrLst._buffer[fetched].data._buffer = (demo_segData*)0;
        }
    ++fetched;
   }

   atrLst._length = fetched;
   iterator->NumItems = atrLst._length;
   if(fetched < wanted)
      iterator->Position = -1;
   else
```

```
      iterator->Position = rowCounter;

   DISCARD fclose(f1);

   return atrLst;

}

demo_contAtr
demo_cont_queryByNm_i(demo_cont or, yoenv * ev, char *name,
              boolean longFmt)
{
   FILE           *f1;
   int             wanted,
                   rowCounter,
                   fetched;
   char           *tok;
   demo_contAtr    atr;
   char            line[MAX_LINE_LENGTH];
   boolean         found = FALSE;

   f1 = open_file("r");

   while (fgets(line, MAX_LINE_LENGTH, f1) != (char *) NULL)
   {
      /*
       * Check to see if this is the line we want
       */
      if (strncmp(line, name, strlen(name)) == 0)
      {
     found = TRUE;
     /*
      * Parse the line
      */
     tok = strtok(line, ":");
     atr.contName = (char *) yoAlloc(strlen(tok) + 1);
     DISCARD strcpy(atr.contName, tok);
     atr.cookie = (mkd_assetCookie)
        yoAlloc((mkd_assetCookieMaxlen + 1) * sizeof(char));
     ysFmtStr((char *) atr.cookie, "%s:%s",
          myImplId, tok);

     if (longFmt)
     {
        int             maxSegments = 0;
        char           *arg;
        demo_segData   *data;
        int             seg;

        arg = ysResGetLast("contsrv.maxSegments");
        if (!arg)
           maxSegments = 5;
        else
           maxSegments = atoi(arg);

        atr.data._length = 0;
        atr.data._maximum = maxSegments;
        atr.data._buffer = (demo_segData *) yoAlloc(
            maxSegments * sizeof(demo_segData));

        seg = 0;
        while ((tok = strtok(NULL, ":")) != NULL &&
           seg < maxSegments)/* Get the segments */
        {
           data = &atr.data._buffer[seg];
           data->fileName = (char *)yoAlloc(sizeof(char)*256);
           sscanf(tok, "%s %d %d %c %c %c",
              data->fileName, &data->startPos, &data->stopPos,
```

```
            &data->rwFlag, &data->ffFlag, &data->pauseFlag);
         ++seg;
         ++atr.data._length;

      }
    }
    else
    {
       atr.data._length = 0;
       atr.data._maximum = 0;
       atr.data._buffer = (demo_segData *) 0;
    }
     }
   }
   if(found == FALSE)
   {
    demo_NoContent ex;
    ex.contName = name;
    ex.errMsg = "No Content Found";
    yseThrowObj(DEMO_EX_NOCONTENT, ex);
   }


   DISCARD fclose(f1);

   return atr;
}

static FILE *
open_file(const char *mode)
{
   char * fileName;
   FILE *f1;

   /*
    * Retrieve the filename we use to store all the data in
    */
   fileName = (char *) yscGetKeyed(YSC_BYNAME,
            (dvoid *) "fileName", strlen("fileName"));
   if((f1 = fopen(fileName,mode)) == (FILE*)NULL)
   {
    demo_FileError ex;
    ex.fileName = fileName;
    ex.errMsg = (char*)(dvoid*)"Could not open file";
    yseThrowObj(DEMO_EX_FILEERROR, ex);
   }
   return f1;
}

static void check_content(demo_cont or, yoenv* ev, char *name)
{
   boolean       noContent = FALSE;
   demo_contAtr  atr;

   yseTry
   {
      atr =  demo_cont_queryByNm_i(or, ev, name, FALSE);
   }
   yseCatchObj(DEMO_EX_NOCONTENT, demo_NoContent, ex)
   {
      noContent = TRUE;
   }
   yseEnd;

   /*
    * Check to make sure a content with this name does not already exist
    * If there is already a content with this name, then it is an error
    * so throw an exception
```

```
    */
   if(noContent == FALSE)
   {
    demo_DuplicateContent ex;
    ex.contName = name;
    ex.errMsg = (char*)(dvoid*)"Cannot create Duplicate Content";
    yseThrowObj(DEMO_EX_DUPLICATECONTENT, ex);
   }

}
```

rslvImpl.c

```
/* Copyright (c) 1996 by Oracle Corporation.  All Rights Reserved.
 *
 * rslvImpl.c - OVS Content Resolver Demo Implementation
 *
 * This is an example of how to write a content resolver for OVS 3.0.
 *
 * This is the actual implementation of the content resolver. It takes the
 * asset cookie which is passed in from the stream service and returns an
 * mkd_segmentlist.
 */
#ifndef SYSX_ORACLE
#include <sysx.h>/* Oracle defs */
#endif
#ifndef SYSFP_ORACLE
#include <sysfp.h>/* filesystem defs */
#endif
#ifndef YS_ORACLE
#include <ys.h>/* System layer defs */
#endif
#ifndef YO_ORACLE
#include <yo.h>/* orb defs */
#endif
#ifndef YOCOA_ORACLE
#include <yocoa.h>/* orb defs */
#endif
#ifndef MTCR_IDL
#include <mtcr.h>
#endif
#ifndef CONTI_H
#include <contI.h>
#endif

