

Informix Video Foundation DataBlade Module

User's Guide

Version 1.2
September 1998
Part No. 000-5184

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Copyright © 1981-1998 by Informix Software, Inc., or its subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are trademarks of Informix Corporation or its affiliates: INFORMIX®, the Informix logo®, Dynamic Scalable Architecture™, Informix® Dynamic Server™, Universal Data Option™, Extended Parallel Option™, Advanced Decision Support Option™, MetaCube™ ROLAP Option, Web Integration Option™, Workgroup Edition™, OnLine Dynamic Server™, Illustra™, DataBlade®, Data Director™, and New Era™.

All other marks or symbols are registered trademarks or trademarks of their respective owners.

ACKNOWLEDGMENTS

Documentation Team: Marian Gallagher, Kimberly Bostrom, Michelle Caisse

Contributors: Farshid Arman, Satish Katiyar, Albert Landeck, Jimmy Zhang

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

In This Introduction	3
About This Manual	3
Organization of This Manual	4
Types of Users	5
Software Dependencies	5
Documentation Conventions	6
Typographical Conventions	6
Icon Conventions	8
Naming Conventions	8
Function Syntax Conventions	9
Additional Documentation	9
Printed Documentation	10
Release Notes, Documentation Notes, and Machine Notes	10
Related Reading	11
Informix Welcomes Your Comments	12

Chapter 1

Overview of the Video Foundation DataBlade Module

In This Chapter	1-3
What Is the Informix Video Foundation DataBlade Module?	1-3
Definition of Terms	1-3
Features of This DataBlade Module	1-6
Who Are the Users of the Informix Video Foundation DataBlade Module? 1-17	
Video Application Developers	1-17
Virtual Storage DataBlade Module Developers	1-17
Content-Processing DataBlade Module Developers	1-18
What Are Some Sample Uses of This DataBlade Module?	1-20
Video Courseware	1-20
TV Station Web Site	1-21

Chapter 2	Temporal Component	
	In This Chapter	2-3
	MedPtr Data Type	2-3
	External Representation	2-4
	Functions	2-5
	MedTC Data Type	2-7
	External Representation	2-7
	Functions	2-9
	MedChunk Data Type	2-12
	External Representation	2-12
	Functions	2-13
	Conversion Functions	2-18
Chapter 3	Storage Component	
	In This Chapter	3-3
	The Device-Independent VSI	3-4
	MedStorage Data Type	3-4
	MedLoc Data Type	3-7
	Using the Supplied VSI Implementations	3-10
	VSI_FILE	3-10
	VSI_CLIENT	3-12
	VSI_OTHER	3-13
Chapter 4	Video Database Schema	
	In This Chapter	4-3
	Inventory Tables	4-7
	VidMDGeneral	4-10
	VidMedia	4-11
	VidPVideo	4-14
	VidPAudio	4-17
	VidPOther	4-19
	Subject Matter Tables	4-20
	VidChunkMD	4-22
	VidAudDesc	4-24
	VidImgDesc	4-26
	VidTextDesc	4-28
	VidVidDesc.	4-30
	VidStrata	4-31
	Lookup Tables	4-32
	VidStandard	4-32

VidFormat	4-34
VidACodec	4-35
VidVCodec	4-37

Chapter 5 **Server Functions**

In This Chapter	5-3
After	5-5
AfterOrEqual	5-7
Before	5-9
BeforeOrEqual	5-11
ClientAccess	5-13
Compare	5-16
Contained.	5-20
Contains	5-23
Copy	5-26
EndP	5-28
Equal	5-30
Fps	5-32
Frame	5-34
GreaterThan	5-36
GreaterThanOrEqual	5-38
Hour	5-40
Intersection	5-42
LessThan	5-44
LessThanOrEqual	5-46
Location	5-48
MedChunkFrom	5-50
MedPtrFrom	5-53
MedStorage	5-57
MedTCFrom	5-59
Meets	5-62
Minus	5-64
Minute.	5-69
Native	5-71
New.	5-73
NotContained	5-75
NotContains	5-77
NotEqual	5-79
NotMeets	5-81
NotOverlap	5-84
Overlap	5-86
Plus	5-89

Second	5-92
Shutdown	5-94
Size	5-96
StartP.	5-98
Startup	5-100
Status.	5-102
Storage	5-104
StorageID	5-106
Time	5-108
Type (MedLoc)	5-110
Type (MedTC).	5-112
VidFrameNumberFrom	5-114

Chapter 6 **Client API**

In This Chapter	6-3
The Client API Library	6-3
Video Foundation Class	6-4
Client Functions	6-6
medChunkGetEndPtr	6-9
medChunkGetSize	6-11
medChunkGetStartPtr	6-13
medChunkMinus	6-15
medChunkPlus	6-19
medFrameNumberToMedPtr	6-22
medFrameNumberToMedTC	6-24
medLocGetLocation.	6-26
medLocGetStorageID	6-28
medLocGetType	6-30
medPtrGetNative	6-32
medPtrGetTime	6-34
medPtrMinus	6-36
medPtrPlus.	6-38
medPtrToFrameNumber	6-40
medPtrToMedChunk	6-43
medPtrToMedTC.	6-45
medPtrsToMedChunk	6-47
medTCGetFPS.	6-49
medTCGetFrame.	6-52
medTCGetHour	6-54
medTCGetMinute	6-56
medTCGetSecond	6-58
medTCGetType	6-60

	medTCMinus	6-62
	medTCPlus	6-64
	medTCToFrameNumber	6-66
	medTCToMedPtr	6-68
	medTimeNativeToMedPtr	6-70
Chapter 7	Examples	
	In This Chapter	7-3
	Preparing to Run the Examples	7-4
	Registering the Video Storage Server Component	7-4
	Inserting Physical Attributes	7-5
	Copy 1: BetaSP Tape	7-6
	Copy 2: MPEG File	7-6
	Inserting Abstract Descriptions	7-8
	Inserting Explicit Values	7-9
	Inserting SELECT Statement Results	7-11
	Issuing Queries	7-12
	Selecting the MediaID Value of a Media Instance	7-12
	Selecting Media Chunks in a Single Strata	7-12
	Selecting Media Chunks that Overlap with a Strata	7-13
	Selecting a Newly Constructed Media Chunk	7-14
	Requesting Access to a Segment	7-15
	Tracking Nondigital Multimedia Assets	7-15
Chapter 8	Writing a Virtual Storage DataBlade Module	
	In This Chapter	8-3
	Registering the DataBlade Module with Informix	8-4
	Implementing a Media Point UDT	8-4
	Defining, Writing, and Registering the VSI UDRs	8-5
	Device-Independent VSI UDRs	8-6
	Defining VSI UDR Objects	8-7
	Writing VSI UDRs	8-9
	Registering VSI UDRs with the Informix Video Foundation Module	8-14
	Documenting the Virtual Storage DataBlade Module	8-17
Chapter 9	VSI Programming Interface Reference	
	In This Chapter	9-3
	The VSI Programming Interface Library	9-3
	MedLoc_fd Data Type	9-3

Functions	9-4
MedLocClose	9-7
MedLocOpen	9-8
MedLocRead	9-10
MedLocSeek	9-11
MedLocTell.	9-13
MedLocWrite	9-15

Glossary

Error Messages

Index

Introduction

In This Introduction	3
About This Manual.	3
Organization of This Manual	4
Types of Users	5
Software Dependencies	5
Documentation Conventions	6
Typographical Conventions	6
Icon Conventions	8
Naming Conventions.	8
Function Syntax Conventions	9
Additional Documentation	9
Printed Documentation	10
Release Notes, Documentation Notes, and Machine Notes	10
Related Reading	11
Informix Welcomes Your Comments	12

In This Introduction

This introduction describes the *Informix Video Foundation DataBlade Module User's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout.

About This Manual

The Informix Video Foundation DataBlade module extends the capabilities of Informix Dynamic Server with Universal Data Option, allowing you to store, manage, and manipulate video data and metadata in the same system in which you store more traditional data.

The *Informix Video Foundation DataBlade Module User's Guide* describes the following features of this DataBlade module:

- The *temporal component*, consisting of data types and their associated functions that provide a format-independent interface between Informix Dynamic Server and video data
- The *storage component*, consisting of data types and their associated functions that provide a device-independent interface between Dynamic Server and large media objects located on specialized storage servers such as the database server's file system, the client file system, video tape recorders (VTRs), Web video streamers, or high-performance video servers
- The *video database schema*, optimized for storage and manipulation of video data and metadata
- The *client API*, providing client-side functions and C++ objects for video application developers

This section discusses the organization of the manual, the intended audience, and the associated software products that you must have to use this DataBlade module.

Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual, describes the documentation conventions used, and explains the generic style of this documentation.
- [Chapter 1, “Overview of the Video Foundation DataBlade Module,”](#) provides a brief introduction to the features of this DataBlade module, and describes the principles and concepts behind the Informix Video Foundation DataBlade module as a whole.
- [Chapter 2, “Temporal Component,”](#) describes the data types and functions that allow Informix Dynamic Server and your video application or DataBlade module to work with video data, independently of the data’s format.
- [Chapter 3, “Storage Component,”](#) describes the data types and functions that allow Dynamic Server and your video application or DataBlade module to work with large media objects, independently of their format and the storage server on which they reside.
- [Chapter 4, “Video Database Schema,”](#) describes the database tables that allow Dynamic Server and your video application or DataBlade module to store and manage video data and metadata, independently of the data’s format or location.
- [Chapter 5, “Server Functions,”](#) provides a reference to all the SQL-callable functions provided by the Informix Video Foundation DataBlade module.
- [Chapter 6, “Client API,”](#) describes the client-side functions and C++ objects for video application developers, including a reference to the functions.
- [Chapter 7, “Examples,”](#) provides an example of how a particular company might integrate the Informix Video Foundation DataBlade module into their media management system.

- [Chapter 8, “Writing a Virtual Storage DataBlade Module,”](#) describes the process of writing a device-specific DataBlade module that implements the Virtual Storage Interface (VSI) of the storage component.
- [Chapter 9, “VSI Programming Interface Reference,”](#) describes the device-independent C library that enables DataBlade module developers to directly access MedLoc data. This chapter includes a reference to the functions.
- Also included in this guide are a glossary containing terms defined in this guide and common video technology terms, a listing of error messages returned by the functions, and a comprehensive index.

Types of Users

This manual is written for the following types of developers:

- Developers writing client video applications
- Developers writing content-processing, virtual storage, or other video DataBlade modules that depend on the data types and interfaces provided by the Informix Video Foundation DataBlade module
- System integrators providing a complete media management system that includes Informix Dynamic Server and the Informix Video Foundation DataBlade module



Tip: *If you are reading this guide because you are a DataBlade module developer, you should read and be familiar with the Informix [DataBlade Developers Kit User’s Guide](#).*

This manual is also a helpful resource for database administrators managing installations that use Informix Dynamic Server and the Informix Video Foundation DataBlade module.

Software Dependencies

To use the Informix Video Foundation DataBlade module, you must be using the appropriate versions of Dynamic Server and the LOB Locator DataBlade module. (See [“Release Notes, Documentation Notes, and Machine Notes”](#) on [page 10](#).)

In addition, this DataBlade module must be installed on your system before you can use any video or virtual storage DataBlade modules.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix documents.

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Naming conventions
- Function syntax conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
<i>italics</i> italics <i>italics</i>	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface boldface	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
monospace <i>monospace</i>	Information that the product displays and information that you enter appear in a monospace typeface.

(1 of 2)



Convention	Meaning
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
◆	This symbol indicates the end of product- or platform-specific information.
→	This symbol indicates a menu item. For example, “Choose Tools→Options ” means choose the Options item from the Tools menu.

(2 of 2)

Tip: When you are instructed to “enter” characters or to “execute” a command, immediately press RETURN after the entry. When you are instructed to “type” the text or to “press” other keys, no RETURN is required.

Icon Conventions

Throughout the documentation, you will find text that is identified by icons in the margin. These icons identify three types of information, as described in the following table. This information is always displayed in *italics*.

Icon	Description
	Identifies paragraphs that contain vital instructions, cautions, or critical information.
	Identifies paragraphs that contain significant information about the feature or operation that is being described.
	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described.

Naming Conventions

The Informix Video Foundation DataBlade module uses the following naming conventions for its data types, functions, and schema tables:

- Informix has reserved the prefixes *Med* and *Vid* for its multimedia and video-specific data types. For example, the data type that represents a specific point in video data is *MedPtr*.
- The prefixes *Med* and *Vid* are also reserved for generic multimedia and video-specific functions. These prefixes are dropped if at least one of the function inputs is a user-defined data type provided by this DataBlade module. For example, the **Native** function takes a *MedPtr* value as input, so it does not have a prefix. But the **MedPtrFrom** function that takes an *INTEGER* value and a *FLOAT* value as input uses the *Med* prefix to distinguish it from functions in other DataBlade modules that might otherwise use the same name and arguments.

- The schema tables provided by the temporal component have the prefix *Vid*, and those provided by the storage component have the prefix *MedVSI*. For example, the table in which each video program is registered is **VidMDGeneral**, and the table in which each storage server is registered is **MedVSIRegister**.

Function Syntax Conventions

This guide uses the following conventions to specify DataBlade function syntax:

- Square brackets ([]) surround optional items.
- Curly brackets ({ }) surround items that can be repeated.
- A vertical line (|) separates alternatives.
- Comma-separated lists of values are indicated by a parameter value with a *_commalist* suffix. For example, you might use `colorlist=color_comma` to indicate `colorlist=red, blue, green`.
- Function parameters are italicized; arguments that you must specify as shown are not italicized.

These function syntax conventions used in context look like this:

```
myfunction([optionalarg], {repeatablearg}, set | noset,
myparam=my_comma)
```

DataBlade functions are used in SQL statements to manipulate the specialized data types that DataBlade modules define. These syntax conventions are used in [Chapter 5, “Server Functions,”](#) the function reference chapter of this guide.

Additional Documentation

This section lists other documentation resources that you might find useful.

Printed Documentation

The following Informix documents complement the information in this manual:

- *Informix Video Foundation Interfaces Supplement*
- *Administrator's Guide for Informix Dynamic Server*
- *Informix Guide to SQL: Reference, Informix Guide to SQL: Syntax, Informix Guide to SQL: Tutorial*
- *DataBlade Developers Kit User's Guide*
- *DataBlade Module Installation and Registration Guide*
- *LOB Locator DataBlade Module Programmer's Guide*
- *DataBlade API Programmer's Manual*
- *Informix ESQL/C Programmer's Manual*
- *C++ Interface Programmer's Guide*

Release Notes, Documentation Notes, and Machine Notes

The following on-line files, located in the **\$INFORMIXDIR/extend** subdirectory for this DataBlade module, supplement the information in this manual.

On-Line File	Purpose
Documentation notes	The documentation notes describe features that are not covered in the manual or that have been modified since publication.
Release notes	The release notes describe feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
Machine notes	The machine notes describe any special actions that are required to configure and use Informix products on your computer.

Please examine these files; they contain vital information about application and performance issues.

Related Reading

For information on object-relational database management systems, see *Object-Relational DBMSs: The Next Great Wave*, by Michael Stonebraker with Dorothy Moore (Morgan Kaufmann Publishers, 1996).

For technical information on database management, see *An Introduction to Database Systems*, by C. J. Date (Addison-Wesley Publishing, 1995).

To learn more about the SQL language, consider the following books:

- *A Guide to the SQL Standard*, by C. J. Date with H. Darwen (Addison-Wesley Publishing, 1993)
- *Understanding the New SQL: A Complete Guide*, by J. Melton and A. Simon (Morgan Kaufmann Publishers, 1993)
- *Using SQL*, by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

For information on digital video, see *A Technical Introduction to Digital Video*, by Charles A. Poynton (John Wiley & Sons, 1996).

For information on video database technology, see *Video Database Systems: Issues, Products, and Applications*, by Ahmed K. Elmagarmid, Haitao Jiang, Abdelsalam A. Helal, Anupam Joshi, and Magdy Ahmed (Kluwer Academic Publishers, 1997).

For more information on the concept of stratification that is introduced in [Chapter 1, “Overview of the Video Foundation DataBlade Module,”](#) consult the following articles:

- “The Stratification System: A Design Environment for Random Access Video,” by Thomas G. Aguierre Smith and Gloriana Davenport, *Network and Operating Systems Support for Digital Audio and Video*, Third International Workshop Proceedings, La Jolla, California, USA, 12–13 November 1992. (Edited by P. Venkat Rangan, Berlin, Germany: Springer-Verlag, 1993, pp. 250–61.)
- “Cinematic Primitives for Multimedia (Stratification Helps Storytelling in Multimedia),” by Gloriana Davenport, Thomas G. Aguierre Smith, and Natalio Pincever, *IEEE Computer Graphics and Applications*, volume 11, number 4 (July 1991), pp. 67–74.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

`doc@informix.com`

We appreciate your suggestions.

Overview of the Video Foundation DataBlade Module

In This Chapter	1-3
What Is the Informix Video Foundation DataBlade Module?	1-3
Definition of Terms	1-3
Features of This DataBlade Module	1-6
The Temporal Component	1-7
The Storage Component	1-7
The Video Database Schema	1-9
The Client API.	1-16
Who Are the Users of the Informix Video Foundation DataBlade Module? .	1-17
Video Application Developers	1-17
Virtual Storage DataBlade Module Developers	1-17
Content-Processing DataBlade Module Developers	1-18
What Are Some Sample Uses of This DataBlade Module?	1-20
Video Courseware.	1-20
TV Station Web Site	1-21

In This Chapter

This chapter provides an overview of the Informix Video Foundation DataBlade module and explains concepts you should know before you use it.

What Is the Informix Video Foundation DataBlade Module?

The Informix Video Foundation DataBlade module extends the capabilities of Informix Dynamic Server with Universal Data Option, allowing you to store, manage, and manipulate *video data* and *metadata* in the same system in which you store more traditional data. In this guide, such a system is referred to as a *media management system*.

This section first defines the italicized words in the previous paragraph and the term *foundation*, then it describes the features of this DataBlade module.

Definition of Terms

Video data represents an image that changes continuously in time. Unlike traditional data that is retrieved and displayed as a discrete value, video data is incrementally accessed and is displayed as a real-time stream of frames.

Video metadata is “data about data.” In this DataBlade module, there are two categories of metadata:

- The physical attributes of the media on which the video data is stored, such as length and format
- Abstract descriptions of the video data’s content, such as text descriptions or graphic images that represent the data, that are assigned and arranged in a process called *stratification*

These concepts are described in more detail later in this chapter.

This DataBlade module is referred to as a *foundation* DataBlade module because it provides a base on which new or specialized video technologies can be quickly added or changed. Because the foundation is open, secure, and scalable, it provides a clear path toward reusing and repurposing valuable video assets among video formats and distribution channels.

A *media management system* is a collection of processes, computers, and other machines that enable you to store and manipulate video (and often, other media) data along with more traditional data, such as text. The Informix Video Foundation DataBlade module provides interfaces among the basic elements of such a media management system. These are the basic elements; see [Figure 1-1](#) for an illustration:

- A video client application, such as an editing or logging application, running on a stand-alone computer.
- Informix Dynamic Server and the Informix Video Foundation DataBlade module, running on a second computer; the database contains the video metadata. (Although they are not shown in [Figure 1-1](#), other Informix and third-party DataBlade modules—such as Informix Image, Text, and Web DataBlade modules, and third-party storage server and video scene-change detection DataBlade modules—can also be running on this computer.)
- Video data located on one or more storage servers, such as Web video streamers, VTRs (video tape recorders), or video servers. Generally, each storage server is hosted on a separate computer. In [Figure 1-1](#), there is just one storage server: a high-performance video server.

Media management systems, their elements, and their configuration can vary greatly: sometimes two or more of these elements reside and execute on the same computer or on multiple computers. Furthermore, the responsibility for the system might be shared by separate developers or teams or might belong to an individual programmer.

Figure 1-1
Basic Media Management System

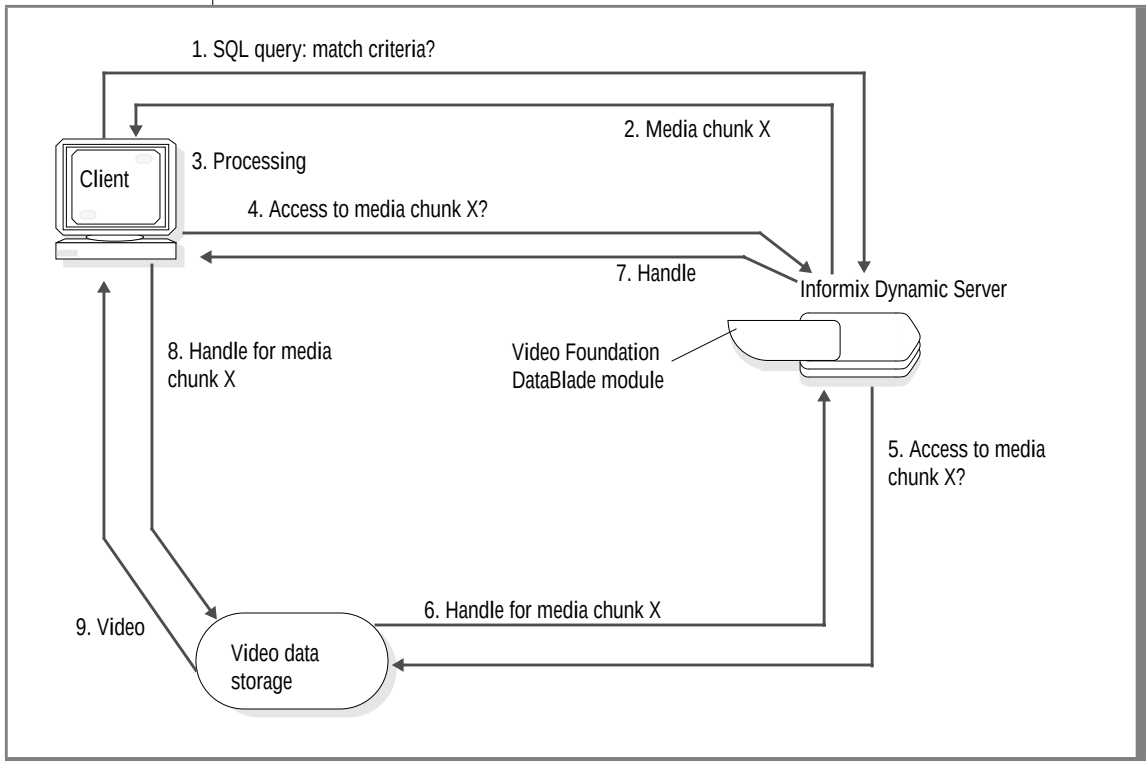


Figure 1-1 also shows the flow of control and data among the elements of a media management system. This flow remains the same regardless of a particular system's configuration or division of labor. In particular, once permissions have been granted by the storage server and Dynamic Server, transmission of the video data to the client occurs without interruptions from Dynamic Server.