#ifndef CLRSTRUCT
#define CLRSTRUCT(a)  DISCARD memset((dvoid *) &(a), 0, sizeof(a))
#endif

#define MAX_SEGMENTS 5

mkd_segmentList
mtcr_resolve_name_i(mtcr_resolve or, yoenv * ev,
         char *name, CORBA_Object authRef, yoany *ckt);


/*---------------------------------------------------------------------*/
externdef CONST_W_PTR struct mtcr_resolve__tyimpl mtcr_resolve__impl =
{
   mtcr_resolve_name_i
};
/*
 * This is the actual code which resolve the asset cookie for the stream
 * service and returns the mkd_segmentList
```

```
 */
mkd_segmentList
mtcr_resolve_name_i(mtcr_resolve or, yoenv * ev,
        char *name, CORBA_Object authRef, yoany *ckt)
{

   mkd_segmentList retVal;
   int             i;
   long            t,
                   h,
                   m,
                   s;
   long            startTime,
                   stopTime;
   char            ffFlag,
                   rwFlag,
                   pauseFlag;
   int             maxSegments = 0;
   char           *arg;
   demo_contAtr    atr;
   boolean noContent;

   CLRSTRUCT(retVal);

/*
    * If you were implementing authorization, you could make a call to
    * the authorization Object passed in here. It may look something like
    * this:
    *
    *  yoenv *env;
    *  yoEnvInit(&env);
    *
    *  result = authIntf_authorize(authRef, &env);
    *  if(!result)
    *      yseThrow(MTCR_EX_AUTHFAILED);
    *
    *
    * You could also do something with the circuit passed in. This is a
    * copy of the circuit that the stream was allocated with and contains
    * the downstream address the stream is going to be sent to. This is similar
    * to what vscontsrv does.
    *
    *    if(*ckt->_type == yotkNull)
    *    {
    *      yseThrow(MTCR_EX_AUTHFAILED);
    *    }
    *    else
    *      circ = (mzc_circuit *)ckt->_value;
    *
    *    down_Addr = circ->info.downstream.info.comm.protocol.info
    *
    *    result = verify_Address(down_Addr)
    *    if(!result)
    *        yseThrow(MTCR_EX_AUTHFAILED);
*/

   /*
    * Invoke the method to get the attributes by the name given
    */
   yseTry
   {
      demo_cont   demoOR;

      demoOR = (demo_cont) yoBind( demo_cont__id, (char *)0, (yoRefData *)0,
                  (char *)0 );

      atr = demo_cont_queryByNm_i(demoOR, ev, name, TRUE);
      yoRelease((dvoid*)demoOR);
```

```
}
yseCatchObj(DEMO_EX_NOCONTENT, demo_NoContent, ex)
{
   noContent = TRUE;
}
yseEnd;

if(noContent == TRUE)
{
   mtcr_badName    ex;
   ex.theName = yoAlloc(strlen(name) + 1);
   strcpy(ex.theName, name);
   yslError("Could not find content %s\n", name);
   yseThrowObj(MTCR_EX_BADNAME, ex);
}

retVal._length = 0;
retVal._maximum = (ub4) atr.data._length;

retVal._buffer = (mkd_segment *) yoAlloc((size_t) (sizeof(mkd_segment) *
             retVal._maximum));
DISCARD memset((dvoid *) retVal._buffer, 0, (size_t) (retVal._maximum *
                              sizeof(mkd_segment)));

/*
 * Process each attribute element in the structure
 */
for (i = 0; i < atr.data._length; ++i)
{

   retVal._buffer[i].mkd_segFile = yoAlloc(
        strlen(atr.data._buffer[i].fileName)+1);
   strcpy(retVal._buffer[i].mkd_segFile, atr.data._buffer[i].fileName);

   /*
    * We create start and stop positions which are based on time
    */
   yslError("Resolved Segment %d\n", i);
   retVal._buffer[i].mkd_segStart._d = (sb4) mkd_posTypeTime;
   t = atr.data._buffer[i].startPos;
   h = t / 3600;
   t = t - 3600 * h;
   m = t / 60;
   t = t - 60 * m;
   s = t;

   retVal._buffer[i].mkd_segStart._u
  .mkd_posTimePos.mkd_posTimeHour = (ub4) h;
   retVal._buffer[i].mkd_segStart._u.
  mkd_posTimePos.mkd_posTimeMinute = (ub1) m;
   retVal._buffer[i].mkd_segStart._u.
  mkd_posTimePos.mkd_posTimeSecond = (ub1) s;
   retVal._buffer[i].mkd_segStart._u.
  mkd_posTimePos.mkd_posTimeHundredth = 0;

   retVal._buffer[i].mkd_segEnd._d = mkd_posTypeTime;
   t = atr.data._buffer[i].stopPos;
   h = t / 3600;
   t = t - 3600 * h;
   m = t / 60;
   t = t - 60 * m;
   s = t;

   retVal._buffer[i].mkd_segEnd._u.
  mkd_posTimePos.mkd_posTimeHour = (ub4) h;
   retVal._buffer[i].mkd_segEnd._u.
  mkd_posTimePos.mkd_posTimeMinute = (ub1) m;
   retVal._buffer[i].mkd_segEnd._u.
```