Figure 1-1 shows the following scenario (terms introduced here are defined later in this chapter):

1. Using the Informix Video Foundation DataBlade module's data types and server functions, and referring to the foundation's video database schema, the client application queries Dynamic Server, asking if there are any media chunks meeting criteria **b** and **c**.
2. Dynamic Server responds to the client, specifying that media chunk **X** matches criteria **b** and **c**.
3. Using the foundation's client API, the client processes media chunk **X** (perhaps converting it to another data type).
4. The client application again queries Dynamic Server, requesting access to media chunk **X**.
5. Using the Informix Video Foundation DataBlade module and the appropriate virtual storage DataBlade module, Dynamic Server requests access to the large media object that corresponds to media chunk **X** from the specified storage server.
6. The storage server returns a handle to Dynamic Server.
7. Dynamic Server returns the handle to the client application.
8. The client application uses the handle to contact the storage server and request the video program (media chunk **X**) for playback.
9. The storage server transmits the video program directly to the client application, without interruption from Dynamic Server.

The client can work with the video as appropriate for that application and its privileges: playback, fast-forward, rewind, and so on.

Features of This DataBlade Module

This section describes the following features of the Informix Video Foundation DataBlade module (for an example of how these features can be used to manage, query, and access a media management system, see [Chapter 7, "Examples"](#)):

- The *temporal component*, consisting of data types and their associated functions that provide a format-independent interface between Informix Dynamic Server and video data

- The *storage component*, consisting of data types and their associated functions that provide a device-independent interface between Dynamic Server and large media objects located on specialized storage servers, such as the database server's file system, the client file system, video tape recorders (VTRs), Web video streamers, or high-performance video servers
- The *video database schema*, optimized for storage and manipulation of video data and metadata
- The *client API*, providing client-side functions and C++ objects for video application developers

The Temporal Component

The temporal component provides user-defined data types and functions that enable Informix Dynamic Server to reference specific points in time (*media points* and *timecodes*) and intervals (*media chunks*) within video data, regardless of the data's format (AVI, MPEG, and so on).

Dynamic Server differentiates between one media point and another in time units of nanoseconds. However, media points can be specified in time units ranging from hours to nanoseconds, in SMPTE timecodes, in frame numbers, or in all of these, as required by the video application or DataBlade module.

A range of media points, or an interval, is referred to in this guide as a *media chunk*. Media chunks are built from media points, so they can be as precisely defined as needed. For example, one chunk might be a one-nanosecond interval, while another might be a five-minute interval.

The functions provided by the temporal component allow Dynamic Server and your video application or DataBlade module to perform arithmetic, Boolean, temporal comparison (before, after, and so on), and conversion operations among these video data types.

The Storage Component

The storage component's user-defined data types and functions provide a device- and format-independent *Virtual Storage Interface (VSI)* between Informix Dynamic Server, large media objects, and those objects' storage servers in a media management system.

The term *large media object* refers to any type of video, audio, or other media data that is to come under database control. The term *storage server* refers to any repository of large media objects that is under database control, whether it be a file system, a video source, or a physical storage device. Examples of storage servers include VTRs, the database server's file system, the client computer's file system, Web video streamers, and high-performance video servers.

This component allows you to transparently save and access video data, regardless of its format or its location. In addition, it provides several other important features:

- If the connection between Dynamic Server and the server is secure, video data is secure, and its integrity is preserved as it is for any data in the database.
- Because the interface is device independent, there is just a single point of administration for user data, video data, and metadata, no matter where it is stored.

For each storage server type under database control, a device-specific implementation of the VSI—also known as a *virtual storage DataBlade module* or as a *Video Foundation Interface*—must be registered with the Informix Video Foundation DataBlade module. The Informix Video Foundation DataBlade module provides the following implementations:

- **VSI_FILE**, for large media objects accessed using Dynamic Server's underlying file system
- **VSI_CLIENT**, for large media objects accessed using the current client's file system (for example, an editing system)
- **VSI_OTHER**, for large media objects not accessible by Dynamic Server, such as VTR media or data on a CD-ROM

Other virtual storage DataBlade modules are available from third-party vendors.

This guide provides information on how to use the device-independent VSI and the three supplied VSI implementations. In addition, this guide describes how to write a device-specific virtual storage DataBlade module using the storage component.

The Video Database Schema

The video database schema is a set of database tables that ties together the Temporal and storage components' data; these tables are optimized for use by Dynamic Server.

More importantly, however, this schema provides a repository for the following types of video metadata:

- The physical attributes of the media on which the video data is stored, such as duration and format.
- Abstract descriptions of the video data's content. These abstract descriptions are also known as *proxies*, and they can be in the form of text annotations, graphic images, audio clips, and video clips. These descriptions can be grouped into meaningful layers called *strata*.

The extent to which you use the video database schema is entirely up to you and the needs of your video application or DataBlade module. You might need to keep track of just your video inventory's physical attributes, or you might be creating a more complex system that chooses particular media chunks by matching a set of abstract descriptions. The following sections discuss these cases in more detail.

Tip: Keep in mind that most third-party video DataBlade modules use this schema. If your work depends on theirs, you will probably need to use it too.

Physical Attributes

The video database schema provides a set of tables referred to as the *inventory tables*. These tables contain columns for various types of physical video attributes:

- Title
- Format
- Capture rate
- Duration
- Number of audio channels
- Frame width and height
- Location

A *proxy* is a description that “stands in” for a piece of data. Usually, a proxy is much smaller and is of lower resolution than the video data it represents.



With these tables, you can provide simple inventory control of the video data and a format-independent way of referencing the data.

Inventory Control

In the simplest case, you can use the inventory tables to keep track of your video inventory: insert the video data into one or more storage servers and then register its locations and other attributes in the inventory tables.

The following terms describe the forms your video data can take, from abstract concept to physical files.

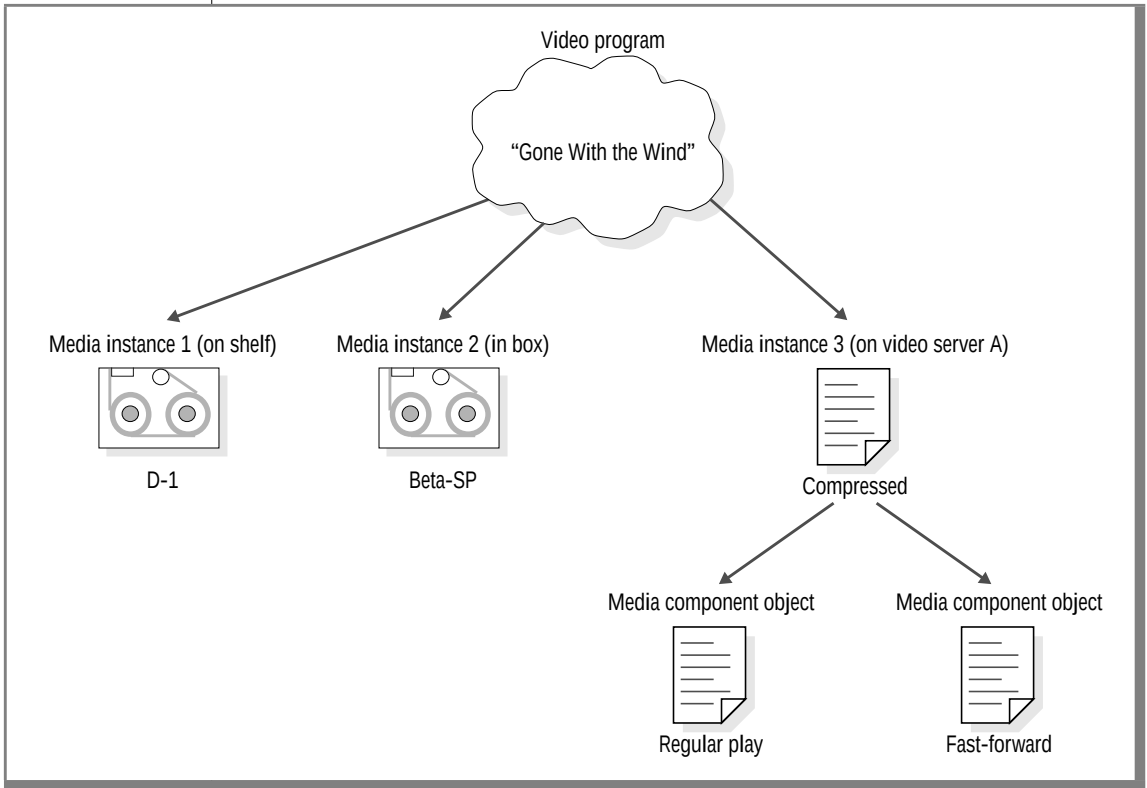
Term	Definition
<i>video program</i>	The format-independent concept of the entire “piece” of video, whether it be a movie, a newscast, a video clip, or even just a sequence of frames.
<i>media instance</i>	A physical copy of the video program, in whole or in part, regardless of the copy’s format. For example, if you have one copy of a video program on Beta-SP tape and two copies in D-1 format, you have three media instances. Similarly, if you have three segments of the program, you have three media instances.
<i>media component objects</i>	The large media objects, such as video output files, that constitute a media instance. Some video formats use just one video output file per media instance, while others use separate or multiple video, audio, and other output files.

Figure 1-2 shows an example of these terms. A particular site is setting up its inventory control system and is inserting its first video program, *Gone With the Wind*. This site has three media instances, or copies, of this program:

- Media instance 1 is in D-1 format and is located on the shelf.
- Media instance 2 is in Beta-SP format, but it is located in a box.
- Media instance 3 is in a compressed format on video server A.

There are two media component objects for media instance 3: a regular play video file, and a fast-forward video file.

Figure 1-2
Storing Physical Metadata



The definition of *media instance* in the table on [page 1-10](#) raises the issue of storing various formats of the same video program in the database. You can, of course, keep a simple inventory by registering each copy as a unique piece of video data. However, the Informix Video Foundation DataBlade module provides a format-independent way of referencing video data, which is described in the next section.

For an illustration of the concepts described in this section, see [Figure 1-5](#) and its accompanying text in “[Video Metadata](#)” on [page 1-14](#).

Format-Independent Referencing of Video Data

Using the inventory tables, you can enable format-independent referencing of video data by making use of the Informix Video Foundation DataBlade module’s *foundation timeline*.

To understand this concept, first consider that a piece of video data (as defined in “[What Is the Informix Video Foundation DataBlade Module?](#)” on [page 1-3](#)) can be seen as a piece of time. The logical view of “a piece of time” is the *timeline*, a horizontal line divided into regularly spaced time units.

Because the Informix Video Foundation DataBlade module stores video data in nanoseconds, the logical view of this data is a timeline divided into nanosecond units: the *foundation timeline*.

The Informix Video Foundation DataBlade module creates the foundation timeline using the video program’s media instance that you have specified is the *source timeline*. The foundation timeline becomes a “translation layer” among the various formats in the database: the Informix Video Foundation DataBlade module “knows” how the formats relate to the foundation; thus it knows how they relate to each other.

Say, for example, that you have a copy of a video program on a Beta-SP tape. You insert this instance into the video database schema, indicating that it is the source timeline. Using internal data types and conversion functions, the Informix Video Foundation DataBlade module uses the Beta-SP timeline to create the foundation timeline (time units of nanoseconds) for the video program. [Figure 1-3](#) shows the relationship of the two timelines.

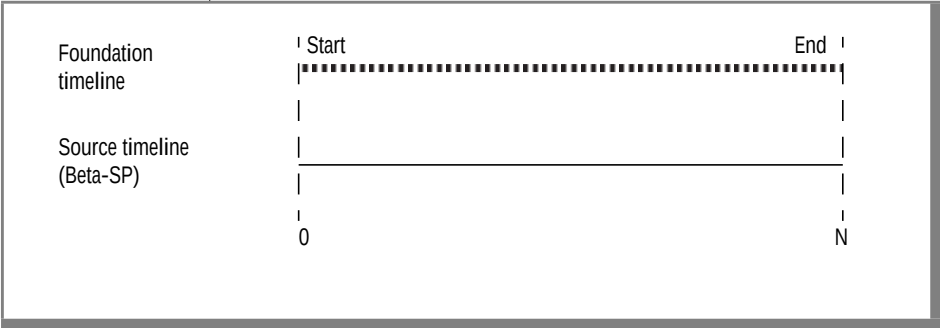


Figure 1-3
*Relationship of
Source Timeline to
Foundation
Timeline*

Perhaps a subsequent instance of the video program consists of the last five minutes of the program in MPEG format. In this case, the MPEG instance is said to have a *nonzero offset* in relation to the source timeline (the Beta-SP instance). This is shown in [Figure 1-4](#).

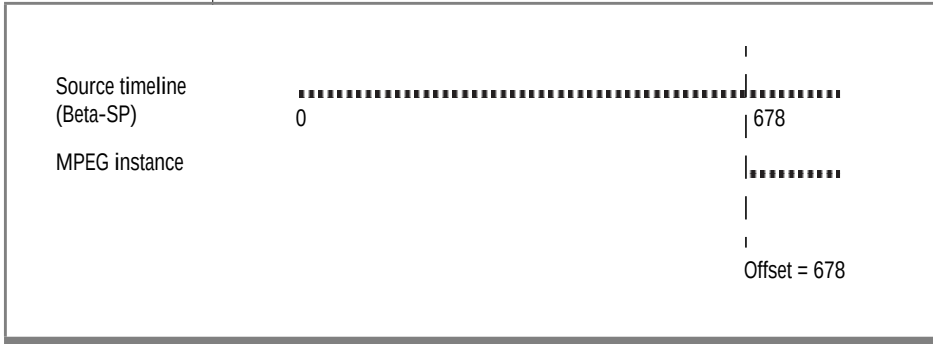


Figure 1-4
Relationship of
Partial Media
Instance to Source
Timeline

Abstract Descriptions and Stratification

In addition to the inventory tables, the video database schema also provides a set of *subject matter tables*. These tables allow you to create and store relationships between a piece of the timeline—a media chunk—and one or more of the following abstract descriptions:

- Text annotations
- Graphic images
- Audio clips
- Video clips

The process of creating such relationships is known as *stratification*. When you create a relationship between a media chunk and a description, you have created a *stratum*. When you collect some of these relationships into a meaningful group, you have created a *strata*. (Stratification can occur either at the user interface level—someone is sitting at a console creating relationships—or as a result of a computerized process, such as an automated scene change detection and logging application.)

For another example of the concepts described in this section, see [Figure 1-5](#) and its accompanying text in “[Video Metadata](#)” on [page 1-14](#).

For example, say you have a two-minute media chunk, called A, that shows “Joe taking a walk.” You create relationships between this chunk and one or more of the following abstract descriptions:

- An audio clip consisting of the sounds of footsteps
- A still image, showing just Joe’s face
- A video clip that is a composite derived from the media chunk
- A text annotation that says “Joe taking a walk”
- Another text annotation that says “This media chunk’s copyright is owned by FooBar Incorporated”

You have another media chunk, called B. It starts halfway through chunk A, ends five minutes later, and shows Mary joining Joe on his walk. You create relationships between this chunk and the same audio clip, video clip, and copyright text descriptions as for chunk A. In addition, you add these unique abstract descriptions:

- A still image, showing just Mary’s face
- A text annotation that says “Mary walking with Joe”

Later, perhaps another user or process collects these relationships into the following meaningful groups:

- A “Walking” strata, including both chunks A and B
- A “Joe” strata, including chunk A and other chunks showing only Joe

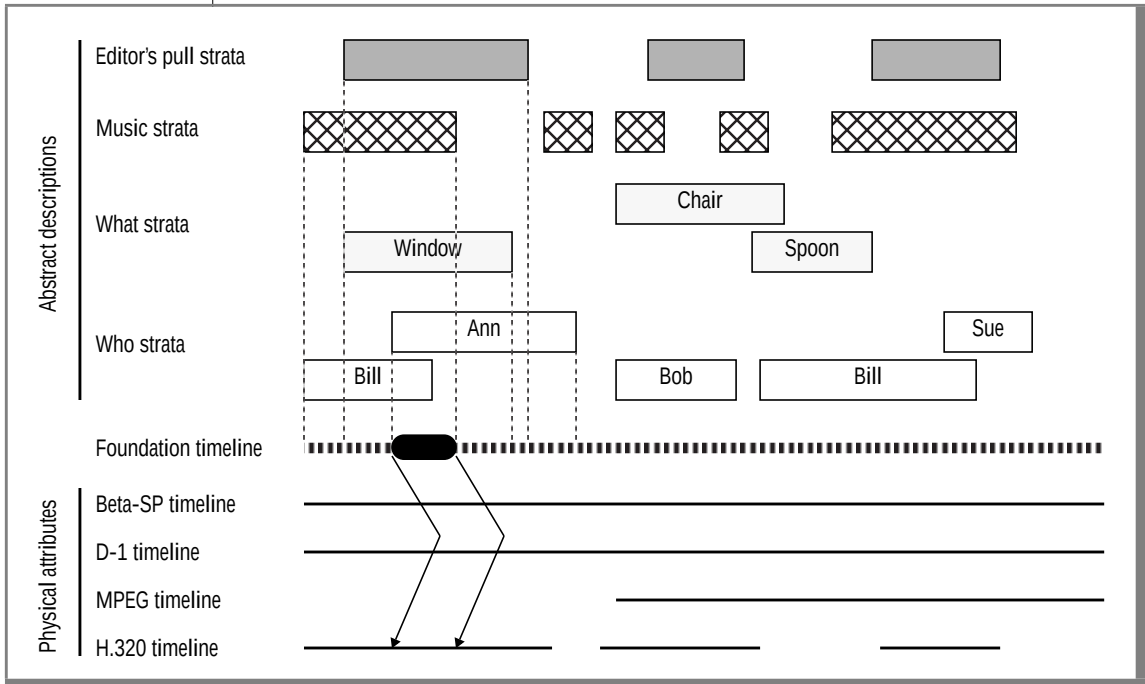
The stratification process can be repeated as many times as necessary, and can continue for as long as necessary, for a particular application. As each user or process creates relationships and strata, they are only adding value to the video inventory; future users and processes will be able to select media chunks of interest with ever-increasing precision.

Video Metadata

This section contains an example and an illustration combining the concepts described in the following two sections.

Figure 1-5 shows a video program that has been entered into a media management system that includes the Informix Video Foundation DataBlade module.

Figure 1-5
Video Metadata



The following media instances of this program exist:

- The original media instance (the source timeline) was Beta-SP tape.
- The foundation timeline was derived from the source timeline.
- Another media instance is in D-1 format; this copy contains the entire video program.
- Another media instance is in MPEG-2 format. However, this instance contains just the last portion of the video program, meaning it has a nonzero offset in relation to the source timeline.
- The last three instances are segments of the video program in H.320 format. The first segment has a zero offset in relation to the source timeline, but the other two have nonzero offsets.

Various users or processes have been working on this video program. Someone created relationships between media chunks and descriptions on who is in each one, then created a Who strata. The music director created relationships among media chunks and descriptions of the music they contain. Finally, the editor created relationships among media chunks and descriptions of their appropriateness for inclusion in the current project.

The black marker on the foundation timeline identifies the result of a query that asked, “I want a chunk that is part of the editor’s pull, that includes music, the window, and Ann.” (Someone else might have formed a more explicit query, asking for Ann+Bill, or Ann-Bill.) Only one chunk was returned, and it is available in Beta-SP, D-1, or H.320 format. It is not available in MPEG format.

The Client API

The Informix Video Foundation DataBlade module provides a client API to assist you in those portions of your video application or DataBlade module that execute on the client computer.

This API provides the Informix Video Foundation DataBlade module’s opaque data types as value objects for developers using Informix’s C++ interface.

In addition, it provides a subset of this DataBlade module’s server functions as client-side functions. They provide the same services as their server-side counterparts, but they execute on your client system rather than being processed by Informix Dynamic Server. These functions are available to developers using Informix’s ESQL/C, DataBlade API, or C++ interfaces.

For technical information on this API, see [Chapter 6, “Client API.”](#)

Who Are the Users of the Informix Video Foundation DataBlade Module?

There are three categories of users of this DataBlade module:

- Video application developers
- Virtual storage DataBlade module developers
- Content-processing and other video DataBlade module developers

A fourth category, system integrators, is really a combination of the other three. This section introduces each type of user.

Video Application Developers

These developers use the Informix Video Foundation DataBlade module, other video-related DataBlade modules, and sometimes other-technology (such as Web and Image) DataBlade modules to create an application for an end user of the media management system.

Video applications are typically event-driven, run on a client computer, and present a GUI to the end user. These applications often allow the end user to insert video data into the storage servers and the video metadata into the video database schema, and access (seek, cue, fast-forward) the video data on the storage server. Video applications can take advantage of the foundation's client API to perform local processing of data returned by Informix Dynamic Server.

Examples of video applications are logging and browsing programs.

Virtual Storage DataBlade Module Developers

These developers are usually concerned solely with providing a storage-server-specific implementation of the Informix Video Foundation DataBlade module's VSI, allowing Informix Dynamic Server to manipulate that storage server.

A virtual storage DataBlade module executes on the same computer hosting Dynamic Server and the Informix Video Foundation DataBlade module. Examples of these types of DataBlade modules include the VSI_FILE implementation provided by the Informix Video Foundation DataBlade module and the fictitious third-party Sample Virtual Storage DataBlade module described in [Chapter 3, “Storage Component.”](#)

For the most part, the work of these developers is transparent to the other users of the video foundation, but these developers need to document a few “hooks” to their interface. Information on writing this type of DataBlade module, including information on the hooks that must be documented, is provided in [Chapter 8, “Writing a Virtual Storage DataBlade Module.”](#)

Content-Processing DataBlade Module Developers

These developers use the Informix Video Foundation DataBlade module, other video-related DataBlade modules, and sometimes other-technology (such as Text and Image) DataBlade modules to create a content-processing DataBlade module. The content-processing DataBlade module is then incorporated into other DataBlade modules or video applications.