```
      mkd_posTimePos.mkd_posTimeSecond = (ub1) s;
       retVal._buffer[i].mkd_segEnd._u.
      mkd_posTimePos.mkd_posTimeHundredth = 0;

       /*
        * Set the Flags
        */
       retVal._buffer[i].mkd_segFlags = 0;
       if (atr.data._buffer[i].ffFlag == 'N')
      retVal._buffer[i].mkd_segProhib |=
         (mkd_prohibBlindFF |
          mkd_prohibVisualFF |
          mkd_prohibFrameAdv);
       if (atr.data._buffer[i].rwFlag == 'N')
      retVal._buffer[i].mkd_segProhib |=
         (mkd_prohibBlindRW |
          mkd_prohibVisualRW |
          mkd_prohibFrameRew);
       if (atr.data._buffer[i].pauseFlag == 'N')
      retVal._buffer[i].mkd_segProhib |=
         mkd_prohibPause;

       ++retVal._length;
   }


   return retVal;

}
```

metafile1.dat

```
Content1:  /mds/video/ovs_mpg1_1536k.mpi 0 5 N N N:/mds/video/ovs_mpg1_1536k.mpi 5 10 N N N:/mds/
video/ovs_mpg1_1536k.mpi 10 20 N N N
Content2:  /mds/video/ovs_mpg1_2048k.mpi 0 10 N N N
Content3:  /mds/video/ovs_mpg1_1536k.mpi 0 30 N N
```

Glossary

ActiveMovie A proprietary multimedia playback architecture for Microsoft Windows 95 and Windows NT that delivers high quality audio and video.

ActiveX Control A custom control to add support for multimedia *streams* to your Internet multimedia applications. This control runs under either Microsoft Windows 95 or Windows NT 4.0 and enables your applications to start, stop, and seek locations within video and audio streams from *OVS*.

administration object Provides the functions you need to create the connection between an *event consumer* and an *event supplier* in the *event channel*.

argument map Associates command-line options to resource names and values.

asset cookie An identifier that the *client application* uses to request a video from the *stream service* that contains either:

- a *logical content* title to be resolved by the *content resolver*
- a *physical content tag file* to be resolved in the *MDS*

asymmetric network A network with two or more unidirectional paths of communication that may have different bandwidths. See also *circuit*.

asynchronous Describing a function that returns control to its calling environment immediately regardless of whether all its operations are complete. Even if its operations take some time, an asynchronous function always returns without waiting. Compare to *synchronous*.

ATM Asynchronous Transfer Mode. ATM is the standard for *broadband* networking, wherein information for multiple service types (such as voice, video, or data) is conveyed in small, fixed-size cells.

AVI Audio Video Interleaved, a file *container format* defined by Microsoft Corporation. An AVI file typically includes both an audio and video sequence. The audio and video data can be in several compression formats, such as Intel Indeo, Iterated Systems ClearVideo (fractal), or Motion JPEG. However, to play a compressed AVI file, a *client* must have the corresponding *codec*.

B-frame Bidirectionally predictive-coded frame. In *MPEG* video, a B-frame is encoded using data describing how its picture has changed from the picture in a previous *I-frame* or *P-frame* and how the picture will change to the next.

bidirectional A two-way connection between the client and server. See *channel* and *circuit*.

bind To locate through *CORBA* a server that implements a specified interface and to return an *object reference*.

bit rate The speed, measured in bits per second (bps), at which a file is transmitted or encoded.

BLOB Binary Large Object. A BLOB is an arbitrary collection of bits (such as a still or executable image) that the *OVS* delivers in binary format to a *client*. OVS uses *reliable delivery* to deliver a BLOB to a client. Compare to *stream*.

broadband A high-bandwidth network environment, capable of delivering large amounts of video and other data at high speeds. Broadband networks are typically able to carry large numbers of concurrent 1 Mbps (or larger) streams.

broadcast data service An *OVS* service (**vsbcastsrv**) that manages *scheduled video* information in the *database*. The broadcast data service reads information, such as *schedules*, from the database and writes it to memory where it can be read by the *scheduler service* and by exporter services, such as the *NVOD exporter service*. The broadcast data service also provides interfaces for scheduling events.

cartridge A manageable object that plugs into and extends the functionality of a piece of software. Cartridges for the *OVS* use an IDL (Interface Definition Language), a language-neutral interface that allows them to identify themselves to other objects in a distributed system. Oracle Corporation writes cartridges such as logical content and distributes them with the OVS. Cartridges such as real-time encoders can also be written by third parties.

channel A generic description of a communications link. Channels are used to represent the common properties of upstream and downstream *circuits*. The properties of channels include:

- maximum supported bit rate
- maximum available bit rate
- whether the channel is enabled or disabled
- whether the channel supports transient operations such as disconnect / reconnect
- optional information about its underlying implementation such as network protocol and network software API

A channel can be one of these types:

- **upstream** — carries data from the *client* to the *OVS*
- **downstream** — carries data from the OVS to the client
- **bidirectional** — carries data in both directions between the client and the OVS

For *scheduled video* you can define a channel with a name, number, and physical network address, and then use *VSM* to *schedule logical content* to play on the channel.

circuit A path of communication between a *client* and the *OVS*. A circuit can be composed either of a single *channel* (a *symmetric* circuit) or of two channels (an *asymmetric* circuit). A circuit may be either:

- **unidirectional** — upstream or downstream, or
- **bidirectional** — upstream and downstream

A circuit can be classified by the type of data it carries:

- **control circuit** — carries all Media Net data and must be associated with a valid, unique Media Net transport address. A control circuit is always bidirectional, containing either a bidirectional channel or both an upstream channel and a downstream channel.
- **isochronous circuit** — delivers *isochronous* real-time data such as video and audio. An isochronous circuit is typically associated with a single *video pump* and can be either unidirectional or bidirectional, though it typically consists of a unidirectional downstream channel.
- **data circuit** — carries data other than Media Net data or video / audio data. A data circuit may be either unidirectional or bidirectional and does not necessarily have an associated transport address.

circuit geometry The physical arrangement of *channels* that make up a *circuit*. A circuit's geometry can be either:

- **bidirectional symmetric** — a circuit made up of a single bidirectional channel
- **bidirectional asymmetric** — a circuit made up of an upstream channel and downstream channel
- **unidirectional symmetric** — a circuit made up of a single channel in the same direction as the circuit
- **unidirectional asymmetric** — a circuit made up of a channel in one direction and a null channel in the other (for example, a downstream channel and a null upstream channel)

A unidiectional asymmetric circuit is functionally the same as a unidirectional symmetric circuit.

client or **client device**, a hardware device used to access the *OVS*, such as an *mux format*, a *PC*, or a *set-top box*. Compare to *server*.

client application An application that runs on a *client* device and communicates over the network with specialized *server applications*. The *VSM* console is an example of a client application.

client/server Distribution of responsibilities among computers on a network. When an application must perform an operation that requires resources or functionality not available locally, it requests that a remote application perform that operation. The application making the request is called the *client*, and the one performing the operation is called the *server*.

clip A logical part of a *physical content* file. Each clip maps to a specific start and stop position (in seconds) within a single content file. You can combine clips to create *logical content* titles using *VSM*.

codec Software or hardware for *encoding* and decoding, or *compressing* and decompressing, video or audio data. Each codec produces encoded video in a *compression format*. *OVS* supports several compression formats. To play a video stream, a *client* must have the appropriate codec to decompress it.

compressing Removing redundant data from a file to reduce its size. Compressed video can be more easily transported over networks and requires less disk space for storage. Compressing is part of *encoding* and is performed by *codecs*.

compression format The format of an encoded video or audio file produced by a *codec*. *OVS* supports several compression formats, including *MPEG*-1, *MPEG*-2, Intel Indeo, Iterated Systems ClearVideo (fractal), Radius Cinepak, and Motion *JPEG*.

container format The way in which an encoder multiplexes video and audio together. Container formats supported by *OVS* include *MPEG-1* system, *MPEG-2* transport, and any container format, such as *OSF*, that meets the *RKF* requirements.

content The multimedia data that the *OVS* stores and delivers to *clients*. Content can be video, audio, or both. Content can refer to either *physical content* or *logical content*.

content resolver A service or portion of the content service that receives *asset cookies* from the *stream service* and resolves them to one or more *clips*.

content service An *OVS* service (**vscontsrv**) that uses a *database* to map *logical content* titles to *physical content* files. The content service enables *client applications* to query the content available in the OVS. The content service also contains a *content resolver* to resolve *asset cookies* for the *stream service*.

continuous real-time feed An ongoing uninterrupted *stream* of video from a real-time encoder. The *OVS* can receive a continuous real-time feed and deliver it to *clients* while keeping the most recent portion stored in the *MDS*. A *client application* can display the ongoing video, such as a 24-hour-a-day news broadcast, and the user can navigate through the most recent portion, such as its last 12 hours.

control circuit See *circuit*.

CORBA Common Object Request Broker Architecture. CORBA is an open standard defined by the OMG (Object Management Group). By using CORBA heterogeneous clients and servers distributed over a network can communicate without concern for each other's location or the details in transporting or converting data among them.

CORBA event service An Oracle Media Net service (**yeced**) that provides client notification when an event occurs in an object. An event is data that indicates a change in an object. For example, when a client requests a content file, the *stream service* prepares to play the file and notifies the CORBA event service.

data circuit See *circuit*.

database A reliable repository for persistent storage of structured data. The *OVS* system can use the Oracle database to store and retrieve data associated with OVS services.

delta frame A video frame that describes a picture in terms of how it is different from the picture in a previous frame in the stream. Compare with *key frame*. In *MPEG* video, *B-frames* and *P-frames* are delta frames; *I-frames* are not.

domain name The network address of the *video server computer*. The domain name and the *host name* produces a fully-qualified host name. You must enter both to begin a *VSM session*.

downstream channel See *channel*.

encoding Capturing a video or audio signal, converting it to digital format, *compressing* it, and multiplexing the video and audio together. Encoding is performed by a *codec*.

event channel A Media Net object that allows event suppliers to communicate with event consumers asynchronously.

event consumer A Media Net object that processes the event data. Compare to *event supplier*.

event service A Media Net service that facilitates the transfer of events between objects.

event supplier A Media Net object that produces the event data. Compare to *event channel*.

factory A type of *CORBA* object that is used to create new objects.

feed server See *real-time feed service*.

FTP File transfer protocol. **1)** FTP is a generic term for an end-to-end network service that enables you to transfer a file between remote and local storage. **2)** FTP is also the name of the most popular file transfer protocol on the Internet.