Content processing is a broad term that refers to any activity—such as editing, annotating, and parsing—that video content can go through in its life cycle from raw footage, to digitized data, to final video-based product.

There is always at least one output from content processing; [Figure 1-6](#) is a simplified flowchart of video content processing:

1. The content-processing DataBlade module receives inputs of raw video footage and previously stored video data.
2. The processing occurs, outputting either media chunks, metadata, data types, or all three.
3. The media chunks are discarded or stored, possibly for further processing (in many media management systems, a *low*-resolution copy of the video data is processed, but a *high*-resolution copy is stored and used as the “final product” of that system).

4. The metadata is stored in the video database schema.

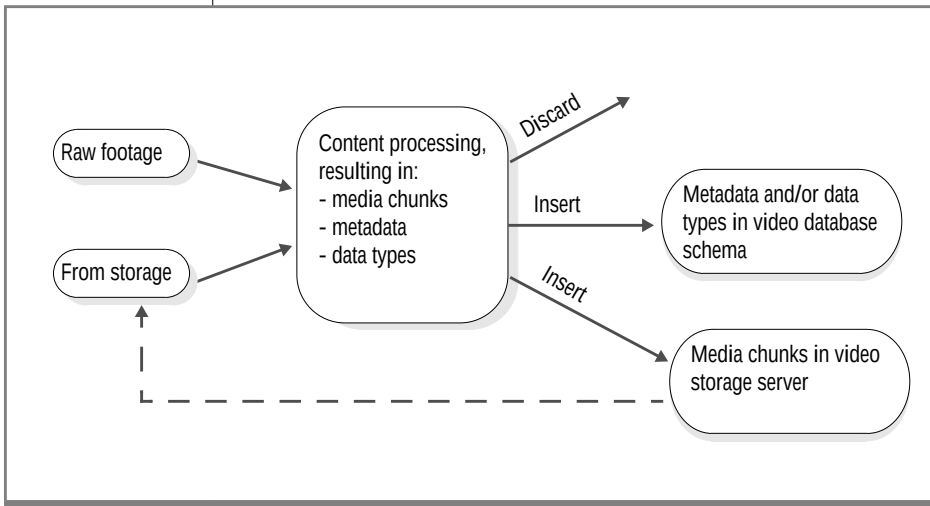


Figure 1-6
Flowchart of
Content Processing

In most media management systems, not all video data is processed. For example, a scene-change-detection DataBlade module might output the frames of a video program that do not contain any camera motion. A face-recognition DataBlade module might accept those frames as input and then output only those that show close-ups of faces.

Because content processing covers such a wide range of potential activities, these developers' tasks and concerns cannot be as narrowly defined as they are for the virtual storage DataBlade module developers. One issue they must all address, however, is where the content processing should take place: on the client computer or on the computer hosting Dynamic Server.

Informix offers these recommendations: if processing occurs only once, let it run on the client computer and store the intermediate results of the processing in the database as data types. This technique is particularly appropriate if such one-time processing activities have application-dependent *thresholds*; that is, specific data points that determine such things as how much metadata is generated on the client and how much of what was generated is stored in the database. By storing intermediate results in the database, the DataBlade module can be less sensitive about the threshold, letting the end user of the media management system—through the user interface, for example—make threshold decisions. In addition, this technique minimizes the effects of CPU-intensive processing on the database engine.



Important: If content processing occurs on the client computer, *MedPtr* values must be calculated before using the *MedPtr* data type and reporting a temporal event, such as a scene change. Information on performing this calculation is provided in the “VidMedia” section, on [page 4-12](#).

What Are Some Sample Uses of This DataBlade Module?

This section provides two hypothetical situations of how a media management system, with the help of the Informix Video Foundation DataBlade module, can use Informix Dynamic Server in its day-to-day operations.

For a more complete example of how a particular company might integrate the Informix Video Foundation DataBlade module into their media management system, see [Chapter 7, “Examples.”](#)

Video Courseware

Imagine that a university has decided to provide distance learning through a multimedia Web site. Professor Hurlihy’s basic anthropology course is being used to test this delivery mechanism.

The university’s database administrator inserts Professor Hurlihy’s collection of case-study interviews into the media management system. The DBA inserts the actual video data into a storage server and then inserts information on the titles, video formats, capture rate, and so on, into the video database schema’s inventory tables.

Repurposing is the process of creating a new video program by recycling and recombining existing video data.

The professor browses the case studies, selecting media chunks of interest for her first anthropology lesson. She creates relationships between these chunks and short, descriptive text annotations and then collects these relationships into a strata called “Lesson 2.” Next, Professor Hurlihy’s teaching assistants create their own strata, sometimes by repurposing some of the professor’s relationships. When the stratification is complete, the video and other media is loaded onto the class Web site.

Students in the class access the lesson through the Web; their login IDs indicate their access levels. If a student has not completed Lesson 1, for example, access to Lesson 2 is blocked. When the students gain access, they view the lesson based on the professor’s and the teaching assistants’ strata, and they create their own relationships and strata.

Back at the university, the professor keeps track of which students have accessed the Web site, collects the students’ relationships into a class-specific strata, and assesses the usefulness of particular video programs. In addition, the DBA gathers usage statistics to plan his or her future video data management tasks.

TV Station Web Site

As another example, perhaps a public television station has an external Web site that includes the station’s independently produced programs. All paying members of the television station fill out a user profile form and can access all of the available programs on the Web site.

Other visitors to the site can access only “teaser” clips of the full programs. If they decide to become paying members—perhaps using advanced Web features such as digital cash and transaction security to do so—they, too, fill out a user profile form and can access all available programs.

Although paying members can access all available programs, however, perhaps the Web site offers the option of providing tailored content. That is, by analyzing the user profile form and the visitor’s history at the Web site, it can start suggesting or showing particular programs based on the visitor’s interests.

Such analysis and activities are possible because of the strata in the database. Various media chunks were associated with abstract descriptions by various people and processes, such as the producer, the video logging application, the producer’s assistants, and the video editing application.

This public television Web site engages the visitor with an interactive experience that is more than just a passive viewing of short video clips, ensuring frequent visits to the site. This dynamic, stratified approach to “customer intimacy” easily extends to dynamic advertising and Internet commerce applications as well.

Temporal Component

In This Chapter	2-3
MedPtr Data Type	2-3
External Representation	2-4
Functions	2-5
MedTC Data Type	2-7
External Representation	2-7
Functions	2-9
MedChunk Data Type	2-12
External Representation	2-12
Functions	2-13
Conversion Functions	2-18

An *opaque data type* is a fundamental data type of a predefined fixed or variable length whose internal structure is not accessible through SQL statements.



In This Chapter

The temporal component of the Informix Video Foundation DataBlade module is a collection of user-defined data types and functions that enable Informix Dynamic Server to reference, store, and manipulate the following objects:

- Locations, described as *media points* or *timecodes*, within time-based, continuous media data—especially video—regardless of the medium’s format
- Specific user-defined intervals, or *chunks*, within media data

These data types and their associated functions are the subject of this chapter. In addition, a table showing the functions that allow you to convert values among the temporal component data types is included.

MedPtr Data Type

MedPtr is an opaque user-defined data type that describes a precise location in time, or *media point*, within a video program. The media point is described by the following elements:

- Its temporal value
Temporal value is the chronological number assigned to the media point, as numbered from the beginning of the video program. This value can be specified in units ranging from hours to nanoseconds, but it is always stored in the database in nanoseconds.
- Optionally, a format-specific string or identifier, such as the temporal value’s SMPTE-timecode equivalent

Important: For those applications or devices using the SMPTE timecode unit, the MedTC data type has also been provided; see “[MedTC Data Type](#)” on page 2-7.

External Representation

MedPtr has the following external representation:

Time[*Unitflag*][*-Native*]

Time A 64-bit integer, greater than or equal to 0, that specifies the media point's temporal value. This value is derived from the video program's original medium, before any compression occurs.

Only this element of MedPtr is interpreted and manipulated by Dynamic Server, allowing for format-independent indexing.

Unitflag An optional, single-character flag specifying the *Time* element's time unit. If no flag is specified, milliseconds is the default unit, although the value is always stored in nanoseconds. (Milliseconds is often a more convenient unit for video applications, but nanoseconds provide the resolution required to index digital video and audio.)

This flag can have one of the following values:

C or c	Microseconds (10^{-6}), for example, 25000000C.
M or m	Milliseconds (10^{-3}), for example, 25000M. This is the default.
N or n	Nanoseconds (10^{-9}), for example, 25000000000N.
S or s	Seconds, for example, 25S.

-Native

An optional 32-character string that is not interpreted by Dynamic Server. If this string is supplied, the preceding hyphen delimiter character (-) must be present.

Usually this string specifies the native, precise representation of a specific point in any continuous media, such as the *Time* element's SMPTE-timecode equivalent or a byte offset.

However, because Dynamic Server ignores this string, it can be used for whatever you want.

Informix recommends that you avoid using the pipe character (|) in the *Native* string. It is allowed in the MedPtr data type, but in the MedChunk data type it is allowed only as a separator between the starting and ending MedPtr values (see [“External Representation” on page 2-12](#)).

The following are examples of MedPtr values. Although each one has a different *Native* element, as far as Dynamic Server is concerned, they reference the same point:

- 1933S
- 1933000M-1933
- 1933000000C
- 1933000000000N-00:32:13:19

Functions

This section lists the server functions provided for the MedPtr data type (for reference information, see [Chapter 5, “Server Functions”](#)). These functions enforce the following rules for the MedPtr data type:

- Dynamic Server evaluates only the *Time* elements of MedPtr values; the *Native* elements are not interpreted.
- A *Time* element must be greater than or equal to 0. Thus, if any operation would result in a negative MedPtr value, an error is raised.

Several of these functions have corresponding client API functions; see [Chapter 6, “Client API,”](#) for details.

- Dynamic Server handles all MedPtr values it receives as input to a particular function as if they are from the same video program. For example, if you add a MedPtr value from one video program to a MedPtr value of another program, you might receive a valid MedPtr value in return, but that value might not represent an existing piece of media.

Function	Description
Time	Takes a MedPtr value and returns its <i>Time</i> element as an INT8 value.
Native	Takes a MedPtr value and returns its <i>Native</i> element as an LVARCHAR value.
MedPtrFrom	Takes inputs of various types and returns a new MedPtr value.
Plus	Takes two MedPtr values, adds them, and returns a MedPtr value.
Minus	Takes two MedPtr values, subtracts the second one from the first, and returns a MedPtr value.
Equal	Takes two MedPtr values and returns a true value if the <i>Time</i> elements reference the same point.
NotEqual	Takes two MedPtr values and returns a true value if the <i>Time</i> elements do not reference the same point.
Compare	Takes two MedPtr values and returns an INTEGER value describing the relationship of the two <i>Time</i> elements: 0 if they are equal; -1 if the first MedPtr is less than the second; 1 if the first MedPtr is greater than the second.
LessThan	Takes two MedPtr values and returns a true value if the <i>Time</i> element of the first MedPtr value is less than the second.
LessThanOrEqual	Takes two MedPtr values and returns a true value if the <i>Time</i> element of the first MedPtr value is less than or equal to the second.

(1 of 2)

Function	Description
GreaterThan	Takes two MedPtr values and returns a true value if the <i>Time</i> element of the first MedPtr value is greater than the second.
GreaterThanOrEqual	Takes two MedPtr values and returns a true value if the <i>Time</i> element of the first MedPtr value is greater than or equal to the second.

(2 of 2)

Although internal support functions are not usually documented in a DataBlade module’s user’s guide, the following functions can be useful in video applications or DataBlade modules, so they are listed here for your convenience.

Function	Description
MedPtrIn	Takes an LVARCHAR value and returns a MedPtr value.
MedPtrOut	Takes a MedPtr value and returns an LVARCHAR value.

For details on these functions, refer to the [DataBlade Developers Kit User’s Guide](#).

MedTC Data Type

MedTC is an opaque user-defined data type that describes a precise location in time within a video program, based on the location’s SMPTE-standard timecode.

External Representation

Important: The default video standard for this data type is drop-frame NTSC (National Television Standards Committee), also known as compensated mode. Frames 00 and 01 are dropped at the start of each minute, except for minutes that are even multiples of 10. For example, 01:07:59:29 is followed by 01:08:00:02, but 01:09:59:29 is followed by 01:10:00:00.

SMPTE stands for Society of Motion Picture and Television Engineers.



MedTC has the following external representation:

HH:MM:SS:FF[-Standard_flag]

<i>HH</i>	A two-character number specifying the hours element of the timecode. Valid values are 00 through 99.
:	The colon character (:) is the standard delimiter character, but you can also use the semicolon character (;) or the period (.).
<i>MM</i>	A two-character number specifying the minutes element of the timecode. Valid values are 00 through 59.
<i>SS</i>	A two-character number specifying the seconds element of the timecode. Valid values are 00 through 59.
<i>FF</i>	A two-character number specifying the frame number of the timecode. The minimum value for this element is 00. The maximum depends on the frames-per-second rate (indicated by <i>Standard_flag</i>), but is one of 23, 24, or 29. If no optional flag is supplied after the <i>FF</i> element, the default drop-frame NTSC video standard is assumed.
<i>-Standard_flag</i>	An optional flag specifying the video standard in use, which determines the frames-per-second rate and the valid values of this timecode's <i>FF</i> element (see previous). If no flag is specified, drop-frame NTSC is the default standard (see next). If a flag is supplied, the preceding hyphen delimiter character (-) must be present. This flag can have one of the following values: -D or -d Drop-frame NTSC. This is the default. The frames-per-second rate is 29.97, and valid values of <i>FF</i> are 02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.

- F or -f Film. The frames-per-second rate is 24, and valid values of *FF* are 00 through 23.
- N or -n NTSC. The frames-per-second rate is 30, and valid values of *FF* are 00 through 29.
- P or -p PAL (Phase Alternation Line). The frames-per-second rate is 25, and valid values of *FF* are 00 through 24.

The following values are examples of MedTC values:

10;03:40;01-N	Specifies the NTSC video standard; valid frame numbers are 00 through 29. In this example, the semicolon character (;) is used as a delimiter.
10:03:40:02	No flag is specified; the default drop-frame NTSC video standard is assumed, and valid frame numbers are 02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.
00.10.64.24-P	Specifies the PAL video standard; valid frame numbers are 00 through 24. In this example, the period character (.) is used as a delimiter.
01:23:12:04-F	Specifies film; valid frame numbers are 00 through 23.

Functions

This section lists the server functions provided for the MedTC data type (for reference information, see [Chapter 5, “Server Functions”](#)). These functions enforce the following rules for the MedTC data type:

- A MedTC value must be greater than or equal to 0. Thus, if any operation results in a negative MedTC value, an error is raised.
- The maximum *HH:MM:SS* elements for a MedTC value are 99:59:59. Thus, if any operation results in a MedTC value greater than this, an error is raised.
- MedTC values must have the same frame rate (that is, their *Standard_flag* elements must be the same) to be arithmetically operated on or compared; otherwise, an error is raised.

Several of these functions have corresponding client API functions; see [Chapter 6, “Client API,”](#) for details.

- For each frames-per-second rate, there is a maximum frame number *per timecode value* (see [“External Representation” on page 2-7](#) for details). There is also a *maximum cumulative* number of frames per frame rate, meaning that a single video program cannot have more frame numbers than this maximum. This can become an issue when you convert between data types, for example, between MedPtr nanosecond values and MedTC values.

The following table shows this maximum number per frame rate.

Frame Rate	Maximum Cumulative Frame Number
24	8,640,000
25	9,000,000
29.97	10,789,200
30	10,800,000

- Dynamic Server handles all MedTC values it receives as input to a particular function as if they are from the same video program. For example, if you add a MedTC value from one video program to a MedTC value of another program, you might receive a valid MedTC value in return, but that value might not represent an existing piece of media.

Function	Description
Hour	Takes a MedTC value and returns its <i>HH</i> (hours) element as an INTEGER value.
Minute	Takes a MedTC value and returns its <i>MM</i> (minutes) element as an INTEGER value.
Second	Takes a MedTC value and returns its <i>SS</i> (seconds) element as an INTEGER value.
Frame	Takes a MedTC value and returns its <i>FF</i> (frame number) element as an INTEGER value.

(1 of 2)

Function	Description
Type	Takes a MedTC value and returns its <i>Standard_flag</i> element as an LVARCHAR value.
Fps	Takes a MedTC value and returns its frames-per-second rate as a FLOAT value.
MedTCFrom	Takes inputs of various temporal data types and returns a new MedTC value.
Plus	Takes two MedTC values, adds them, and returns a MedTC value.
Minus	Takes two MedTC values, subtracts the second one from the first, and returns a MedTC value.
Equal	Takes two MedTC values and returns a true value if they reference the same point.
NotEqual	Takes two MedTC values and returns a true value if they do not reference the same point.
Compare	Takes two MedTC values and returns an INTEGER value describing their relationship: 0 if they are equal; -1 if the first MedTC value is less than the second; 1 if the first MedTC value is greater than the second.
LessThan	Takes two MedTC values and returns a true value if the first MedTC value is less than the second.
LessThanOrEqual	Takes two MedTC values and returns a true value if the first MedTC value is less than or equal to the second.
GreaterThan	Takes two MedTC values and returns a true value if the first MedTC value is greater than the second.
GreaterThanOrEqual	Takes two MedTC values and returns a true value if the first MedTC value is greater than or equal to the second.

(2 of 2)

Although internal support functions are not usually documented in a DataBlade module's user's guide, the following functions can be useful in video applications or DataBlade modules, so they are listed here for your convenience.

Function	Description
MedTCIn	Takes an LVARCHAR value and returns a MedTC value.
MedTCOut	Takes a MedTC value and returns an LVARCHAR. value.

For details on these functions, refer to the [DataBlade Developers Kit User's Guide](#).

MedChunk Data Type

MedChunk is an opaque user-defined data type that represents a user-defined interval, or *chunk*, in any continuous medium, such as a video program. The media chunk is described by a starting media point and an ending media point, both of type MedPtr.

External Representation

MedChunk has the following external representation:

StartP|*EndP*

- *StartP* and *EndP* are valid MedPtr values; see “[MedPtr Data Type](#)” on [page 2-3](#).
- *StartP* and *EndP* are assumed to be from the same video program.
- The two MedPtr values must be separated by the pipe character (|). (Thus, the *Native* elements of the MedPtr values cannot contain a pipe character.)
- *EndP* must be greater than or equal to *StartP*; if they are equal, the single-point MedChunk is the equivalent of a single MedPtr value.

The following is an example of a MedChunk data type, where the *Time* element of both MedPtr values is specified in seconds, and the *Native* element is an SMPTE-standard timecode for a video using the drop-frame NTSC standard:

```
20s-00:00:20:00-D| 50s-00:00:50:00-D
```

Functions

This section lists the server functions provided for the MedChunk data type (for reference information, see [Chapter 5, “Server Functions”](#)). These functions enforce the following rules for the MedChunk data type:

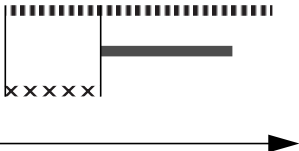
- MedChunk values received as input must be valid, as described in [“External Representation” on page 2-12](#).
- Dynamic Server evaluates only the *Time* elements of the MedPtr values that make up the MedChunk values; the *Native* elements are not interpreted.
- Dynamic Server handles all MedChunk values it receives as input to a particular function as if they are from the same video program. For example, if you add a MedChunk value from one video program to a MedChunk value of another program, you might receive a valid MedChunk value in return, but that value might not represent an existing piece of media.

The following table lists the MedChunk server functions. For many of the functions, a picture of one of the true or non-null value conditions is shown. For pictures of all true or non-null value conditions for a particular function, see [Chapter 5, “Server Functions.”](#)

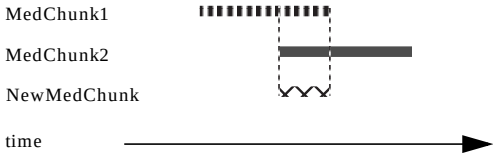
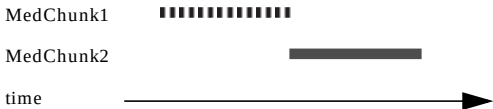
Function	Description
StartP	Takes a MedChunk value and returns its starting MedPtr value (the <i>StartP</i> element).
EndP	Takes a MedChunk value and returns its ending MedPtr value (the <i>EndP</i> element).
Size	Takes a MedChunk value and returns its size (<i>EndP</i> minus <i>StartP</i>) in nanoseconds.

(1 of 5)

Several of these functions have corresponding client API functions; see [Chapter 6, “Client API,”](#) for details.