glitch A flaw in delivery of a video *stream*. Glitches may manifest themselves as misaligned portions of the video screen, skipped or dropped portions of the video, video “noise”, or unsynchronized audio and video.

GOP Group of pictures, or a part of an *MPEG* video file made up of a sequence of frames that includes a single *I-frame*.

host name The name of the *video server computer*. The host name and the *domain name* produce a fully qualified host name. You must enter both to begin a *VSM session*.

IEC International Electrotechnical Commission. IEC develops industrial standards for information technology.

I-frame Intra-coded frame. An I-frame is a *key frame* in *MPEG* video.

I-frame regularity *OVS* expects *I-frames* to be provided at regular intervals throughout the video stream. Since *rate control* operations are accurate to the nearest I-frame, Oracle Corporation recommends encoding 1 or 2 I-frames per second. Fewer I-frames yield poor rate control performance, while more yield a larger video file. *MPEG encoding* software often enables you to configure I-frame frequency.

Internet A world wide multi-user computing environment composed of corporate, government, and educational networks delivering data such as video, audio, and Web pages using *IP* (Internet Protocol) over a variety of physical connections including fibre-optic links and PSTN (public switched telephone network).

intranet A corporate computing environment delivering data such as video, audio, and Web pages over a LAN (local-area network) or WAN (wide-area network) within a company.

IP Internet Protocol. IP is a low-level unreliable connectionless network protocol used in the *Internet* environment for passing data packets between host computers. IP provides addressing and some primitive error handling and the foundation for *TCP* and *UDP*.

ISO International Standards Organization. ISO develops industrial standards in fields including computing and data communications. ISO define *MPEG* standards in a joint technical committee with *IEC*.

isochronous Occurring at equal intervals of time. Applications receiving video streams are time-sensitive and depend on isochronous, or time-based, delivery. Video data packets must arrive on time to be useful, or the video suffers a brief interruption and the user may notice a flaw, or *glitch*, in the video. See *circuit*.

Java A general purpose, object-oriented programming language that is designed for heterogeneous networks, multiple host architectures, and secure delivery. As such, compiled Java code has to survive transport across networks, operate on any *client*, and assure the client that it is safe to run.

JPEG Joint Photographic Experts Group. JPEG is a static-image *compression format*. Many static images on the World Wide Web are JPEG.

key frame A video frame compressed using only its own redundancies and not depending on data in other frames. Rate control operations are accurate to the nearest key frame. Compare with *delta frame*. In *MPEG* video, key frames are called *I-frames*.

logical content A collection of titles that a *client application* can request from the *OVS*. Each logical content title represents a defined sequence of one or more *clips*, which are based on *physical content* files. By defining logical content titles, you can customize the content your client applications can request. In the simplest case, a logical content title represents one complete physical content file. In a more complex case, a logical content title can represent a sequence of parts of physical content files. Compare to *physical content*.

looping content Continuously repeating playback of a logical content title. Looping content is useful for displaying a moving logo or for using advertisements or public service announcements to fill time between the end of one *NVOD* program and the beginning of the next.

MDS Oracle Media Data Store. The MDS is a real-time file system for storing and delivering uninterrupted video in real time. The MDS is composed of:

- the physical disk subsystem that provides secondary storage
- the *MDS directory service* that tracks and manages the layout of files in the MDS volumes, controls access to the files, and monitors and limits the bandwidth used to access them
- the MDS utilities that load files into the MDS and perform standard file operations

MDS directory service An *OVS* service (**mdsdirsrv**) that controls access to *MDS* files and manages their layout on disk. Services that read or write MDS files must first gain access through the MDS directory service so that bandwidth can be reserved for their operations.

MDS FTP service MDS File Transfer Protocol service (**mdsftpsrv**). The MDS FTP service is an *OVS* service that performs binary file transfers between the *MDS* and *FTP* clients on remote computers.

MDS remote file service An *OVS* service (**mdsrmtsrv**) that provides access to *MDS* files for remote services, such as applications and MDS utilities on remote computers. The MDS remote file service also delivers *BLOB*s to *client applications* on request.

metadata Information, such as *bit rate* and compression format, that the *OVS* must have to stream a content file. Registering a content file writes its metadata into its associated *tag file* and to the *database*, if your OVS system includes one.

MPEG Moving Picture Experts Group, an *ISO* body focused on developing *compression formats* for full-motion video and audio signals and the synchronization of these signals during playback. MPEG formats are popular because they provide:

- a high storage compression ratio compared to storing each frame of the video in its original, decompressed form
- full-motion (up to 30 frames per second), full-screen video with high-fidelity / stereo audio playback

MPEG files are composed of three layers: video, audio, and system. The video and audio layers (also called elementary streams) contain the coded video and audio data, respectively. The system layer defines the multiplexed structure for the video and audio data, as well as the timing information required to replay synchronized segments in real time.