Function	Description
MedChunkFrom	Takes inputs of various types and returns a new MedChunk value.
Plus	Takes two MedChunk values and returns a new MedChunk value: its <i>StartP</i> element is the lesser of the two <i>StartP</i> inputs, and its <i>EndP</i> is the greater of the two <i>EndP</i> inputs. MedChunk1 MedChunk2 NewMedChunk time 
Minus	Takes two MedChunk values and returns a new MedChunk value consisting of those points of the first MedChunk value not shared with the second MedChunk value. MedChunk1 MedChunk2 NewMedChunk time 
Equal	Takes two MedChunk values and returns a true value if their <i>StartP</i> and <i>EndP</i> values reference the same point. MedChunk1 MedChunk2 time 
NotEqual	Takes two MedChunk values and returns a true value if their <i>StartP</i> or <i>EndP</i> values do not reference the same point. MedChunk1 MedChunk2 time 

(2 of 5)

Function	Description
Compare	Takes two MedChunk values and returns an INTEGER value describing the relationship of the two <i>StartP</i> elements: 0 if they are equal; -1 if the first MedChunk is less than the second; 1 if the first MedChunk is greater than the second.
Intersection	Takes two MedChunk values and returns a new MedChunk value consisting of the points in common between them. MedChunk1 MedChunk2 NewMedChunk time 
Meets	Takes two MedChunk values and returns a true value if the first MedChunk value meets the second, either at its start or end. MedChunk1 MedChunk2 time 
NotMeets	Takes two MedChunk values and returns a true value if the first MedChunk value does not meet the second, either at its start or end. MedChunk1 MedChunk2 time 
Overlap	Takes two MedChunk values and returns a true value if the first MedChunk value has any points in common with the second. MedChunk1 MedChunk2 time 

Function	Description
NotOverlap	Takes two MedChunk values and returns a true value if the first MedChunk value has no points in common with the second. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time —————▶
Contains	Takes two MedChunk values and returns a true value if the first value contains the second: the first <i>StartP</i> element is less than or equal to the second, <i>and</i> the first <i>EndP</i> element is greater than or equal to the second. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time —————▶
NotContains	Takes two MedChunk values and returns a true value if the second MedChunk value has any points not shared by the first. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■■■■■■■■■ time —————▶
Contained	Takes two MedChunk values and returns a true value if the first value is contained in the second: the first <i>StartP</i> element is greater than or equal to the second, <i>and</i> the first <i>EndP</i> element is less than or equal to the second. MedChunk1 ■■■■■■■■ MedChunk2 ■■■■■■■■■■■■■■■■ time —————▶

(4 of 5)

Function	Description
NotContained	Takes two MedChunk values and returns a true value if the first MedChunk value has any points not shared by the second. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time _____➡
Before	Takes two MedChunk values and returns a true value if the first EndP element is before the second StartP element. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time _____➡
BeforeOrEqual	Takes two MedChunk values and returns a true value if the EndP element of the first MedChunk value is before or equal to the StartP element of the second MedChunk value. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time _____➡
After	Takes two MedChunk values and returns a true value if the first StartP element is after the second EndP element. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time _____➡
AfterOrEqual	Takes two MedChunk values and returns a true value if the StartP element of the first MedChunk value is after or equal to the EndP element of the second MedChunk value. MedChunk1 ■■■■■■■■■■ MedChunk2 ■■■■■■■■ time _____➡

Although internal support functions are not usually documented in a DataBlade module's user's guide, the following can be useful in video applications or DataBlade modules, so they are listed here for your convenience.

Function	Description
MedChunkIn	Takes an LVARCHAR value and returns a MedChunk value.
MedChunkOut	Takes a MedChunk value and returns an LVARCHAR. value.

For details on these functions, refer to the [DataBlade Developers Kit User's Guide](#).

Conversion Functions

The temporal component provides the following conversion functions:

- **MedChunkFrom**, which returns a MedChunk value
- **MedPtrFrom**, which returns a MedPtr value
- **MedTCFrom**, which returns a MedTC value
- **VidFrameNumberFrom**, which returns an INTEGER frame number

This table shows which data types each conversion function accepts as input. See each function's reference page in [Chapter 5, "Server Functions,"](#) for complete information on its input and return values.

Conversion Function	Input Data Types Accepted			
	MedPtr	MedTC	MedChunk	Other
MedPtrFrom		Yes		Yes
MedTCFrom	Yes			Yes
MedChunkFrom	Yes			Yes
VidFrameNumberFrom (integer)	Yes	Yes		

In this table, *Other* refers to any built-in data types these functions might take as inputs, such as INTEGER frame numbers, FLOAT frame rates, and LVARCHAR strings.

Storage Component

In This Chapter	3-3
The Device-Independent VSI	3-4
MedStorage Data Type	3-4
External Representation	3-5
MedVSIRegister Table	3-6
Functions	3-7
MedLoc Data Type	3-7
External Representation	3-8
Functions	3-9
Using the Supplied VSI Implementations	3-10
VSI_FILE	3-10
VSI_CLIENT.	3-12
VSI_OTHER.	3-13

In This Chapter

The storage component provides the device- and format-independent *Virtual Storage Interface (VSI)* between Informix Dynamic Server, large media objects, and those objects' storage servers in a media management system.

The term *large media object* refers to any type of video, audio, or other media data that is to come under database control. The term *storage server* refers to any repository of large media objects that is under database control, whether it be a file system, a video source, or a physical storage device. Examples of storage servers include VTRs (video tape recorders), Web video streamers, and high-performance video servers.

Although the *interface* that the VSI provides is device independent, the particular activities that occur on a storage server when this interface is used are device specific. Thus, for each storage server type under database control, a device-specific implementation of the VSI must be registered with the Informix Video Foundation DataBlade module. Such an implementation is referred to as a *virtual storage DataBlade module* or as a *Video Foundation Interface*.

This chapter describes the device-independent VSI data types and functions that you use to reference and manipulate large media objects and the storage servers on which they reside. It then describes the three built-in, generalized VSI implementations (VSI_FILE, VSI_CLIENT, and VSI_OTHER).

For information on Informix VSI implementations for popular storage servers, see the *Informix Video Foundation Interfaces Supplement*. For information on third-party VSI implementations, see the vendors' documentation.

If your media management system includes third-party virtual storage DataBlade modules, read this chapter in conjunction with the user documentation for those modules. If you are writing a virtual storage DataBlade module, see [Chapter 8, "Writing a Virtual Storage DataBlade Module."](#)

The Device-Independent VSI

The device-independent VSI provides data types and functions that allow you to reference and manipulate storage servers and large media objects. These data types and functions are the subject of this section.

Some of the examples in this section are based on the fictitious third-party Sample Virtual Storage DataBlade module, which implements the VSI for the fictitious Sample Company's SAM 4000-PRO video server.

MedStorage Data Type

MedStorage is a named row complex data type that describes a storage server, including the information needed for Dynamic Server to connect to the storage server. This data type forms the rows of the **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)).

A *complex data type* is built from elements of other types; these elements can be accessed with SQL statements. One form that a complex data type can take is that of a *row data type*, whose elements take the form of *fields*. Fields are similar to database table columns: they have a name and a data type associated with them. If a row type can be distinguished from other row types by a name (as the MedStorage row type can be), it is known as a *named row data type*.

This section defines the MedStorage data type and the **MedVSIRegister** table in which it is used; then it lists MedStorage's associated functions.

External Representation

The MedStorage named row type consists of the following fields. The valid values for each field are determined by the provider of a storage server-specific VSI implementation. See [“Using the Supplied VSI Implementations” on page 3-10](#) for information on what is valid for the VSI_FILE, VSI_CLIENT, and VSI_OTHER implementations.

Field Name	Type	Sample Data	Explanation
Type	CHAR (18)	'SAM'	Type of storage server. The first three letters must match a DataBlade module prefix registered with Informix, or the entire string can be VSI_FILE (see “VSI_FILE” on page 3-10).
StorageID	INTEGER	1	Identification number of this storage server; must be a unique number among storage servers of this Type . (For example, there can be type FOO #1 and type SAM #1, but there cannot be two FOO #1s.)
ConnectionName	LVARCHAR	'SAM-4000-1'	Device-specific identifier; for example, a network port address.
UserID	LVARCHAR	'root'	Login name for Dynamic Server access to storage server.
Password	LVARCHAR	'root1234'	Password associated with UserID for this storage server.
ConnectionString	LVARCHAR	'host4:port3'	Additional information needed to connect to the storage server.
Vendor	CHAR (30)	'Sample Company'	Name of the vendor that manufactured this storage server.
Model	CHAR (30)	'4000-PRO, 3.1'	Model identifier for this storage server; can consist of name, number, version number, and so on.

For an example MedStorage row value, see [page 3-6](#).

MedVSIRegister Table

The **MedVSIRegister** table is a table of MedStorage row values, one for each supported storage server in the media management system. VSI functions use this table to locate the specified storage server.



Important: Each separate storage server (except *VSI_CLIENT* and *VSI_OTHER* implementations) in your media management system that has large media objects stored in the video database schema must be represented by a row in the **MedVSIRegister** table. This row must have a unique combination of **MedStorage Type** and **StorageID** field values. If there are several servers of the same type (for example, if there are several SAM 4000-PRO video servers in the system), they can share the same implementation of VSI functions by specifying the same value in the **MedStorage Type** field. Usually, a database administrator performs these tasks when setting up your media management system.

Column Name	Type	Sample Data	Explanation
Storage	MEDSTORAGE	MedStorage('SAM', 1, 'SAM-4000-1', 'root', 'root1234', NULL, 'Sample Company', '4000-PRO.3.1')	MedStorage row value describing the storage server.

The following example shows a MedStorage row value being inserted into the **MedVSIRegister** table for the first SAM 4000-PRO video server in a media management system. (The **MedStorage** function returns a MedStorage row value; see [“Functions” on page 3-7](#).)

```
insert into MedVSIRegister values(  
    MedStorage('SAM', 1, 'SAM-4000-1', 'root', 'root1234',  
        'host4:port3', 'Sample Company', '4000-PRO, 3.1'));
```



Functions

The following table lists the server functions provided for the MedStorage data type (for reference information, see [Chapter 5, “Server Functions”](#)).

Tip: Some third-party virtual storage DataBlade modules provide additional administrative capabilities. For information, consult the user’s guide for each virtual storage DataBlade module in your system.

Function	Description
MedStorage	Takes data for each field of a MedStorage data type and returns a MedStorage row value.
Startup	Takes a MedStorage row value and returns a storage-server-specific INTEGER value. This value indicates the status of the connection between Dynamic Server and the storage server after an attempt to initialize that connection.
Shutdown	Takes a MedStorage row value and returns a storage-server-specific INTEGER value. This value indicates the status of the connection between Dynamic Server and the storage server after an attempt to close that connection and release its associated resources.
Status	Takes a MedStorage row value and returns a storage-server-specific INTEGER value indicating the status of the connection between Dynamic Server and the storage server.

MedLoc Data Type

MedLoc is an opaque user-defined data type that represents a large media object, such as a video or audio file, stored on a storage server. The elements of a MedLoc value uniquely identify the storage server and the location, in storage-server-specific syntax, of the large media object.



Warning: When a database containing MedLoc values is unloaded or exported, just those database values are affected. The actual media objects on the storage server are not affected. Similarly, when MedLoc values are reloaded or imported into the database, no consistency checking is performed between the database values and the actual storage objects. If you are using a third-party virtual storage DataBlade module, consult its user documentation to see if it provides archival and consistency checking utilities.

External Representation

The MedLoc data type has the following external representation:

Type[*#StorageID*]: *Location*

Type A text string, with a maximum length of 18 characters, identifying the type of storage server on which this large media object resides.

This string must match one of the **Type** field values in the **MedVSIRegister** table, or it can have a value of **VSI_CLIENT** or **VSI_OTHER** (even though these implementations do not require rows in **MedVSIRegister**; see “[VSI_CLIENT](#)” on page 3-12 and “[VSI_OTHER](#)” on page 3-13).

#StorageID An integer value, preceded by the pound-sign character (#), that identifies on which storage server of type *Type* this large media object resides. This number must match one of the **StorageID** column values for this **Type** in the **MedVSIRegister** table (see “[MedVSIRegister Table](#)” on page 3-6).

If *#StorageID* is not supplied, the default value is #1.

: A colon character (:) is required before the *Location* element.

Location A variable-length text string that describes the large media object’s location in storage server-specific syntax.

For details on the syntax of this element, consult the documentation provided with each VSI implementation you are using.

These are example MedLoc values for the supplied VSI implementations:

- **VSI_FILE**#2:14
- **VSI_CLIENT**:C:\video\myfile.mpg
- **VSI_OTHER**:08/12/96-my summer vacation. VHS, 35min, #37.

Several of these functions have corresponding client API functions; see [Chapter 6, “Client API,”](#) for details.

Functions

This section lists the server functions provided for the MedLoc data type (for reference information, see [Chapter 5, “Server Functions”](#)).

Function	Description
Location	Takes a MedLoc value and returns an LVARCHAR value representing the <i>Location</i> element.
Type	Takes a MedLoc value and returns an LVARCHAR value representing the <i>Type</i> element.
StorageID	Takes a MedLoc value and returns an INTEGER value representing the <i>StorageID</i> element.
Storage	Takes a MedLoc value and returns the MedStorage row value from the MedVSIRegister table with the same Type and StorageID field values. An error is raised if no such row exists in the table. For a VSI_CLIENT MedLoc input value, creates a new MedStorage row value with the ConnectionString field set to the MedLoc value’s location. This VSI_CLIENT MedStorage row value can be used as the destination for the Copy function.
ClientAccess	Returns a MedLoc value to be used by the client application to access large media object data directly from a storage server. The client must request access operation permissions using the arguments to ClientAccess . The ability to handle particular operations varies by VSI implementation.
Copy	Returns a MedLoc value that tells Dynamic Server where to create a new large media object on the specified storage server, and where to get the source data. (The creating and copying operations are handled internally, except when the destination is VSI_CLIENT. In this case, the Copy function handles the copy operation.)
New	Returns a MedLoc value that tells Dynamic Server where to create a new, empty, large media object on the specified storage server. (The creation operation is handled internally.)

Using the Supplied VSI Implementations

The Informix Video Foundation DataBlade module provides the following VSI implementations:

- **VSI_FILE**, for large media objects accessed using the Dynamic Server file system.
- **VSI_CLIENT**, for large media objects accessed using the current client's file system (for example, an editing system). **VSI_CLIENT** location values are typically not valid for other clients; thus, do not store them in the database.
- **VSI_OTHER**, for large media objects not accessible by Dynamic Server, such as VTR media or data on a CD-ROM.

This section describes how to set up and use these implementations.

VSI_FILE

The **VSI_FILE** implementation lets you store and process large media objects using the Dynamic Server file system. Because remote clients cannot directly access this file system, **VSI_FILE** MedLoc values are primarily used for temporary storage of large media objects.

Before you use the **VSI_FILE** implementation, you must insert a row into the **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)) for each storage server of this type, and a corresponding row into the **MedVSI_File** table (see [page 3-11](#)), as follows:

- The MedStorage row value inserted into the **MedVSIRegister** table must have a valid directory path for the **VSI_FILE** MedLoc values in the **ConnectionString** field, and it must have a unique **StorageID** field among storage servers of type **VSI_FILE**.

In the following example, these fields have values of `'/videos/vsi_file/directory1/'` and `2`, respectively, and the MedStorage row value is created using the **MedStorage** function:

```
insert into MedVSIRegister values (  
    MedStorage('VSI_FILE', 2, NULL, NULL, NULL,  
        '/tmp/videos/vsi_file/directory1/', NULL, NULL));
```

- The corresponding row in the **MedVSI_File** table must have a **Path** value that matches the **MedVSIRegister** table's **ConnectionString** value and a **LastNum** value of 0. For example:

```
insert into MedVSI_File values
('/tmp/videos/vsi_file/directory1/', 0);
```

The **MedVSI_File** table has the following definition.

Column Name	Type	Explanation
Path	LVARCHAR	Identifies a unique VSI_FILE row in the MedVSIRegister table.
LastNum	INTEGER	Identifies the highest number used for a unique filename. When registering a VSI_FILE storage server, insert a value of 0 into this column. As MedLoc values for this storage server are created, this number is incremented.

After the rows are inserted into the **MedVSIRegister** and **MedVSI_File** tables, the storage server identified as VSI_FILE#2 can be used. For example, the following example shows the creation of a table called **Media**, then a large media object being copied from the client to the **/videos/vsi_file/directory1/** server directory, and finally the MedLoc value returned by the **Copy** function being inserted into the **Media** table:

```
create table media (name lvarchar, val MedLoc);

insert into Media values ('newname',
  Copy('VSI_CLIENT:C:\videos\myfile.mpg',
    Storage('VSI_FILE#2:')));
```

(The **Storage** function returns the MedStorage row value corresponding to the MedLoc 'VSI_FILE#2:'. The **ConnectionString** field for this row value is **/videos/vsi_file/directory1/**, as shown in the example on [page 3-10](#).)

Tip: For VSI_FILE MedLoc values, the **ClientAccess** function returns the same MedLoc value that was given as input, and service requests are ignored.



VSI_CLIENT

The VSI_CLIENT implementation lets you copy large media objects to and from the current client file system. The *current client* is determined by the SQL statement that is currently executing.

Because many clients may connect to the database—yet a VSI_CLIENT MedLoc value is valid only in the context of a single client connection—the following exceptions apply to the VSI_CLIENT implementation:

- Storage servers of type VSI_CLIENT are not required to have entries in the **MedVSIRegister** table.
- The MedLoc value's *StorageID* element for this *Type* element is ignored and should be omitted; thus, the following example shows a VSI_CLIENT MedLoc value:

```
'VSI_CLIENT:C:\videos\myfile.mpg'
```

Important: *Because VSI_CLIENT values are usually meaningful only for a single client, Informix recommends that you not store them in the database. Consider using a VSI_OTHER MedLoc value (next section) to store location information on large media objects that are local to a particular client computer.*

The following scenario shows how the VSI_CLIENT implementation might be used:

1. Perhaps your media management system includes a Windows NT client, a UNIX-based client, and the computer hosting Dynamic Server. You create the **Media** table, as shown on [page 3-11](#).
2. You have a large media object on the Windows NT client with a MedLoc of 'VSI_CLIENT:C:\videos\myfile.mpg'. You copy this large media object to the Dynamic Server file system and insert a reference to it in your **Media** table, as shown on [page 3-11](#) and repeated here:

```
insert into Media values ('newname',
    Copy('VSI_CLIENT:C:\videos\myfile.mpg',
        Storage('VSI_FILE#2:')));
```

The **Storage** function returns the MedStorage row value corresponding to the MedLoc 'VSI_FILE#2: '. The **myfile.mpg** large media object is copied to the directory specified in this MedStorage row value's **ConnectionString** field.



3. Now your UNIX client can select this large media object and copy it to its own file system, as follows:

```
select Copy(val,
Storage('VSI_CLIENT:/tmp/newname.mpg'))
from Media
where name='newname';
```

When the **Storage** function is called with a VSI_CLIENT MedLoc value as input, it creates a new MedStorage row value with the **ConnectionString** field set to the MedLoc value's location. The **ConnectionString** value is used as the destination for the client file copy.

In addition, because the **Copy** function is called with a destination of VSI_CLIENT, the **Copy** function handles the copying operation itself (it is not handled internally). The **Copy** function returns the MedLoc value that was used as input to the **Storage** function, meaning that this MedLoc value need not be inserted into the database.

Tip: For VSI_CLIENT MedLoc values, the **ClientAccess** function returns the same MedLoc value that was given as input, and service requests are ignored.



VSI_OTHER

The VSI_OTHER implementation allows you to reference arbitrary objects in your database, but it does not provide any support for managing or accessing those objects. This can be useful if you are storing video content on storage servers not managed by Dynamic Server (such as video tape, CD-ROM, or even cardboard boxes), but you want to store and manipulate metadata about that content in your database.

As another example, a VSI_OTHER MedLoc value can store the network node address and pathname of a large media object on a particular client computer. If a client application can use this information, it can perform remote file access using ftp or some other networking protocol.

Because large media objects (MedLoc values) on storage servers of type VSI_OTHER cannot be created, accessed, or destroyed by Dynamic Server, the following exceptions apply to the VSI_OTHER implementation:

- Storage servers of type VSI_OTHER are not required to have entries in the **MedVSIRegister** table.



- The MedLoc value's *StorageID* element for this *Type* is ignored and should be omitted; thus, the following example shows a MedLoc value for the VSI_OTHER implementation:

```
'VSI_OTHER:08/12/96-my summer vacation'
```

Tip: For VSI_OTHER MedLoc values, the **ClientAccess** function returns the same MedLoc value that was given as input, and service requests are ignored.

Video Database Schema

In This Chapter	4-3
Inventory Tables	4-7
VidMDGeneral	4-10
VidMedia	4-11
VidPVideo	4-14
VidPAudio	4-17
VidPOther	4-19
Subject Matter Tables	4-20
VidChunkMD	4-22
VidAudDesc	4-24
VidImgDesc	4-26
VidTextDesc	4-28
VidVidDesc	4-30
VidStrata	4-31
Lookup Tables	4-32
VidStandard	4-32
VidFormat	4-34
VidACodec	4-35
VidVCodec	4-37

A *proxy* is a description that “stands in” for a piece of data. Usually, a proxy is much smaller and is of lower resolution than the video data it represents.



In This Chapter

In addition to the temporal and storage components, the Informix Video Foundation DataBlade module also provides a video database schema. This schema is a set of database tables that ties together the temporal and storage components’ data, and it is installed and populated with default data when you install this DataBlade module.

The most important feature of the video database schema is that it provides a repository for the following types of video metadata:

- The physical attributes of the media on which the video data is stored, such as duration and format.
- Abstract descriptions of the video data’s content. These abstract descriptions are also known as *proxies* and can be in the form of text annotations, graphic images, audio clips, and video clips. These descriptions can be grouped into meaningful layers called *strata*.

The way in which these types of metadata are stored in the schema is the subject of this chapter.

The extent to which you use the video database schema is entirely up to you and the needs of your video application or DataBlade module. You might need to keep track of just your video inventory’s physical attributes, or you might be creating a more complex system that chooses particular media chunks based on matching a set of abstract descriptions. If you decide to use a set of tables, the primary-to-foreign key constraints are enforced.

Tip: Keep in mind that most third-party video DataBlade modules use this schema. If your work depends on theirs, you will probably need to use it too.

For an example of how a particular company might use the video database schema in their media management system, see [Chapter 7, “Examples.”](#)

The Informix Video Foundation DataBlade module schema includes the following types of tables:

- *Inventory tables*, for registration of each video program, its media instances, and the files that constitute those instances.
- *Subject matter tables*, for providing layers of abstract descriptions from which to derive complex queries of the video programs in your media management system.
- *Lookup tables*, for referencing various media standards and formats.
- *Storage component tables*, for registration of each storage server in your media management system that has large media objects stored in the video database schema; see [Chapter 3, “Storage Component.”](#)

Important: *You can, of course, add application- or DataBlade-module-specific tables to this schema, although they are not replicated if you install this DataBlade module in another database.*



The schema's tables and their relationships, not including the storage component tables, are shown in Figure 4-1.