Two MPEG standards, MPEG-1 and MPEG-2, are widely-used. For a complete technical explanation of these standards, please refer to the appropriate specification (*IEC*/*ISO* 11172-1, 2, 3 for MPEG-1 and IEC/ISO 13818-1, 2, 3 for MPEG-2).

mux format See *container format*.

NC Network Computer, a network-based *client* device. NCs feature Oracle Corporation's thin-client architecture, which provides users with access to a distributed *NCA* environment. NCs have no local disk storage, so they download their software from a network. Compare with *PC* and *set-top box*.

NCA Network Computing Architecture. The NCA is a comprehensive, open, network-based architecture providing extensibility for distributed environments.

NVOD Near video-on-demand, a type of *scheduled video*. You can program the *OVS* to play a *logical content* title beginning at regular intervals (for example, every 15 minutes) on different channels. A *client application* can tune in to a channel when the video is beginning there, so the video is available on a "near-demand" basis. Compare to *VOD*.

NVOD exporter service An OVS service (**vsnvodsrv**) that plays *scheduled video*. When notified by the *scheduler service*, the NVOD exporter service reads a *schedule* and plays the scheduled *logical content* title on a scheduled *channel*. You can use this service to handle any scheduled video, such as NVOD, pay-per-view, or regular TV broadcasting.

object reference The unique name needed to identify an object in a *CORBA* call. The *ORB daemon* uses the object reference to direct the call to the appropriate server.

one-step encoding Encoding video, generating *metadata*, and storing them fast enough that the resulting content can be played as it is encoded. The *OVS* can accept a video stream and its associated metadata from a real-time encoder and store it in the *MDS* in one step.

Oracle Media Net Oracle Corporation's implementation of *CORBA*. Oracle Media Net is the ORB-based communication infrastructure used by all components of the *OVS* system. Using heterogeneous network protocols, Oracle Media Net allows connectionless communication among components on different platforms.

Oracle Video Client See *OVC*.

Oracle Video Server See *OVS*.

Oracle Video Web Plug-in A component of the *OVC* that enables World Wide Web pages to display video from the *OVS*.

Oracle VSM See *VSM*.

ORACLE_HOME An environment variable on most platforms that stands for the root of the Oracle product directory structure where Oracle software is installed.

ORB daemon Object Request Broker daemon (**mnorbsrv**). The ORB daemon:

- is an Oracle Media Net service
- enables applications, or objects, to make transparent requests to and receive responses from other objects that reside either on the same server computer or across a network
- is a consistent interface through which distributed applications communicate with one another

In addition to managing the communication between distributed client and server objects, the ORB daemon routes client requests to best balance the load across servers and hosts.

OSF Oracle Streaming Format. OSF is an *RKF container format* designed to enable efficient delivery of multimedia content that has not been previously optimized for streaming. When you load an *AVI* or *WAV* file into the *MDS*, the *OVS* converts it to OSF so that the *OVC* can play it.

OVC Oracle Video Client, the software component of the *OVS* system that runs on *client* devices and enables *client applications* to request, control, receive, and display multimedia data from the OVS. Specialized third-party software or hardware *codecs* decompress video data so it can be displayed by a client application.

OVS Oracle Video Server, an Oracle product that provides an end-to-end software solution for networked client and server computers that store, manage, deliver, and display digital video and audio on demand. Client applications run on a wide variety of consumer and corporate platforms. OVS is supported on a variety of server platforms and scales to serve many concurrent users.

packet The basic unit of information that travels between computers on a network. All data transmitted over a network, whether a small message or a large video file, is divided into packets. Each packet contains a header and data.

parity check The process of verifying the integrity of a character. A parity check involves appending a parity bit that makes the total number of binary "1" digits in a character or word (including the parity bit) either odd (for odd parity) or even (for even parity).

password A string of alphanumeric characters associated with a *username*. To connect to *OVS* with *VSM*, you must enter a username and password. VSM then verifies them on the *video server computer*.

PC Personal computer. A PC can be used as a *client* device in the *OVS* system. PCs have their own disks, so they store their *OVC* software locally, rather than downloading it from a network. Compare with *mux format* and *set-top box*.

P-frame Predictive-coded frame. In *MPEG* video, a P-frame is encoded using data describing how its picture has changed from the picture in a previous *I-frame* or *P-frame*.

physical content The actual physical files containing video and / or audio data that *OVS* stores and delivers to *clients*. Physical content files are created through *encoding*. Compare to *logical content*.

VSM enables you to manipulate physical content and shows you the *metadata* for each physical content file.

plug-in A browser-compatible application that opens within, or plugs into, a running browser.

proxy An entity, in the interests of efficiency, that essentially stands in for another entity.

proxy consumer A proxy object that represents an event consumer in the event channel. Compare to *proxy supplier*.

proxy object A Media Net object used by an *event consumer* and an *event supplier* to communicate with one another through the *event channel*. See also *event service*.

proxy supplier A proxy object that represents an event supplier in the event channel. Compare to *proxy consumer*.

push model The model in which the *event supplier* initiates the sending of event data to the *event consumer*. See also *event service*.

RAID protection Redundant Arrays of Inexpensive Disks protection. RAID protection is a software method for storing data redundantly across multiple disks. The *MDS* uses RAID protection to ensure data is available and recoverable in the event of a disk failure.

RAID set A group of disks in an *MDS volume*. If *RAID protection* is enabled for a volume, the disks in the volume are divided into RAID sets. The number of disks in a RAID set is the *RAID size*. When an MDS client writes to the disks in a RAID set, it writes original data to all but one disk and then writes redundant data to the remaining disk. If a disk fails, its data can be recovered from the redundant data stored in its RAID set. A volume can tolerate a single failed disk in each RAID set and still deliver video.

RAID size The number of disks in a *RAID set*. The RAID size is the same for all RAID sets in an *MDS volume* and is established in the *voltab file*.

rate control The ability to play video at different speeds and in different directions, to *seek* in either direction, to scale in either direction, or to pause and resume play of video.

read consistency In the *MDS*, read consistency guarantees the integrity of MDS data being read by a client. While a client is reading a file, no other client can write to the file, so the reading client always reads consistent data.

real-time encoding Encoding video and / or audio fast enough that the resulting content can be played as it is encoded. The *OVS* supports these forms of real-time encoding:

- *one-step encoding*
- *continuous real-time feed*

real-time feed A stream of video or audio of a live event.

real-time feed service An *OVS* service (**vsfeedsrv**) that receives encoded *physical content* and *metadata* from a real-time encoder and stores them in the MDS. The real-time feed service also calls the *content service* to create logical content.

For *one-step encoding*, the real-time feed service writes the encoded content to a single physical content file and the metadata to an associated *tag file* and to the *database*, if your OVS system includes one. The *stream service* and *video pump* can then play the content.

For a *continuous real-time feed*, the real-time feed service maintains a circular buffer of content files and writes the encoded content to one content file after another. The real-time feed service overwrites the oldest content files with new content as they become obsolete. The real-time feed service also maintains a single tag file for the circular buffer of content files.

registering Enabling the *OVS* to play a *physical content* file stored in the *MDS*. Registering a file includes:

- creating a *tag file* for it in the MDS
- creating a *clip* and a *logical content* title for it in the *database*, if your OVS system includes one

The OVS provides utilities (such as **vstag**) for registering content.

reliable delivery Delivery from server to client that results in either complete delivery or an error message, but not corrupted data. Reliable delivery uses *parity checks*, receipt acknowledgments from the client, sequence numbers, and retransmissions from the server. Compare to *unreliable delivery*.

RKF Raw Key Frame. The *OVS* can play content in any *container format* that conforms to the rules of RKF:

- **stateless** — all the data for displaying the picture in a video frame is contained entirely in that frame, rather than in any previous frames
- **contiguous** — all the data for a frame is stored together in the video file and no other data mixed with it

scalability The ability of the *OVS* to deliver increasing numbers of concurrent *streams* to concurrent users without *glitches* or failures. Scaling may require you to run multiple instances of some OVS services like the *stream service* and the *video pump*. Note that the scalability of the entire OVS system may be limited by the scalability of the network.

scan To fast forward or rewind a video stream while the user watches.

schedule Information in the database used for *scheduled video* that associates a *logical content* title with a delivery *channel* and a delivery time. For example, you might create a schedule to play "The Binky Bopper Cartoon Show" on channel 3 at 8 AM.

scheduled video Defining a *schedule* for the *OVS* to play a specific *logical content* title on a specific *channel* at a specific time. Clients can then "tune in" to see the video they want with no signals required from the client to the OVS. Scheduled video enables you to implement:

- regularly-scheduled television broadcasting
- pay-per-view
- *NVOD*

Compare to *VOD*.

scheduler service An *OVS* service (**vsschdsrv**) that keeps track of the current time and a list of *scheduled video* events and notifies the appropriate exporter service, such as the *NVOD exporter service*, when a broadcast event is scheduled to occur.

segment The data in a *physical content* file that makes up a *clip*.

server A process or computer that provides one or more services to *clients*.

server application An application that performs specific tasks on behalf of the *client application*, such as providing information on available videos.

service See *server application*.

session A collection of resources maintained by *OVS* on a *client*'s behalf. When connecting to the OVS, a client establishes a session, which consists of:

- an identifier for the client
- one or more control *circuits* that transport messages between the client and OVS
- zero or more isochronous circuits that transport video from OVS to the client
- zero or more data circuits
- resources, or state information, for the client and its connection to the OVS

session and circuit configuration file

A configuration file which assigns groups of video pumps to groups of *clients*. When a client connects to the *OVS*, the *session and circuit service* uses this file to assign it a *video pump*. Configuring sessions and circuits with this file enables you to flexibly deal with segmented network topologies, perform well-defined load distribution, and deliver multiple network protocols from the same OVS.

session and circuit service

An *OVS* service (**vscsmsrv**) that allocates *sessions* and *circuits* for *clients* connecting to the OVS. When a client ends its session, the session and circuit manager deallocates all the session's circuits and resources.

set-top box A *client* device typically used in the *broadband* environment. Set-top boxes are diskless and download their software from a network. A set-top box often incorporates hardware to decode and display video on a television set. Compare with *mux format* and *PC*.