Figure 4-1
Video Database Schema

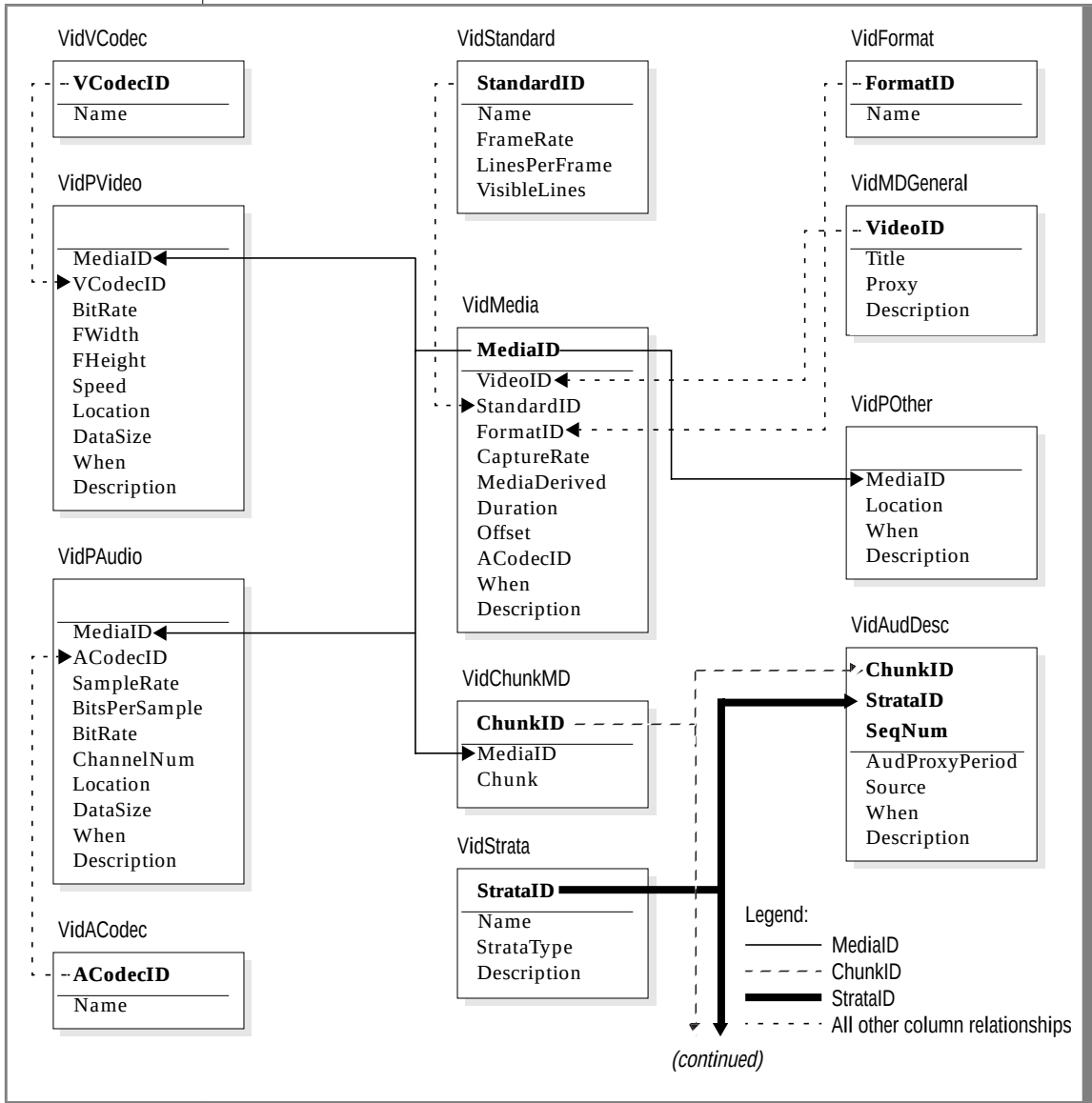
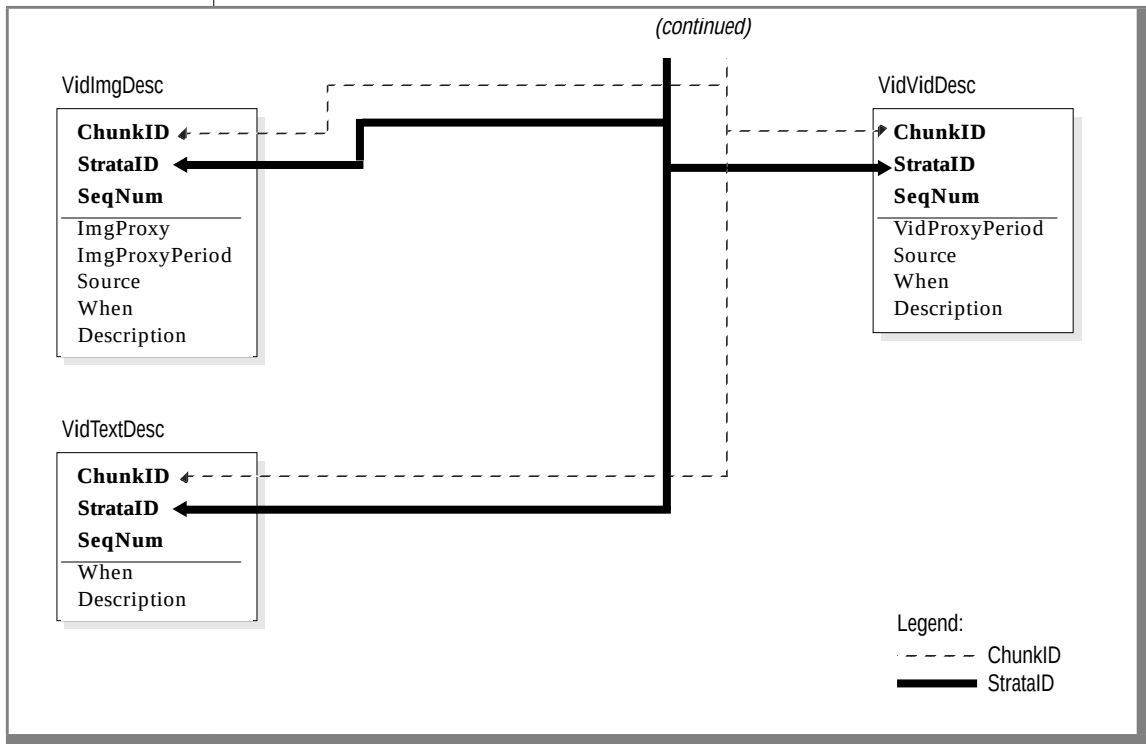


Figure 4-1 (continued)
Video Database Schema



Inventory Tables

The following terms are used in this chapter to describe the forms your video data can take, from abstract concept to physical files.

Term	Definition
<i>video program</i>	The format-independent concept of the entire “piece” of video, whether it be a movie, a newscast, a video clip, or even just a sequence of frames.
<i>media instance</i>	A physical copy of the video program, in whole or in part, regardless of the copy’s format. For example, if you have one copy of a video program on Beta-SP tape and two copies in D-1 format, you have three media instances. Similarly, if you have three segments of the program, you have three media instances.
<i>media component objects</i>	The large media objects, such as video output files, that constitute a media instance. (Large media objects are stored by Dynamic Server as the MedLoc data type; see “MedLoc Data Type” on page 3-7.) For example, some video formats use just one video output file per media instance, while others use separate or multiple video, audio, and other output files.

For each video program that is stored in the database, use the inventory tables to store metadata that describes the physical attributes of the media on which the program resides. The following steps describe how the inventory tables are typically used:

1. For each video program that is stored in the database, no matter how many copies or formats exist, insert one row into the **VidMDGeneral** table.
2. For each media instance of that video program, insert one row in the **VidMedia** table.

For example, if you have two copies of the program in Beta-SP and one in AVI format, insert three rows into this table. (The media instance from which this DataBlade module is to create the *foundation timeline* is known as the *source timeline*. For more information, see [“VidMedia” on page 4-11](#) or [“Physical Attributes” on page 1-9.](#))

Encoding is the process of digitizing, then applying some sort of minimal compression to, analog data.

RTSP stands for *real-time streaming protocol*, a proposed standard for streaming video.

3. If a media instance in **VidMedia** is encoded, insert rows into the remaining inventory tables for its media component files, as follows:
 - Insert one row in the **VidPVideo** table per video output file.
For example, there might be one video output file for the normal playback version and another for the fast-forward version of the media instance.
 - If the video program includes sound, insert at least one row in the **VidPAudio** table.
There can be more than one row per video program: for example, if audio channels are stored in separate files or if there are alternative sound tracks—say, one in English and one in Spanish.
 - If the video format produces other types of output files, insert one row in the **VidPOther** table row for each one.
For example, there might be a row in this table for the MPEG-2-Transport user stream, or for a stream of RTSP commands.
- If the media instance in **VidMedia** is *not* encoded, do not make insertions into the **VidPVideo**, **VidPAudio**, or **VidPOther** tables.

Figure 4-2 shows an example of a how a site that owns two copies of a movie might use these tables:

1. One row is inserted into **VidMDGeneral** to represent the format-independent concept of the movie *Babe*.
2. One row is inserted into **VidMedia** for each copy of the movie: one for the D-3 format, the other for the generic compressed format.
3. One row is inserted into **VidPVideo** for each of the media component files that make up the compressed media instant: one for regular playback, the other for rewind.

The rest of this section describes each inventory table in detail.

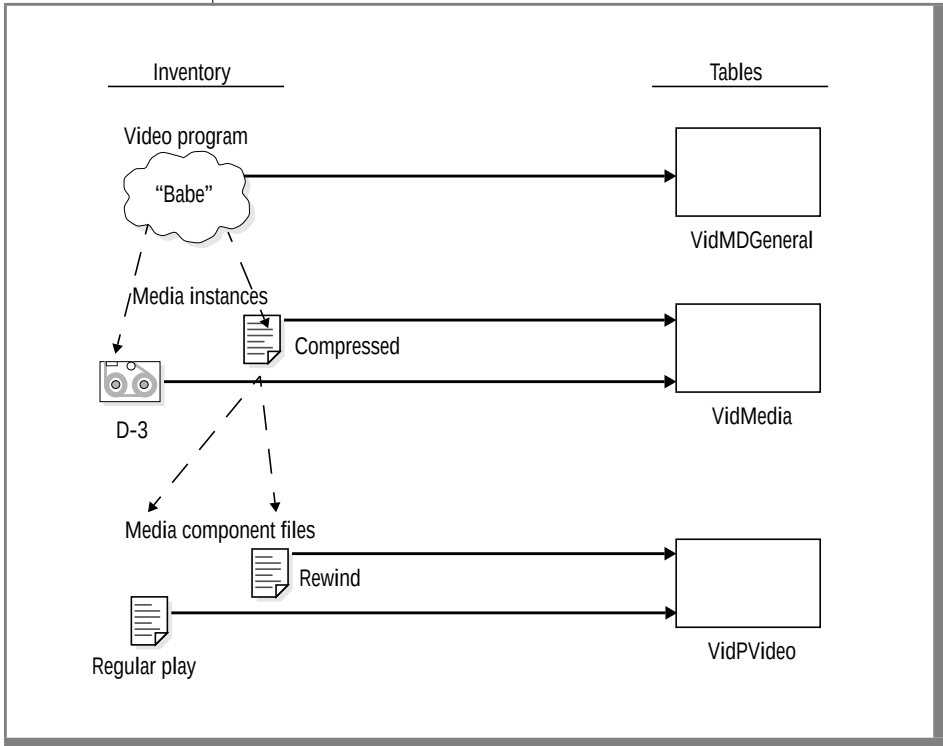


Figure 4-2
Example Use of
Inventory Tables



VidMDGeneral

The **VidMDGeneral** table is a registry for each video program stored in the database. Usually, there is one row per video program in this table no matter how many formats or copies of the video program exist, but this is not enforced.

Tip: You can “preload” this table with entries for video programs that have not yet been shot or that are still in the planning stages.

Column Name	Type	Sample Data	Explanation
VideoID	SERIAL NOT NULL	1	Primary Key: ID number assigned to this video program by Dynamic Server.
Title	CHAR (128)	'blue sky toys commercial'	Short title of this video program.
Proxy	LLD_LOCATOR	"row('IFX_BLOB', FILETOBLOB('C:\video\sky.img', 'client'), NULL)"	Image or icon acting as a proxy for this video program in its entirety. For information on the LLD_LOCATOR data type, see the LOB Locator DataBlade Module Programmer's Guide . For information on the FILETOBLOB function, see <i>Informix Guide to SQL: Syntax</i> .
Description	LVARCHAR	'Commercial for blue sky toys, produced by Lee Scott.'	Description of this video program.

VidMedia

The **VidMedia** table contains general physical attribute metadata for each media instance in the system.

For each video program in **VidMDGeneral**, insert one row per copy into the **VidMedia** table.



Important: The media instance from which the Informix Video Foundation DataBlade module is to create the **foundation timeline** is known as the **source timeline**. Insert this row into **VidMedia** with the **MediaDerived** column set to *NULL*: the source is usually an analog tape. For a review of these concepts, see [“Physical Attributes” on page 1-9](#).

Get the **VideoID** value from the **VidMDGeneral** table and the **StandardID** value from the **VidStandard** table (see [“VidStandard” on page 4-32](#)). As you insert additional media instances into **VidMedia**, indicate their relationship to the source timeline by inserting its **MediaID** value in the **MediaDerived** column.

There are two exceptions to the guidelines of “one row per copy,” as follows:

- If you know there will be one or more instances of a video program, but the data is not yet available (the program is still in the planning stages or has not yet been shot), you can “preload” this table, filling in the column values you know and updating the rest later.
- If you have several, partial copies of a video program (for example, three 7-minute segments of the movie *Babe* in MPEG format), insert a row in **VidMedia** with a different value in the **Offset** column for each piece.

The **Offset** column is also used if you have just one partial copy of a video program. For example, [Figure 4-3](#) shows the source timeline starting at 0 and a media instance that starts at MedPtr value '5S'. Thus the value of this media instance's **Offset** column would be '5S'.

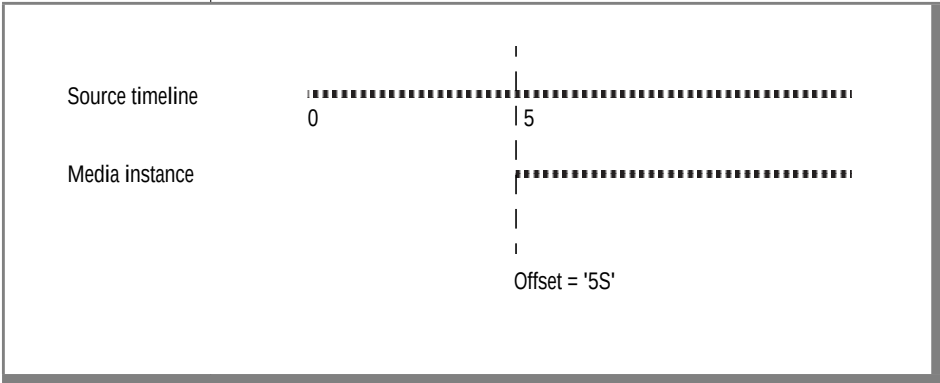


Figure 4-3
Offset from Source
Timeline



If a media instance in **VidMedia** is encoded, insert a row for each media component file into the appropriate table: **VidPVideo**, **VidPAudio**, or **VidPOther**.

Important: The **VidMedia** table serves another useful purpose: it provides the values of the source timeline (the media instance with a **MediaDerived** value of NULL) needed to calculate a MedPtr value for content-processing that occurs on the client computer. To make this calculation, you need to know the MedPtr value's frames-per-second rate, which you can derive from the **StandardID** column, and the **CaptureRate** value.

For example, a video program with a PAL source timeline (**StandardID** value of 7) has a frames-per-second rate of 25. If the **CaptureRate** column value is 10 frames per second, then for frame 256, the MedPtr value is 102.4 seconds, as follows:

$$256 * 10 / 25 = 102.4 \text{ seconds}$$

Column Name	Type	Sample Data	Explanation
MediaID	SERIAL NOT NULL	1001	Primary Key: ID number assigned to this media instance by Dynamic Server.

Column Name	Type	Sample Data	Explanation
VideoID	INTEGER NOT NULL	1	Foreign Key: Video ID number assigned to this media instance, from the VidMDGeneral table.
StandardID	INTEGER NOT NULL	7	Foreign Key: Video standard of this media instance, such as NTSC, SECAM, or PAL, from the VidStandard table.
FormatID	INTEGER NOT NULL	13	Foreign Key: Video format of this media instance, such as D-1, VHS, or Hi-8, from the VidFormat table.
CaptureRate	FLOAT	29.97	Frame rate, in frames-per-second, at which this media instance was captured.
MediaDerived	INTEGER	1000	MediaID of the video program from which this media instance was derived. Set to NULL if this media instance is the source timeline.
Duration	MedPtr	'326250M'	Length of this media instance. For information on the MedPtr data type, see “MedPtr Data Type” on page 2-3 .
Offset	MedPtr	MedPtrFrom ('01;02;12;05-D')	Offset from the beginning of the source timeline, indicating the beginning of this media instance.

(2 of 3)

Column Name	Type	Sample Data	Explanation
ACodecID	INTEGER	4	Foreign Key: Audio format of this media instance if it is on tape, from the VidACodec table. (For digital video, this information goes in the VidPAudio table; see “VidPAudio” on page 4-17.)
NumAChannels	SMALLINT	3	Number of audio channels used in this media instance.
When	DATETIME YEAR TO SECOND	'96-11-13 11:41:00'	Date and time this row was inserted into the VidMedia table.
Description	LVARCHAR	'MPEG-1 version of blue sky toys commercial'	Description of this media instance.

(3 of 3)

VidPVideo

Each encoded media instance is made up of one or more media component files. Each media component file is referenced by at least one row in the **VidPVideo** table, and there can be additional rows in the **VidPAudio** and **VidPOther** tables. These tables describe the physical characteristics and locations of the media component files.

The following steps describe how the media-specific inventory tables are typically used. The video compression format you use determines whether or not you need to use the **VidPAudio** and **VidPOther** tables.

1. Insert a row into the **VidPVideo** table for each video output media component file; there is usually just one. Get the **MediaID** value from the **VidMedia** table, and the **VCodecID** value from the **VidVCodec** table (see [“VidVCodec” on page 4-37](#)).

If you are using the **VidPVideo** table’s **Speed** column, Informix recommends that you enter a value based on the arbitrarily chosen 1000 for normal, forward playback. Values for rewind versions are negative, while values for a fast-forward speed are greater than 1000.

For example, a media instance in the **VidMedia** table can have three video media component files in **VidPVideo**:

- One for normal playback, **Speed** set to 1000
 - One for rewind at 15 times normal, **Speed** set to -15000
 - One for fast forward at 7 times normal, **Speed** set to 7000
2. If there is an audio portion to the media instance, even if it is contained in the video output file, insert at least one row into the **VidPAudio** table (for details, see [“VidPAudio” on page 4-17](#)).
 3. If there are any other media component files, insert a row into the **VidPOther** table for each file (for details, see [“VidPOther” on page 4-19](#)).

None of these tables has a primary key. This means that no uniqueness is enforced, allowing duplicate records in each table.

Column Name	Type	Sample Data	Explanation
MediaID	INTEGER NOT NULL	1001	Foreign Key: ID number for this video media component file, from the VidMedia table.
VCodecID	INTEGER NOT NULL	2	Foreign Key: Video compression format of this video media component file, such as MPEG-1-system or H.320, from the VidVCodec table.

(1 of 2)

Column Name	Type	Sample Data	Explanation
BitRate	INTEGER	1151600	Bit rate, in bits-per-second, of this video media component file.
FWidth	INTEGER	352	Frame width, in pixels, of this video media component file.
FHeight	INTEGER	240	Frame height, in pixels, of this video media component file.
Speed	INTEGER	1000	Playback speed and its direction. 1000 is normal forward, a negative number is reverse, and a number greater than 1000 is fast forward. At least one VidPVideo row per VidMedia instance should have Speed set to 1000.
Location	MEDLOC	'SAM#1:1357.mpg'	Location of this video media component file. For information on the MedLoc data type, see “MedLoc Data Type” on page 3-7 .
DataSize	DOUBLE PRECISION	5.017	Size, in megabytes, of this video media component file.
When	DATETIME YEAR TO SECOND	'96-11-21 11:04:00'	Date and time this row was inserted into the VidPVideo table.
Description	LVARCHAR	'Sample Company-based MPEG-1 system stream'	Description of this video media component file.

(2 of 2)

VidPAudio

If an encoded media instance includes sound, insert at least one row into the **VidPAudio** table, as follows:

- If the audio is included in the video media component file, insert a row into the **VidPAudio** table to describe the audio's characteristics. Set **Location** to NULL.
- If the audio is in one or more audio media component files (*separate from the video media component file*), insert rows into the **VidPAudio** table to describe the characteristics and locations of these files. Do not set **Location** to NULL. For example:
 - If multiple audio channels were used, and each channel is stored in a separate file, insert one row per channel.
 - If there are multiple sound track files available for this video program—for example, one in English and another in Spanish—insert one row per sound track file.

For each row in **VidPAudio**, get the **MediaID** value from the **VidMedia** table, and the **ACodecID** value from the **VidACodec** table (see [“VidACodec” on page 4-35](#)).

Column Name	Type	Sample Data	Explanation
MediaID	INTEGER NOT NULL	1001	Foreign Key: ID number of the media instance with which this audio file is associated, from the VidMedia table.
ACodecID	INTEGER NOT NULL	4	Foreign Key: Audio compression format of this audio media component file, such as MPEG-1-Audio, PCM, and Real-Audio, from the VidACodec table.
SampleRate	FLOAT	8000	Sample rate, in samples-per-second, of this audio media component file, such as 8 KHz, 44.1 KHz, and so on.

(1 of 2)

Column Name	Type	Sample Data	Explanation
BitsPerSample	INTEGER	8	Bits-per-sample, 8 or 16, of this audio media component file.
BitRate	INTEGER	64000	Bits-per-second of this audio media component file. The value in this column is the result of SampleRate * BitsPerSample
ChannelNum	SMALLINT	1	Audio channel number with which this audio media component file is associated.
Location	MEDLOC	'SAM#1:1357.au'	Location of this audio media component file. For information on the MedLoc data type, see “MedLoc Data Type” on page 3-7 .
DataSize	DOUBLE PRECISION	2.45	Size, in megabytes, of this audio media component file.
When	DATETIME YEAR TO SECOND	'96-11-13 13:41:00'	Date and time this row was inserted into the VidPAudio table.
Description	LVARCHAR	'Spanish version'	Description of this audio media component file.