seek To skip forward or backward in a video stream. The user cannot watch the video while skipping.

stand-alone mode A mode in which the *content service* runs without a *database* to map *logical content* titles to *physical content* files. In stand-alone mode if the *client application* requests a list of content, the content service returns a list of the *tag files* in *MDS*. The client application can only request content with a *tag file asset cookie*.

stream To deliver data from a server to a client in real time, on demand as the client needs it, instead of delivering all of it at once to be played when delivery is done. The *OVS* streams video to its *clients* so that a client does not have to store an entire video file at once. Video files are often too large to store on *client* devices. The *client application* displays the video as it is received, so the data must arrive in a timely manner.

The term *stream* also refers to data being streamed. Because *reliable delivery* is difficult to perform in real time, the *OVS* uses *unreliable delivery* to deliver video streams to clients, although you may choose a reliable network protocol, such as *TCP*. Compare to *BLOB*.

stream service An *OVS* service (**vsstrmsrv**) that handles client requests for content. The *client application* uses an *asset cookie* to identify either a *tag file* or a *logical content* title, in which case the stream service uses the *content resolver* to resolve the logical content title to *clips*. The stream service then tells the *video pump* which parts of which *physical content* files to play to meet the request.

stripe A portion of a disk reserved for a single piece of a file stored using *striping*.

striping Dividing a file into pieces and storing the pieces across several different disks, rather than concentrating it on one. Striping improves performance when the file is accessed by many users concurrently.

symmetric See *circuit*.

synchronous Describing a function that completes its operations before returning control to its calling environment. If the operations take some time, a synchronous function waits until they are complete. Compare to *asynchronous*.

tag file An *MDS* file that stores *metadata* about a *physical content* file, such as filename, file type, bit rate, file size, and information the *OVS* needs to randomly access the file and perform *rate control*. For each physical content file you load into the MDS, you must create a tag file. Tag files must have an **.mpi** extension. Creating a tag file is part of *registering* a physical content file so it can be played.

TNS alias Transparent Network Substrate alias. A TNS alias is a string that describes to a client how to connect to a remote Oracle *database*.

TCP Transmission Control Protocol. TCP is a connection-oriented transport layer protocol belonging to the *IP* family. TCP uses *reliable delivery* and delivers properly ordered data. Compare to *UDP*.

UDP User Datagram Protocol. UDP is a connectionless transport layer protocol belonging to the *IP* family. UDP uses *unreliable delivery*, sequencing, fragment reassembly, and data checksums. Compare to *TCP*.

unidirectional See *circuit*.

unreliable delivery Delivery from server to client that does not return an error message in the event of dropped, reordered, or corrupted data. Compare to *reliable delivery*.

upstream channel See *channel*.

username A name by which a user is known. To connect to the *OVS* with *VSM*, you must have a username and associated *password* on the *video server computer*.

video pump An *OVS* service (**vspump**) that reads video files from the *MDS* and delivers them to the network in real time. When a *client application* requests video, the video pump receives a message from the *stream service*, reads the appropriate portion of the appropriate *physical content* file, and sends video data over the network through the appropriate downstream channel to the client.

video server computer The computer on which the Oracle Video Server runs.

VOD Video-on-demand. *OVS* meets the client's demand to play a specific video at a specific time. Compare to *scheduled video.*

voltab file A file on the host file system of the *video server computer* with which you define the *MDS*. The **voltab** file specifies each MDS *volume* along with information about it, such as disks, RAID, striping, spare disks, and maximum bandwidth.

volume A named collection of disks that store *MDS* files.

vsbcastsrv See *broadcast data service*.

vscontsrv See *content service*.

vscsmsrv See *session and circuit service*.

vsfeedsrv See *real-time feed service*.

VSM Oracle Video Server Manager. The VSM console is an object-oriented, Java application with a graphical user interface for monitoring and managing the *OVS*. VSM simplifies the management of distributed multimedia servers and clients in heterogeneous environments.

vsnvodsrv See *NVOD exporter service*.

vspump See *video pump*.

vsschdsrv See *scheduler service*.

vsstrmsrv See *stream service*.

WAV Waveform-audio. WAV is a standard file *container format* developed by Microsoft Corporation for storing digital audio files.

write consistency In the *MDS*, write consistency means:

- An MDS file can be written by only one MDS client at a time.
- While a file is being written by one client, no other client can rename, remove, or truncate the file or lock the file into read-only mode.
- If a client fails while writing to a file, the *MDS directory service* detects the failure and makes the file available to be written by other clients.

yeced See *CORBA event service*.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

X

Y

Reader's Comment Form

Name of Document: Oracle Video Server Developer's Guide, Version 3.0
Part Number A53960-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

_____

_____

_____

_____

_____

_____

Please send your comments to:

Oracle Video Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Mailstop 6OP5
Redwood Shores, CA 94065

You can e-mail comments to: omsdoc@us.oracle.com

You can also fax us at (650) 506-7615.

If you would like a reply, please give your name, address, and telephone number below:

_____

_____

_____

_____

Thank you for helping us improve our documentation.