(2 of 2)

VidPOther

If the encoding process produces any other media component files besides video and audio, insert a row for each of these files into the **VidPOther** table. For example, there might be a row in this table for the MPEG-2-Transport user stream, or for a stream of RTSP (real-time streaming protocol) commands.

For each row in **VidPOther**, get the **MediaID** value from the **VidMedia** table.

Column Name	Type	Sample Data	Explanation
MediaID	INTEGER NOT NULL	1001	Foreign Key: ID number of the media instance with which this media component file is associated, from the VidMedia table.
Location	MEDLOC	'SAN#1:1357.txt'	Location of this media component file. For information on the MedLoc data type, see “MedLoc Data Type” on page 3-7.
When	DATETIME YEAR TO SECOND	'96-11-14 02:53:00'	Date and time this row was inserted into the VidPOther table.
Description	LVARCHAR	'More output for the blue sky toys commercial.'	Description of this media component file.

A *proxy* is a description that “stands in” for a piece of data. Usually, a proxy is much smaller and is of lower resolution than the video data it represents.

Subject Matter Tables

The subject matter tables allow you to create and store relationships among pieces of a video program’s timeline and abstract descriptions (also known as *proxies*) of that program’s contents. These abstract descriptions can be in one or more of the following forms:

- Text annotations
- Graphic images
- Audio clips
- Video clips

As defined in [Chapter 1, “Overview of the Video Foundation DataBlade Module,”](#) the process of creating such relationships is known as *stratification*. When you create a relationship between a media chunk (stored by Dynamic Server as the MedChunk data type; see [“MedChunk Data Type” on page 2-12](#)) and a description, you have created a *stratum*. When you collect some of these relationships into a meaningful group, you have created a *strata*.

The following steps describe how the subject matter tables are typically used to create stratums and stratas:

1. Find a piece of the timeline—a media chunk—of interest, and “register” that chunk by inserting a row into the **VidChunkMD** table.

This table provides a link to the inventory tables by referencing the **VidMedia** table for the media instance that contains this chunk—thus, there is a link between a video program’s physical attributes and its abstract descriptions.

This link is particularly important for media management systems that are storing multiple video formats, and that require precision indexing and conversion among the formats. The *Native* elements of the MedPtr values that make up the MedChunk values allow precision indexing *in a particular format*. For example, a *Native* element containing a timecode must be “linked” to the Beta-SP media instance for that timecode to make sense, while a *Native* element containing a byte offset must be linked to the MPEG instance. After a query, the *Time* elements of the MedPtr values are in synch.

2. Create a relationship between that media chunk and one or more descriptions by making insertions into the appropriate abstract description tables, as follows:
 - For each audio clip proxy of the chunk, insert a row into the **VidAudDesc** table.
 - For image proxies, insert rows into the **VidImgDesc** table.
 - For text annotation proxies, insert rows into the **VidTextDesc** table.
 - For video clip proxies, insert rows into the **VidVidDesc** table.

Because **VidChunkMD** and the **VidMediaDesc** tables are likely to receive heavy, concurrent use (for example, by concurrent logging applications), they were created with row-level locking rather than page-level locking.
3. Each description is uniquely identified by the combination of the following three values:
 - The ID of the media chunk it describes
 - Its sequence number, if it is a member of a set of descriptions for the media chunk (for example, a still image from the beginning of the media chunk might be description #1, and an image from the end of the chunk might be description #3)
 - The ID of the strata in which it has been collected (see next)
4. Collect a meaningful group of these relationships into a strata by inserting a row into the **VidStrata** table, then updating the **VidMediaDesc** tables with the **StrataID** value.

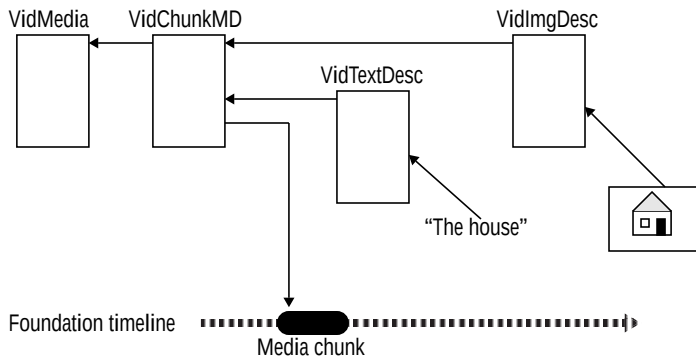
Figure 4-4 shows an example of this process:

1. A media chunk of interest was found, so a row was inserted into the **VidChunkMD** table.
2. A stratum was created between this chunk and the text annotation, “The house,” by inserting a row into the **VidTextDesc** table.
3. Another stratum was created between this chunk and the graphic image of the house by inserting a row into the **VidImgDesc** table.

Later, these relationships and others like them (perhaps of “the school,” “the store,” and “the office”) might be collected into a strata (such as a “buildings” strata) with an insertion into the **VidStrata** table.

The rest of this section describes each subject matter table in detail.

Figure 4-4
Example Use of
Subject Matter
Tables



VidChunkMD

The **VidChunkMD** table is a registry for each media chunk defined. A media chunk can be defined by a user, a video application, or a video DataBlade module, for example:

- A cataloging application might define media chunks based on *where* a scene takes place (the setting), *who* is in each scene, or *what* they are doing. In this case, the media chunks vary in size.
- A time- or size-constrained application might define media chunks based on a particular time unit, such as 20-second video clips.
- A library application might define media chunks based on patron permissions: a patron who has paid the fee has access to a single large chunk representing the entire video program, while a patron who hasn't paid has access only to smaller, "teaser" chunks.
- Another application might allow the user to define all of these types of chunks, and any other types of chunks that make sense to them.

No matter how it is defined, for each media chunk, insert one row into the **VidChunkMD** table. Get the **MediaID** value from the **VidMedia** table ([“VidMedia” on page 4-11](#)). This provides a link between a video program’s physical attributes and its abstract descriptions.

For each abstract description of this chunk, insert a row into the appropriate description table: **VidAudDesc**, **VidImgDesc**, **VidTextDesc**, or **VidVidDesc**.

Column Name	Type	Sample Data	Explanation
ChunkID	SERIAL8 NOT NULL	1001	Primary Key: ID number assigned to this media chunk by Dynamic Server.
MediaID	INTEGER NOT NULL	101	Foreign Key: ID number of the media instance from which this media chunk is derived, from the VidMedia table.
Chunk	MEDCHUNK	MedChunkFrom ('01:02:12:05' , '01:02:21:07')	Definition of this media chunk. For information on the MedChunk data type, see “MedChunk Data Type” on page 2-12 .

VidAudDesc

The **VidAudDesc** table stores abstract descriptions of media chunks in the form of audio clips. An audio clip proxy can be derived from the media chunk it describes (or from the media instance that contains the media chunk), or it can be an entirely unrelated clip.

For each audio clip proxy, insert one row into the **VidAudDesc** table. Get the **ChunkID** value from the **VidChunkMD** table and assign a **SeqNum** value to the proxy. If this audio clip proxy is derived from the media chunk it describes, or its media instance, use the **Source** column; otherwise, use the **AudProxyPeriod** column.

If this description becomes part of a strata, get the **StrataID** value from the **VidStrata** table.

Column Name	Type	Sample Data	Explanation
ChunkID	INT8 NOT NULL	101	Primary Key with StrataID and SeqNum, and Foreign Key: ID number of the media chunk for which this audio clip is a proxy, from the VidChunkMD table.
StrataID	INTEGER NOT NULL	5	Primary Key with ChunkID and SeqNum, and Foreign Key: ID number of the strata to which this audio clip proxy belongs, from the VidStrata table.
SeqNum	INTEGER NOT NULL	1	Primary Key with ChunkID and StrataID: position of this proxy in a set of audio descriptions of this media chunk.
AudProxyPeriod	MEDCHUNK	'0 2571'	Audio clip proxy, not derived from the media chunk or its media instance.

Column Name	Type	Sample Data	Explanation
Source	MedLoc	'SAM#1:1357.au'	Name of file from which this audio clip proxy was derived.
When	DATETIME YEAR TO SECOND	'96-10-22 02:53:00'	Date and time this row was inserted into the VidAudDesc table.
Description	LVARCHAR	'Comments from George on the blue sky toys commercial.'	Description of this audio clip proxy.

(2 of 2)

VidImgDesc

The **VidImgDesc** table stores abstract descriptions of media chunks in the form of graphic images. An image proxy can be derived from the media chunk it describes (or from the media instance that contains the media chunk), or it can be an entirely unrelated image.

Image proxies can be one or any combination of the following; their order (if one exists) can be noted in the **SeqNum** column:

- Thumbnail images: representative frames from the media chunk or its media instance, or stock photos
- Salient stills: overlays of representative frames from the media chunk or its media instance
- Icons: symbols created with a graphics application

For each image proxy, insert one row into the **VidImgDesc** table. Get the **ChunkID** value from the **VidChunkMD** table, and assign a **SeqNum** value to the proxy. If this image proxy is derived from the media chunk it describes, or its media instance, use the **ImgProxyPeriod** column; otherwise, use the **ImgProxy** column.

If this description becomes part of a strata, get the **StrataID** value from the **VidStrata** table.

Column Name	Type	Sample Data	Explanation
ChunkID	INT8 NOT NULL	101	Primary Key with StrataID and SeqNum, and Foreign Key: ID number of the media chunk for which this image is a proxy, from the VidChunkMD table.
StrataID	INTEGER NOT NULL	5	Primary Key with ChunkID and SeqNum, and Foreign Key: ID number of the strata to which this image proxy belongs, from the VidStrata table.

Column Name	Type	Sample Data	Explanation
SeqNum	INTEGER NOT NULL	1	Primary Key with ChunkID and StrataID: position of this proxy in a set of image descriptions of this media chunk.
ImgProxy	LLD_LOCATOR	"row('IFX_BLOB', FILETOBLOB('3579.img', 'client'), NULL)"	Image proxy, not derived from the media chunk or its media instance. For information on the LLD_LOCATOR data type, see the LOB Locator DataBlade Module Programmer's Guide . For information on the FILETOBLOB function, see <i>Informix Guide to SQL: Syntax</i> .
ImgProxyPeriod	MedChunk	'2357 4893'	Media chunk that contains this image proxy, derived from the media chunk or its media instance. For information on the MedChunk data type, see “MedChunk Data Type” on page 2-12 .
Source	MedLoc	'SAM#1:3579.img'	Name of file from which this image proxy was derived.
When	DATETIME YEAR TO SECOND	'96-09-02 02:03:00'	Date and time this row was inserted into the VidImgDesc table.
Description	LVARCHAR	'Picture icon of the blue sky toys commercial.'	Description of this image proxy.

(2 of 2)

VidTextDesc

The **VidTextDesc** table stores abstract descriptions of media chunks in the form of text annotations. Examples of such annotations include copyright and royalty information.

For each such description, insert one row into this table. Get the **ChunkID** value from the **VidChunkMD** table, and assign a **SeqNum** value to the proxy. If this description becomes part of a strata, get the **StrataID** value from the **VidStrata** table. Unlike the other description tables, which provide pointers to the proxies, this table contains the text annotation.

Column Name	Type	Sample Data	Explanation
ChunkID	INT8 NOT NULL	101	Primary Key with StrataID and SeqNum, and Foreign Key: ID number of the media chunk for which this annotation is a proxy, from the VidChunkMD table.
StrataID	INTEGER NOT NULL	5	Primary Key with ChunkID and SeqNum, and Foreign Key: ID number of the strata to which this annotation proxy belongs, from the VidStrata table.
SeqNum	INTEGER NOT NULL	1	Primary Key with ChunkID and StrataID: position of this proxy in a set of text annotations of this media chunk.
When	DATETIME YEAR TO SECOND	'96-12-24 12:30:00'	Date and time this row was inserted into the VidTextDesc table.

(1 of 2)

Column Name	Type	Sample Data	Explanation
Description	LVARCHAR	'This segment features a boy and his dog, and might be useful in Psych 105, Human-Animal Relationships.'	Text annotation proxy.

(2 of 2)

VidVidDesc

The **VidVidDesc** table stores abstract descriptions of media chunks in the form of video clips. A video clip proxy can be derived from the media chunk it describes (or from the media instance that contains the media chunk), or it can be an entirely unrelated clip.

For each video clip proxy, insert one row into the **VidVidDesc** table. Get the **ChunkID** value from the **VidChunkMD** table and assign a **SeqNum** value to the proxy. If this video clip proxy is derived from the media chunk it describes, or its media instance, use the **Source** column; otherwise, use the **VidProxyPeriod** column.

If this description becomes part of a strata, get the **StrataID** value from the **VidStrata** table.

Column Name	Type	Sample Data	Explanation
ChunkID	INT8 NOT NULL	101	Primary Key with StrataID and SeqNum, and Foreign Key: ID number of the media chunk for which this image is a proxy, from the VidChunkMD table.
StrataID	INTEGER NOT NULL	5	Primary Key with ChunkID and SeqNum, and Foreign Key: ID number of the strata to which this image proxy belongs, from the VidStrata table.
SeqNum	INTEGER NOT NULL	1	Primary Key with ChunkID and StrataID: position of this proxy in a set of video descriptions of this media chunk.
VidProxyPeriod	MedChunk	'000025 997456'	Video clip proxy, not derived from the media chunk or its media instance.

Column Name	Type	Sample Data	Explanation
Source	MedLoc	'SAM#1:2468.avi '	Name of file from which this video clip proxy was derived.
When	DATETIME YEAR TO SECOND	'96-09-02 02:03:00'	Date and time this row was inserted into the VidVidDesc table.
Description	LVARCHAR	'Video clip showing dancing toys.'	Description of this video clip proxy.

(2 of 2)

VidStrata

The **VidStrata** table is a registry for each strata that is created. Insert one row per strata in this table, then update the appropriate description tables (**VidAudDesc**, **VidImgDesc**, **VidTextDesc**, and **VidVidDesc**) with the **StrataID** value.

Column Name	Type	Sample Data	Explanation
StrataID	SERIAL NOT NULL	5	Primary Key: ID number assigned to this strata by Dynamic Server.
Name	CHAR (64)	'Prof. Hurlihy annotations'	Name of this strata.
StrataType	CHAR (18)	'PROF'	Strata type code or label.
Description	LVARCHAR	'Comments from Prof. Hurlihy after viewing the program, in preparation for next lesson.'	Description of this strata.

Lookup Tables

The Informix Video Foundation DataBlade module provides lookup tables in its schema for reference by its inventory tables. When the schema is installed, these tables are loaded with default data. Applications and DataBlade modules can insert rows into these tables with site-specific values and can reference the tables in their own database tables or functions.

VidStandard

The **VidStandard** table lists video standards.

Column Name	Type	Sample Data	Explanation
StandardID	SERIAL NOT NULL	2	Primary Key: ID number assigned to this video standard by Dynamic Server.
Name	CHAR (18)	'NTSC'	Name of this video standard.
FrameRate	FLOAT	29.97	Frames-per-second rate.
LinesPerFrame	INTEGER	525	Number of lines per frame before scanning returns to the top of the monitor.
VisibleLines	INTEGER	483	Number of lines per frame that are visible (<i>invisible</i> lines are used for vertical blanking, and so on).

The following shows the default data installed in the **VidStandard** table.

StandardID	Name	FrameRate	LinesPerFrame	VisibleLines
1	Unknown	NULL	NULL	NULL
2	NTSC	29.97	525	483
3	SECAM	25.0	625	576
4	D-SECAM	25.0	625	576
5	K-SECAM	25.0	625	576
6	L-SECAM	25.0	625	576
7	PAL	25.0	625	576
8	PAL-M	29.97	525	483
9	PAL-N	25.0	625	576
10	PAL-B	25.0	625	576
11	PAL-G	25.0	625	576
12	PAL-H	25.0	625	576
13	PAL-I	25.0	625	576

VidFormat

The **VidFormat** table lists video formats.

Column Name	Type	Sample Data	Explanation
FormatID	SERIAL NOT NULL	2	Primary Key: ID number assigned to this video format by Dynamic Server.
Name	CHAR (18)	'D-1 '	Name of this video format.

The following shows the default data installed in the **VidFormat** table.

FormatID	Name
1	Unknown
2	D-1
3	D-2
4	D-3
5	VHS
6	Hi-8
7	8mm
8	S-VHS
9	Film
10	BetaSP
11	BetaSP-30
12	BetaSP-60
13	Compressed

VidACodec

The **VidACodec** table lists audio compression formats.

Column Name	Type	Sample Data	Explanation
ACodecID	SERIAL NOT NULL	2	Primary Key: ID number assigned to this audio compression format by Dynamic Server.
Name	CHAR (18)	'MPEG-1-Audio'	Name of this audio compression format.

The following shows the default data installed in the **VidACodec** table.

ACodecID	Name
1	Unknown
2	MPEG-1-Audio
3	MPEG-2-Audio
4	ADPCM
5	IMA ADMCM
6	DVI ADPCM
7	GSM 6.10
8	G.721
9	IRCAM
10	PSION
11	MACE
12	uLaw
13	aLaw
14	AIFF
15	AIFC

(1 of 2)

ACodecID	Name
16	Wave
17	MIDI
18	MOD
19	CD-DA

(2 of 2)

VidVCodec

The **VidVCodec** table lists video compression formats.

Column Name	Type	Sample Data	Explanation
VCodecID	SERIAL NOT NULL	2	Primary Key: ID number assigned to this video compression format by Dynamic Server.
Name	CHAR (18)	'MPEG-1-SYSTEM'	Name of this video compression format.

The following shows the default data installed in the **VidVCodec** table.

VCodecID	Name
1	Unknown
2	MPEG-1-System
3	MPEG-1-Video
4	MPEG-2-Transport
5	MPEG-2-Program
6	MPEG-2-System
7	MPEG-2-Video
8	H.320
9	H.263
10	H.261

Server Functions

In This Chapter	5-3
After	5-5
AfterOrEqual	5-7
Before	5-9
BeforeOrEqual	5-11
ClientAccess	5-13
Compare	5-16
Contained	5-20
Contains	5-23
Copy.	5-26
EndP.	5-28
Equal	5-30
Fps	5-32
Frame	5-34
GreaterThan	5-36
GreaterThanOrEqual	5-38
Hour.	5-40
Intersection	5-42
LessThan	5-44
LessThanOrEqual	5-46
Location	5-48
MedChunkFrom.	5-50
MedPtrFrom	5-53
MedStorage	5-57
MedTCFrom	5-59
Meets	5-62
Minus	5-64

Minute	5-69
Native	5-71
New	5-73
NotContained.	5-75
NotContains	5-77
NotEqual	5-79
NotMeets	5-81
NotOverlap	5-84
Overlap	5-86
Plus	5-89
Second	5-92
Shutdown	5-94
Size	5-96
StartP	5-98
Startup	5-100
Status	5-102
Storage	5-104
StorageID	5-106
Time	5-108
Type (MedLoc)	5-110
Type (MedTC)	5-112
VidFrameNumberFrom	5-114

In This Chapter

This chapter is an alphabetical reference to the server functions provided by the Informix Video Foundation DataBlade module. You can execute these functions using an Informix SQL client interface, such as INFORMIX-ESQL/C, the DataBlade API, DB-Access, or SPL.

The following information is provided for each function:

- A brief description of the task it performs
- Its syntax
- A brief description of each of its input arguments
- A brief description of the value it returns
- A list of other server functions or client API functions (see [Chapter 6](#), “Client API”) that may be of interest to you

In addition, one or more examples is provided for each function. Most function examples for each data type were set up using the same script; these are included here for your reference. (Others do additional setup within the example itself.)

This is the setup script for the MedPtr data type functions:

```
create table medptrtest (id int, val MedPtr);
```

This is the setup script for the MedTC data type functions:

```
create table medtctest (vidid integer, tc medtc);  
insert into medtctest values (1000,'00:53:12:19-F');  
insert into medtctest values (1001,'01:10:00:00-D');
```

This is the setup script for the MedChunk data type functions:

```
create table medchunktest (id int, val MedChunk);
insert into medchunktest values (1, '00-00:00:00:00|10-00:00:00:10');
insert into medchunktest values (2, '20-00:00:00:20|40-00:00:01:10');
insert into medchunktest values (3, '50-00:00:01:20|60-00:00:02:00');
```

This is the setup script for the MedLoc and MedStorage data type functions:

```
delete from MedVSIRegister;
delete from MedVSI_File;

create table medloctest (name varchar(30), loc MedLoc);
insert into MedVSI_File values ('/tmp/',0);
```

After

Tests whether the first media chunk starts after the second media chunk ends.

Syntax

```
After(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: t if the first *StartP* element is after the second *EndP* element; otherwise, f.

The following illustration shows a true condition.



Example

This example references the **medchunktest** table (page 5-4). The statement selects each row from the table and tests whether that row’s MedChunk value starts after the MedChunk value of '10-00:00:00:10|20-00:00:00:20' ends.

```
select id, 'After',
       After(val, '10-00:00:00:10|20-00:00:00:20') from medchunktest;
```

This is the result; only the MedChunk value in row 3 starts after the input media chunk ends:

```
id (constant) (expression)
```

1	After	f
2	After	f
3	After	t

See Also

Server Functions:

[“AfterOrEqual” on page 5-7](#)

[“Before” on page 5-9](#)

[“BeforeOrEqual” on page 5-11](#)

Client API Functions:

N.A.

AfterOrEqual

Tests whether the first media chunk starts after or at the same time that the second media chunk ends.

Syntax

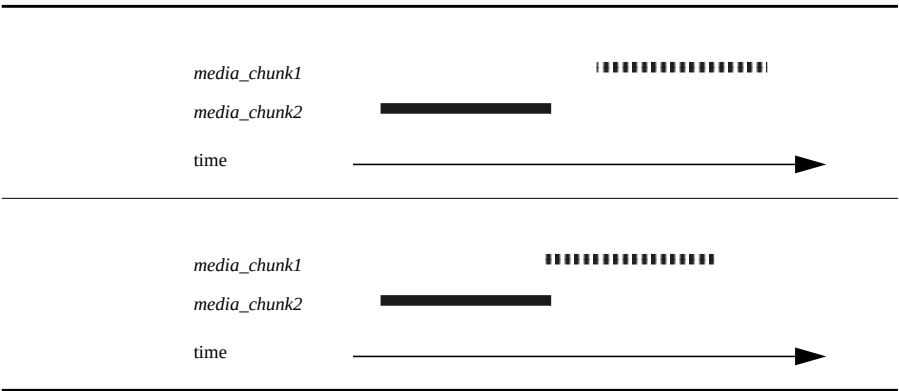
```
AfterOrEqual(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element is after or equal to the second *EndP* element; otherwise, *f*.

Each of the following illustrations shows a true condition.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value starts after or at the same time that the MedChunk value of '10-00:00:00:10|20-00:00:00:20' ends.

```
select id, 'AfterOrEqual',
       AfterOrEqual(val, '10-00:00:00:10|20-00:00:00:20') from medchunktest;
```

This is the result; the MedChunk values in rows 2 and 3 start after or at the same time that the input media chunk ends:

id (constant)		(expression)
1	AfterOrEqual	f
2	AfterOrEqual	t
3	AfterOrEqual	t

See Also

Server Functions:

[“After” on page 5-5](#)

[“Before” on page 5-9](#)

[“BeforeOrEqual” on page 5-11](#)

Client API Functions:

N.A.

Before

Tests whether the first media chunk ends before the second media chunk starts.

Syntax

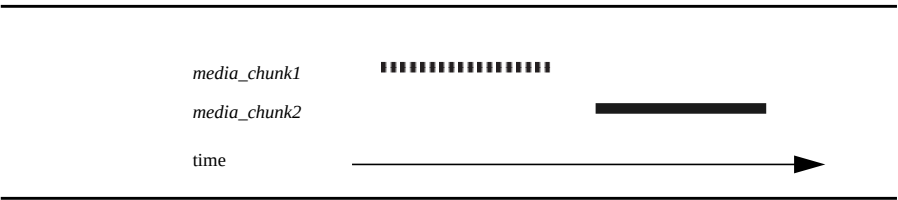
```
Before(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: `t` if the first *EndP* element is before the second *StartP* element; otherwise, `f`.

The following illustration shows a true condition.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value ends before the MedChunk value of '40-00:00:01:10|50-00:00:01:20' starts.

```
select id, 'Before',
       Before(val, '40-00:00:01:10|50-00:00:01:20') from medchunktest;
```

This is the result; only the MedChunk value in row 1 ends before the input media chunk starts:

id (constant) (expression)		
1	Before	t
2	Before	f
3	Before	f

See Also

Server Functions:

[“After” on page 5-5](#)

[“AfterOrEqual” on page 5-7](#)

[“BeforeOrEqual” on page 5-11](#)

Client API Functions:

N.A.

BeforeOrEqual

Tests whether the first media chunk ends before or at the same time the second media chunk starts.

Syntax

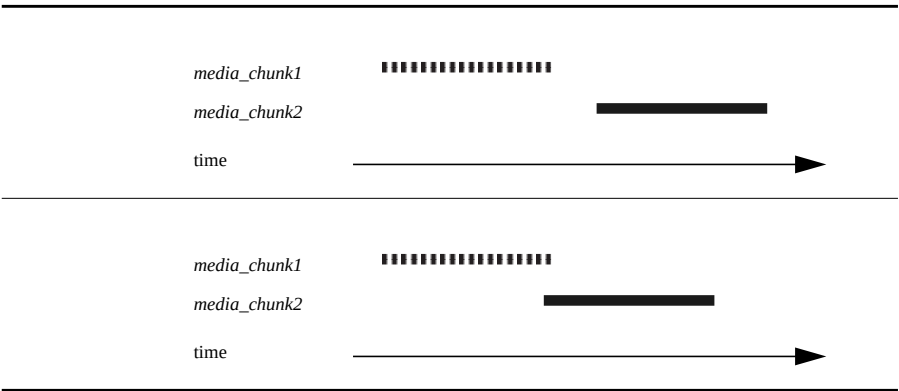
```
BeforeOrEqual(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
<i>media_chunk1</i> , <i>media_chunk2</i>	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *EndP* element is before or equal to the second *StartP* element; otherwise, *f*.

Each of the following illustrations shows a true condition.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value ends before or at the same time that the MedChunk value of '40-00:00:01:10|50-00:00:01:20' starts.

```
select id, 'BeforeOrEqual',
       BeforeOrEqual(val, '40-00:00:01:10|50-00:00:01:20') from medchunktest;
```

This is the result; the MedChunk values in rows 1 and 2 end before or at the same time that the input media chunk starts:

id	(constant)	(expression)
1	BeforeOrEqual	t
2	BeforeOrEqual	t
3	BeforeOrEqual	f

See Also

Server Functions:

[“After” on page 5-5](#)

[“AfterOrEqual” on page 5-7](#)

[“Before” on page 5-9](#)

Client API Functions:

N.A.

ClientAccess

Returns a handle to be used by the client application to access MedLoc data directly from a storage server. The client must request access operation permissions using the arguments to **ClientAccess**. The returned handle can encode the permissions granted and is not valid for other operations. For example, a handle returned for video playback is not valid for overwriting the video data.

Syntax

```
ClientAccess(media_object, services, reserved, 'arguments')  

```

Argument	Purpose	Restrictions
<i>media_object</i>	Specifies the large media object to which the caller requests access.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 .
<i>services</i>	Specifies the access services the caller is requesting.	Must be a positive number; see below for bit settings.
<i>reserved</i>	Reserved for future use.	Set to 0.
<i>arguments</i>	Specifies additional, storage server-specific arguments to the storage server for the large media object.	Specify the arguments in single quotes. If not used, set to the empty string ''. Refer to the documentation provided with the VSI implementation you are using for supported arguments.

The following table shows the decimal value and definition for each of the *services* argument’s bit settings. By using bitwise OR, multiple access permissions can be requested in one call.

Bit	Decimal	Service
0	1	Play
1	2	Seek
2	4	Rate control
3	8	Write from file
4	16	Write from livefeed
5	32	Delete file
6	64	Add callback
7	128	Delete callback
8–31	—	Reserved

Returns

Returns a MedLoc value (for information on this data type, see “[MedLoc Data Type](#)” on page 3-7). This handle can indicate success or failure of the request, or it can be encoded or encrypted with access- or service-control information about this large media object. Refer to the documentation provided with the storage server-specific VSI implementation you are using for details.

Similarly, access or security enforcement is storage server-specific. Some VSI implementations do not allow access unless you have first obtained the appropriate handle using the **ClientAccess** function; others are more lenient. Again, refer to the VSI implementation’s documentation for details.

Example

This example references the **medloctest** table ([page 5-4](#)) and the row that was inserted into the table in the **New** function's example ([“New” on page 5-73](#)). The statement requests access to write to the new MedLoc value, which is identified by the string 'Empty Media'.

```
select clientAccess(loc,8,0,'') from medloctest where  
name='Empty Media';
```

See Also

Server Functions:

N.A.

Client API Functions:

N.A.

Compare

Compares two values of the same data type and reports on their relationship.

Syntax

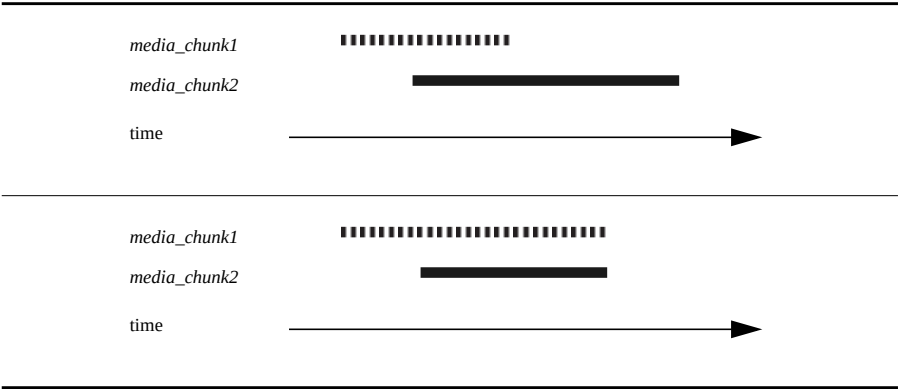
```
Compare(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

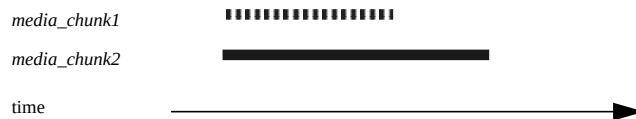
Returns an INTEGER value, as follows:

- 0 if *data_type1* = *data_type2*.
The two values reference the same media point, timecode, media object location, or media chunk.
- -1 if *data_type1* < *data_type2*.
For MedChunk values, this means one of the following conditions:
 - The first media chunk starts before the second media chunk starts. It doesn't matter where the first media chunk ends.





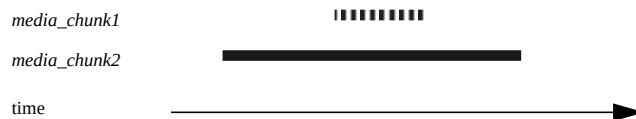
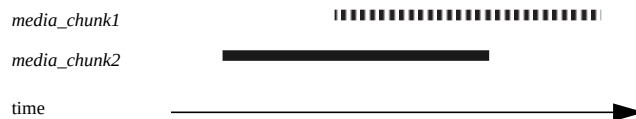
- The two media chunks start at the same point, but the first media chunk ends before the second ends.



- 1 if $data\_type1 > data\_type2$.

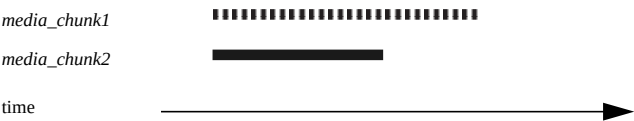
For MedChunk values, this means one of the following conditions:

- The first media chunk starts after the second media chunk starts. It doesn't matter where the first media chunk ends.





- ❑ The two media chunks start at the same point, but the first media chunk ends after the second ends.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and compares that row's MedChunk value to the MedChunk value '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'Compare',
       Compare(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result. Row 1's value is less than the input media chunk, row 2's value is equal to the input, and row 3's value is greater than the input.

id (constant) (expression)		
1	Compare	-1
2	Compare	0
3	Compare	1

See Also

Server Functions:

[“After” on page 5-5](#)

[“Before” on page 5-9](#)

[“Equal” on page 5-30](#)

[“GreaterThan” on page 5-36](#)

[“LessThan” on page 5-44](#)

Client API Functions:

N.A.

Contained

Tests whether the first media chunk is contained in the second media chunk.

Syntax

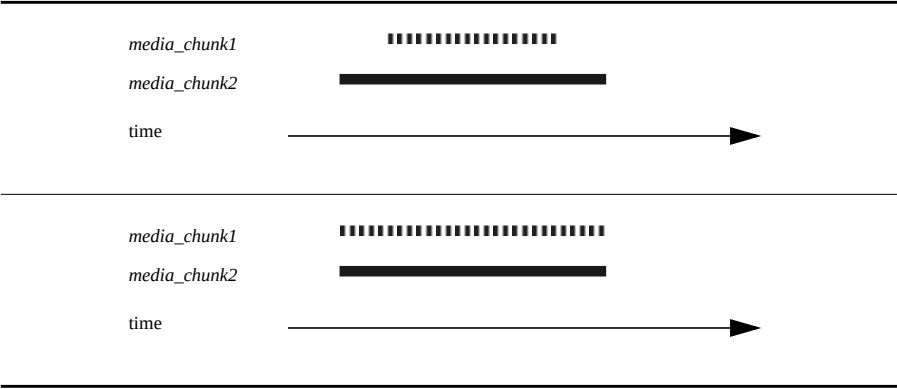
```
Contained(media_chunk1, media_chunk2)
```

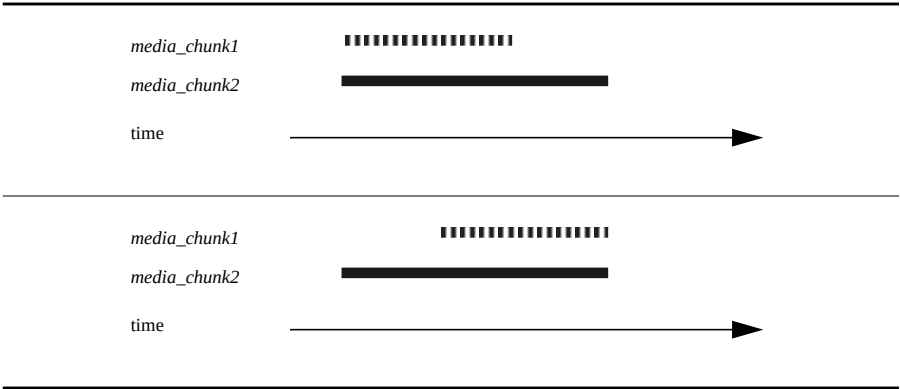
Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element is greater than or equal to the second, *and* if the first *EndP* element is less than or equal to the second. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value is contained in the MedChunk value of '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'Contained',
       Contained(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result; only the MedChunk value in Row 2 is contained in the input media chunk:

```
id (constant) (expression)
1 Contained      f
2 Contained      t
3 Contained      f
```

See Also

Server Functions:

[“Contains” on page 5-23](#)

[“NotContained” on page 5-75](#)

[“NotContains” on page 5-77](#)

Client API Functions:

N.A.

Contains

Tests whether the first media chunk contains the second media chunk.

Syntax

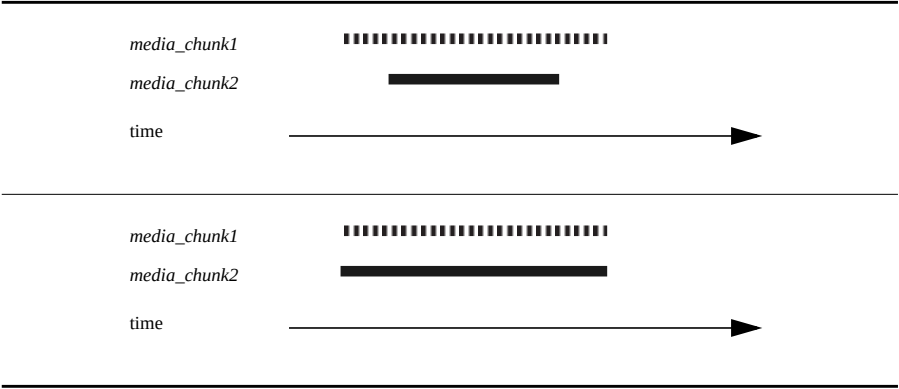
```
Contains(media_chunk1, media_chunk2)
```

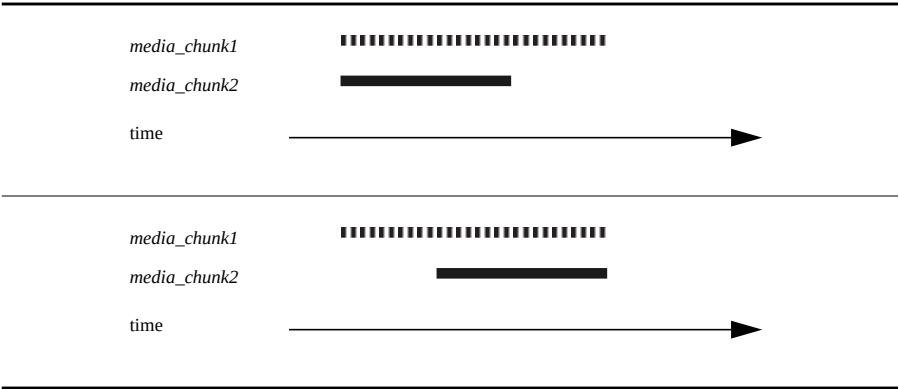
Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element is less than or equal to the second, *and* the first *EndP* is greater than or equal to the second. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value contains the MedChunk value of '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'Contains',  
       Contains(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result; only the MedChunk value in Row 2 contains the input media chunk:

id (constant) (expression)		
1	Contains	f
2	Contains	t
3	Contains	f

See Also

Server Functions:

[“Contained” on page 5-20](#)

[“NotContained” on page 5-75](#)

[“NotContains” on page 5-77](#)

Client API Functions:

N.A.



Copy

Returns a MedLoc value that tells Dynamic Server where to create a new large media object on the specified storage server and where to find the data to be copied into the new large media object. (The actual creating and copying operations are handled internally.)

Important: Because `VSI_CLIENT` values are usually meaningful only for a single client, Informix recommends that you not store them in the database. Thus, if the **Copy** function is called with a destination of `VSI_CLIENT`, the **Copy** function handles the copying operation itself (it is not handled internally). See [“VSI_CLIENT” on page 3-12](#) for more information.

Syntax

```
Copy(media_src, storage_dest)
```

Argument	Purpose	Restrictions
<code>media_src</code>	Specifies the large media object from which to copy the data.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 .
<code>storage_dest</code>	Specifies the storage server to receive the data.	Must be a MedStorage row value; see “MedStorage Data Type” on page 3-4 .

Returns

Returns a MedLoc value representing a new copy of the source data.

Example

This example references the **medloctest** table ([page 5-4](#)). The **Storage** function ([“Storage” on page 5-104](#)) returns the row from the **MedVSIRegister** table ([“MedVSIRegister Table” on page 3-6](#)) for the storage server of type VSI_FILE, #1. Then the **Copy** function returns a MedLoc value informing Dynamic Server where to create a new large media object on VSI_FILE#1 and where to find the source for copying (on the current client, in **/tmp/myvideo.mpg**). Because the **Copy** function was called from an INSERT statement, the creating and copying operations occur and the returned MedLoc value is inserted into the **medloctest** table as a row identified by the string 'My Video'.

```
insert into medloctest values
('My Video', copy('VSI_CLIENT:/tmp/myvideo.mpg',storage('VSI_FILE#1:')) );
```

See Also

Server Functions:

[“New” on page 5-73](#)

Client API Functions:

N.A.

EndP

Returns the *EndP* element (the ending media point) of a MedChunk value.

Syntax

```
EndP(media_chunk)
```

Argument	Purpose	Restrictions
<i>media_chunk</i>	Specifies the media chunk for which the caller wants the ending media point.	Must be a MedChunk value; “MedChunk Data Type” on page 2-12.

Returns

Returns a MedPtr value. For information on this data type, see [“MedPtr Data Type” on page 2-3.](#)

Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns the ending media point of that row’s MedChunk value.

```
select id, 'EndP', EndP(val) from medchunktest;
```

These are the results:

id	1
(constant)	EndP
(expression)	10000000N-00:00:00:10
id	2
(constant)	EndP
(expression)	40000000N-00:00:01:10
id	3
(constant)	EndP
(expression)	60000000N-00:00:02:00

See Also

Server Functions:

[“Size” on page 5-96](#)

[“StartP” on page 5-98](#)

Client API Functions:

[“medChunkGetEndPtr” on page 6-9](#)

Equal

Tests whether two values of the same data type are equal: that is, whether they reference the same media point, timecode, media object location, or media chunk.

Syntax

```
Equal(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* equals *data_type2*; otherwise, *f*.

The following illustration shows the true condition for the MedChunk data type.



Example

This example references the **medchunktest** table (page 5-4). The statement selects each row in the table and tests whether that row's MedChunk value is equal to the MedChunk value '20-AnyString|40-NotInterpreted'.

```
select id, 'Equal',  
       Equal(val, '20-AnyString|40-NotInterpreted') from medchunktest;
```

This is the result; only the MedChunk value in Row 2 is equal to the input media chunk:

```
id (constant) (expression)
```

1	Equal	f
2	Equal	t
3	Equal	f

See Also

Server Functions:

[“Compare” on page 5-16](#)

[“NotEqual” on page 5-79](#)

Client API Functions:

N.A.

Fps

Returns the frames-per-second rate of a MedTC value.

Syntax

```
Fps ( timecode )
```

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the frames-per-second rate.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns a FLOAT value identifying the frames-per-second rate (and thus the video standard and valid frame numbers) of this timecode, as follows.

Fps Value	Video Standard	Valid Frame Numbers
24	Film	00 through 23.
25	PAL	00 through 24.
29.97	Drop-frame NTSC	02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.
30	NTSC	00 through 29.

Example

These examples reference the **medtctest** table ([page 5-3](#)).

The following statement selects the row from the table where the MedTC value does not have a frames-per-second rate of 29.97:

```
select vidID,tc from medtctest where fps(tc) <> 29.97;
```

This is the result; the -F specifies that this is a MedTC value for film, with a frames-per-second rate of 24:

```
vidid  1000
tc      00:53:12:19-F
```

The following example uses both the **Frame** (“Frame” on page 5-34) and **Fps** functions to select the row with a MedTC value with a frame number of 0 and a frames-per-second rate of 29.97:

```
select vidID,tc from medtctest where frame(tc)=0 and fps(tc)=29.97;
```

This is the result; remember that drop-frame NTSC timecodes (frames-per-second rate of 29.97) have frames 0 and 1 only for minutes divisible by 10:

```
vidid  1001
tc      01:10:00:00-D
```

See Also

Server Functions:

[“Frame” on page 5-34](#)

[“Hour” on page 5-40](#)

[“Minute” on page 5-69](#)

[“Second” on page 5-92](#)

[“Type \(MedTC\)” on page 5-112](#)

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

Frame

Returns the *FF* element (the frame number) of a MedTC value.

Syntax

`Frame(timecode)`

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the <i>FF</i> element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an INTEGER value that specifies the frame number. (This number is determined by the video standard in use; see [“MedTC Data Type” on page 2-7](#).)

Example

This example references the **medtctest** table ([page 5-3](#)). The following statement selects the row from the table where the MedTC value has a frame number of 0 and a frames-per-second rate of 29.97 (The **Fps** function is described in [“Fps” on page 5-32](#)).

```
select vidID,tc from medtctest where frame(tc)=0 and fps(tc)=29.97;
```

Because drop-frame NTSC timecodes have frame 0 only for minutes divisible by 10, this is the result:

```
vidid  1001
tc      01:10:00:00-D
```

See Also

Server Functions:

[“Fps” on page 5-32](#)

[“Hour” on page 5-40](#)

[“Minute” on page 5-69](#)

[“Second” on page 5-92](#)

[“Type \(MedTC\)” on page 5-112](#)

Client API Functions:

[“medTCGetFrame” on page 6-52](#)



GreaterThan

Tests whether the first data type value is greater than the second.

Tip: You cannot use this function with the *MedChunk* data type. Instead, use the *After* function, which is described in [“After” on page 5-5](#).

Syntax

```
GreaterThan(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* is greater than *data_type2*; otherwise, *f*.

Example

This example references the **medtctest** table ([page 5-3](#)). The statement selects a row from the table and tests whether the input value of '00.34.12-D' is greater than the selected row's MedTC value.

```
select vidid, 'GreaterThan' , '00.34.12.29-D' >
tc from medtctest
where type(tc) = 'D';
```

This is the result; the input value is not greater than the row's MedTC value:

```
vidid (constant) (expression)
1001 GreaterThan f
```

See Also

Server Functions:

[“After” on page 5-5](#)

[“GreaterThanOrEqual” on page 5-38](#)

[“LessThan” on page 5-44](#)

Client API Functions:

N.A.



GreaterThanOrEqualTo

Tests whether the first data type value is greater than or equal to the second.

Tip: You cannot use this function with the *MedChunk* data type. Instead, use the *AfterOrEqual* function, which is described on [“AfterOrEqual” on page 5-7](#).

Syntax

```
GreaterThanOrEqualTo(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* is greater than or equal to *data_type2*; otherwise, *f*.

Example

This example references the **medtctest** table ([page 5-3](#)). The statement selects a row from the table and tests whether the input value of '00:10:00:00-D' is greater than or equal to the selected row's MedTC value.

```
select vidid, 'GreaterThanOrEqualTo', '01:10:00:00-D' >= tc
from medtctest
where type(tc) = 'D';
```

This is the result; the input value is greater than or equal to the row's MedTC value:

```
vidid (constant)      (expression)
1001 GreaterThanOrEqualTo t
```

See Also

Server Functions:

[“AfterOrEqual” on page 5-7](#)

[“GreaterThanOrEqualTo” on page 5-36](#)

[“LessThanOrEqualTo” on page 5-46](#)

Client API Functions:

N.A.

Hour

Returns the *HH* element (the hours) of a MedTC value.

Syntax

```
Hour( timecode )
```

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the <i>HH</i> element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an INTEGER value that specifies the hours of the timecode.

Example

This example references the **medtctest** table ([page 5-3](#)). First, the example finds the row in the table where the MedTC value’s frames-per-second rate is 24. The MedTC value in that row is '00:53:12:19-F', which is added to '01:34:25:16-F', resulting in the MedTC value '02:27:38:11-F'. The SELECT statement thus returns 2, which is the *HH* element of that resulting MedTC value.

```
select hour( tc + '01:34:25:16-F' ) from medtctest where fps(tc)=24;
```


See Also

Server Functions:

[“Fps” on page 5-32](#)

[“Frame” on page 5-34](#)

[“Minute” on page 5-69](#)

[“Second” on page 5-92](#)

[“Type \(MedTC\)” on page 5-112](#)

Client API Functions:

[“medTCGetHour” on page 6-54](#)

Intersection

Returns a media chunk representing the points in common between two other media chunks.

Syntax

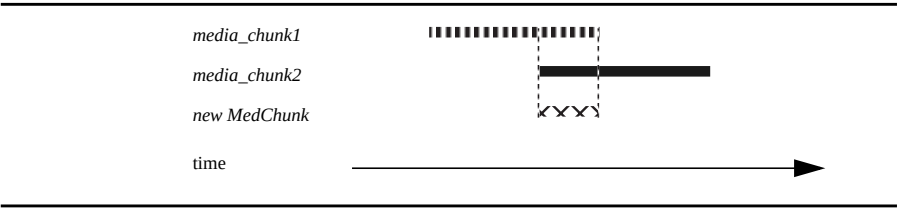
```
Intersection(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a MedChunk value, which is null if there are no points in common.

The following illustration shows a new MedChunk value formed from the intersection of two existing MedChunk values.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns a MedChunk value that consists of the points in common between that row's MedChunk value and the MedChunk value '15-00:00:00:15|25-00:00:00:25'.

```
select id, 'InterSection',
       InterSection(val, '15-00:00:00:15|25-00:00:00:25') from medchunktest;
```

This is the result; the MedChunk values in Rows 1 and 3 had no points in common with the input media chunk:

id	1
(constant)	InterSection
(expression)	
id	2
(constant)	InterSection
(expression)	20000000N-00:00:00:20 25000000N-00:00:00:25
id	3
(constant)	InterSection
(expression)	

See Also

Server Functions:

[“Meets” on page 5-62](#)

[“Overlap” on page 5-86](#)

Client API Functions:

N.A.



LessThan

Tests whether the first data type value is less than the second.

Tip: You cannot use this function with the *MedChunk* data type. Instead, use the *Before* function, which is described in “*Before*” on page 5-9.

Syntax

```
LessThan(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1, data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* is less than *data_type2*; otherwise, *f*.

Example

This example references the **medtctest** table (page 5-3). The statement selects a row from the table and tests whether the input value of '00.34.12-D' is less than the selected row's MedTC value.

```
select vidid, 'LessThan' , '00.34.12.29-D' < tc
from medtctest
where type(tc) = 'D';
```

This is the result; the input value is less than the row's MedTC value:

```
vidid (constant) (expression)
1001 LessThan t
```

See Also

Server Functions:

[“Before” on page 5-9](#)

[“GreaterThan” on page 5-36](#)

[“LessThanOrEqual” on page 5-46](#)

Client API Functions:

N.A.



LessThanOrEqual

Tests whether the first data type value is less than or equal to the second.

Tip: You cannot use this function with the *MedChunk* data type. Instead, use the *BeforeOrEqual* function, which is described in [“BeforeOrEqual” on page 5-11](#).

Syntax

```
LessThanOrEqual(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* is less than or equal to *data_type2*; otherwise, *f*.

Example

This example references the **medtctest** table ([page 5-3](#)). The statement selects a row from the table and tests whether the input value of '00.34.12-D' is less than or equal to the selected row's MedTC value.

```
select vidid, 'LessThanOrEqual' , '00.34.12.29-D' <= tc
from medtctest
where type(tc) = 'D';
```

This is the result; the input value is less than or equal to the row's MedTC value:

```
vidid (constant)      (expression)
1001 LessThanOrEqual    t
```

See Also

Server Functions:

[“BeforeOrEqual” on page 5-11](#)

[“GreaterThanOrEqual” on page 5-38](#)

[“LessThan” on page 5-44](#)

Client API Functions:

N.A.

Location

Returns the *Location* element of a MedLoc value.

Syntax

```
Location(media_object)
```

Argument	Purpose	Restrictions
media_object	Specifies the large media object for which the caller wants the <i>Location</i> element.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 .

Returns

Returns a LVARCHAR value identifying a large media object’s location on a particular storage server. The syntax of this string is storage-server-specific.

Example

This example references the **medloctest** table ([page 5-4](#)). The statement inserts a row into the table and then selects that row.

```
insert into medloctest values ('Nightly News','VSI_OTHER:VHS Cassette #1234,
10/24/96');
select location(loc) from medloctest where name='Nightly News';
```

This is the result:

```
(expression)  VHS Cassette #1234, 10/24/96
```

See Also

Server Functions:

[“Storage” on page 5-104](#)

[“StorageID” on page 5-106](#)

[“Type \(MedLoc\)” on page 5-110](#)

Client API Functions:

[“medLocGetLocation” on page 6-26](#)

MedChunkFrom

Converts one or more input arguments of various types into a MedChunk value. For information on the MedChunk data type, see “[MedChunk Data Type](#)” on page 2-12.

Syntax

To convert a single MedPtr value into a single-point MedChunk value (the *StartP* and *EndP* values are the same), use:

```
MedChunkFrom(media_point1)
```

To convert two MedPtr values (a starting media point and an ending media point) into a MedChunk value, use:

```
MedChunkFrom(media_point1, media_point2)
```

To convert two *Timecode* LVARCHAR values into a MedChunk value, use:

```
MedChunkFrom('time1', 'time2')
```

Tip: This function accepts inputs of type LVARCHAR rather than of type MedTC to save you from having to first convert the strings into MedTC values.



Argument	Purpose	Restrictions
media_point1, media_point2	Specifies one or two media points to convert to a media chunk. If one is given, a single-point MedChunk is constructed—that is, <i>StartP</i> and <i>EndP</i> have the same value. If two are given, the first is used as <i>StartP</i> and the second as <i>EndP</i>.	Both values must be MedPtr values; see “MedPtr Data Type” on page 2-3.

Argument	Purpose	Restrictions
<i>time1, time2</i>	Specifies two timecodes (variable-length character strings) to convert to a media chunk. The inputs are first converted into the <i>Time</i> elements of <i>StartP</i> and <i>EndP</i> , and then <i>the same inputs</i> are used as the <i>Native</i> elements.	Specify the timecodes in single quotes.

Returns

Returns a MedChunk value.

Examples

This section presents examples of using the **MedChunkFrom** function to convert MedPtr values into MedChunk values.

Input of a Single MedPtr Value

This example references the **medchunktest2** table created here. The statement converts a single MedPtr value into a single-point MedChunk value; then it inserts that value into the table and selects it.

```
create table medchunktest2 (id int, val MedChunk);

insert into medchunktest2 values (1,
    MedChunkFrom('20-00:00:00:20'::MedPtr));
select id, 'MedChunkFrom', val from medchunktest2;
```

This is the result; the starting MedPtr value is the same as the ending MedPtr value:

```
id          1
(constant)  MedChunkFrom
val         20000000N-00:00:00:20|20000000N-00:00:00:20
```

Input of Two MedPtr Values

This example references the **medchunktest3** table created here. The statement converts two MedPtr values into a MedChunk value; then it inserts that value into the table and selects it.

```
create table medchunktest3 (id int, val MedChunk);

insert into medchunktest3 values (1,
    MedChunkFrom('20-00:00:00:20'::MedPtr, '40-00:00:01:10'::MedPtr));
select id, 'MedChunkFrom', val from medchunktest3;
```

This is the result:

```
id          1
(constant)  MedChunkFrom
val         20000000N-00:00:00:20|40000000N-00:00:01:10
```

See Also

Server Functions:

[“MedPtrFrom” on page 5-53](#)

[“MedTCFrom” on page 5-59](#)

[“VidFrameNumberFrom” on page 5-114](#)

Client API Functions:

[“medPtrToMedChunk” on page 6-43](#)

[“medPtrsToMedChunk” on page 6-47](#)

MedPtrFrom

Converts one or more input arguments of various types into a MedPtr value. For information on the MedPtr data type, see [“MedPtr Data Type” on page 2-3](#).

Syntax

To convert a *Time* INTEGER value and a *Native* LVARCHAR value into a MedPtr value, use:

```
MedPtrFrom(time, 'native')
```

To convert a frame number INTEGER value and a frames-per-second FLOAT value into a MedPtr value, use:

```
MedPtrFrom(frame_num, fps)
```

To convert a MedTC value into a MedPtr value, use:

```
MedPtrFrom(timecode)
```

Argument	Purpose	Restrictions
<i>time</i>	Specifies the value to use as the <i>Time</i> element.	Must be a positive number.
<i>native</i>	Specifies the variable-length character string to use as the <i>Native</i> element.	Specify the <i>Native</i> element in single quotes.
<i>frame_num</i>	Specifies the frame number used (with <i>fps</i> , next row) in calculating the <i>Time</i> element. This same input is used for the <i>Native</i> element.	Must be a positive number.

(1 of 2)

Argument	Purpose	Restrictions
<i>fps</i>	Specifies the frames-per-second rate used (with <i>frame_num</i> , previous row) in calculating the <i>Time</i> element.	Must be a positive number.
<i>timecode</i>	Specifies the timecode to convert. The single input is first converted into the <i>Time</i> element, and then the <i>same input</i> is used for the <i>Native</i> element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

(2 of 2)

Returns

Returns a MedPtr value.

Example

This section presents examples of using the **MedPtrFrom** function to convert input values of various data types into MedPtr values.

Input of Time, Native Values

This example references the **medptrtest** table ([page 5-3](#)). The **MedPtrFrom** function converts a *Time* element (INT8) and a *Native* element (LVARCHAR) into a MedPtr value. The statement inserts that value into the table and then selects it.

```
insert into medptrtest values (1,  
    MedPtrFrom(40000000000::int8, '00:00:01:10'));  
select id, 'MedPtrFrom', val from medptrtest;
```

This is the result:

```
id          1
(constant)  MedPtrFrom
val         40000000000N-00:00:01:10
```

Input of Frame Number, Fps Values

This example references the **medptrtest** table ([page 5-3](#)). The **MedPtrFrom** function converts a *Frame* number (INTEGER) and an *Fps* value (FLOAT) into a MedPtr value. The statement inserts that value into the table and then selects it.

```
insert into medptrtest values (1, MedPtrFrom(40, 29.97));
select id, 'MedPtrFrom', val from medptrtest;
```

This is the result:

```
id          1
(constant)  MedPtrFrom
val         1334668001N-40
```

Input of MedTC Value

This example references the **medtctest** table ([page 5-3](#)) and the **medtctest2** table (created here). The **MedPtrFrom** function is first used to insert a MedTC value into a column of type MedPtr. It is then used to convert the entries in the **medtctest** table into MedPtr values; these values are inserted into **medtctest2**. Finally, the entries of the **medtctest2** table are selected.

```
create table medtctest2 (ptr medptr);
insert into medtctest2 values (medptrfrom('00:00:10:02-P'::MedTC));
insert into medtctest2 select medptrfrom(tc) from medtctest;
select ptr from medtctest2;
```

This is the result. The MedTC values were first converted into the *Time* elements of the MedPtr value and then those same MedTC values were used as the *Native* elements.

```
ptr 100800000000N-00:00:10:02-P  
ptr 3192791666667N-00:53:12:19-F  
ptr 4200000000000N-01:10:00:00-D
```

See Also

Server Functions:

[“MedChunkFrom” on page 5-50](#)

[“MedTCFrom” on page 5-59](#)

[“VidFrameNumberFrom” on page 5-114](#)

Client API Functions:

[“medFrameNumberToMedPtr” on page 6-22](#)

[“medTCToMedPtr” on page 6-68](#)

[“medTimeNativeToMedPtr” on page 6-70](#)

MedStorage

Constructs a MedStorage row value, usually as a row for the **MedVSISRegister** table. For information on the MedStorage data type and its relationship to the **MedVSISRegister** table, see [“MedStorage Data Type” on page 3-4](#).

Syntax

```
MedStorage('type', storage_id, 'conn_name', 'user_id',  
           'password', 'conn_string', 'vendor', 'model')
```

Argument	Purpose	Restrictions
<i>type</i>	Specifies the type of storage server.	Specify in single quotes.
<i>storage_id</i>	Specifies the identification number of this storage server.	A positive number.
<i>conn_name</i>	Specifies this storage server’s identifier, in storage-server-specific syntax; for example, a network port address.	Specify in single quotes.
<i>user_id</i>	Specifies the login name for Dynamic Server access to storage server.	Specify in single quotes.
<i>password</i>	Specifies the password associated with <i>user_id</i> for this storage server.	Specify in single quotes.
<i>conn_string</i>	Specifies additional information needed by VSI UDRs to connect to the storage server.	Specify in single quotes.
<i>vendor</i>	Specifies the name of the vendor that manufactured this storage server.	Specify in single quotes.

Argument	Purpose	Restrictions
<i>model</i>	Specifies additional manufacturer information for this storage server; can consist of name, number, version number, and so on.	Specify in single quotes.

Returns

Returns a MedStorage row value.

Example

This example references the Informix Video Foundation DataBlade module's **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)). The statement inserts a row into the table for a storage server of type VSI_FILE, #1.

```
insert into MedVSIRegister values
(MedStorage('VSI_FILE',1,NULL,NULL,NULL,'/tmp/',NULL,NULL));
```

This example references the same table, inserting a row for a storage server of type SAM, #1.

```
insert into MedVSIRegister values(
    MedStorage('SAM', 1, 'SAM-4000-1', 'root', 'root1234',
        'host4:port3', 'Sample Company', '4000-PR0, 3.1'));
```

See Also

Server Functions:

[“Storage” on page 5-104](#)

Client API Functions:

N.A.

MedTCFrom

Converts two input arguments of various temporal data types into a MedTC value. For information on the MedTC data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

To convert a MedPtr and frames-per-second FLOAT value into a MedTC value, use:

```
MedTCFrom(media_point, fps)
```

To convert a frame number INTEGER value and a frames-per-second FLOAT value into a MedTC value, use:

```
MedTCFrom(frame_num, fps)
```

Argument	Purpose	Restrictions
<i>media_point</i>	Specifies the media point used (with <i>fps</i> , below) in calculating the media timecode. The <i>Native</i> element of the MedPtr is not interpreted.	Must be a MedPtr value; see “MedPtr Data Type” on page 2-3 .
<i>frame_num</i>	Specifies the frame number used (with <i>fps</i> , next row) in calculating the media timecode.	Must be a positive number.
<i>fps</i>	Specifies the frames-per-second rate used (with <i>media_point</i> , or <i>frame_num</i> , above) in calculating the media timecode.	Must be a valid SMPTE frame rate: 24.0, 25.0, 29.97, 30.0.

Returns

Returns a MedTC value.

Example

This example references the **medtctest4** table, created and populated here. The first SELECT statement selects all the rows in the table. The next SELECT statement converts a MedPtr value and frames-per-second rate into a MedTC value, then converts a frame number value and frames-per-second rate into a MedTC value, and finally subtracts the second MedTC value from the first.

```
create table medtctest4 ( frameNum integer,ns medptr);
insert into medtctest4 values
(vidframenumberfrom('00:32:12:09-f'), medptrfrom('00:32:12:09-f'::MedTC));

select * from medtctest4;

select medtcfrom(ns,24.0) - medtcfrom(frameNum::integer,24.0)
from medtctest4;
```

This is the result, showing the output of the two SELECT statements:

```
framenum    46377
ns           1932375000000N-00:32:12:09-F

(expression) 00:00:00:00-F
```

See Also

Server Functions:

[“MedChunkFrom” on page 5-50](#)

[“MedPtrFrom” on page 5-53](#)

[“VidFrameNumberFrom” on page 5-114](#)

Client API Functions:

[“medFrameNumberToMedTC” on page 6-24](#)

[“medPtrToMedTC” on page 6-45](#)

Meets

Tests whether the first media chunk meets the second media chunk, either at its start or its end.

Syntax

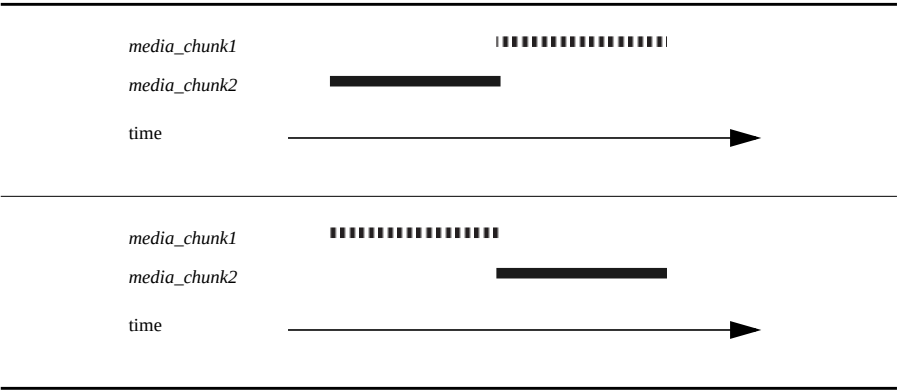
```
Meets(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element equals the second *EndP* element, or if the first *EndP* element equals the second *StartP* element. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value meets the MedChunk value of '10-00:00:00:10|40-00:00:01:10', either at its start or end.

```
select id, 'Meets',
       Meets(val, '10-00:00:00:10|40-00:00:01:10') from medchunktest;
```

This is the result; only the MedChunk value in Row 1 meets the input media chunk:

id (constant) (expression)		
1	Meets	t
2	Meets	f
3	Meets	f

See Also

Server Functions:

[“Intersection” on page 5-42](#)

[“NotMeets” on page 5-81](#)

[“Overlap” on page 5-86](#)

Client API Functions:

N.A.



Minus

Takes two values of the same data type and subtracts the second one from the first.

Tip: *You cannot use this function with the MedLoc data type.*

Syntax

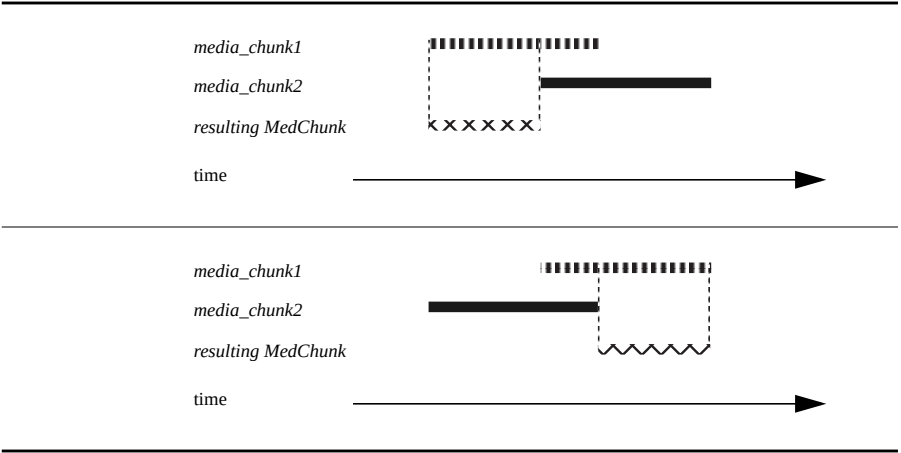
```
Minus(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies which data type value is subtracted (<i>data_type2</i>) from which (<i>data_type1</i>).	Both values must be of the same data type.

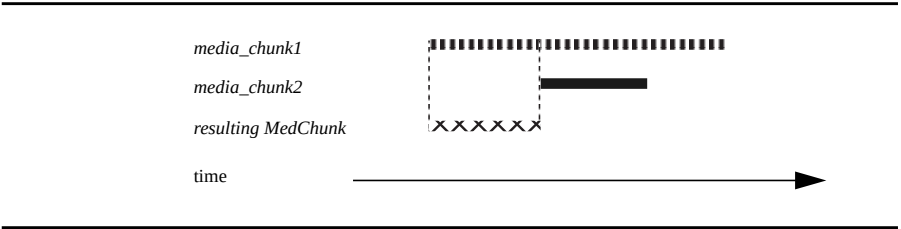
Returns

The returned result is of the same data type as the inputs.

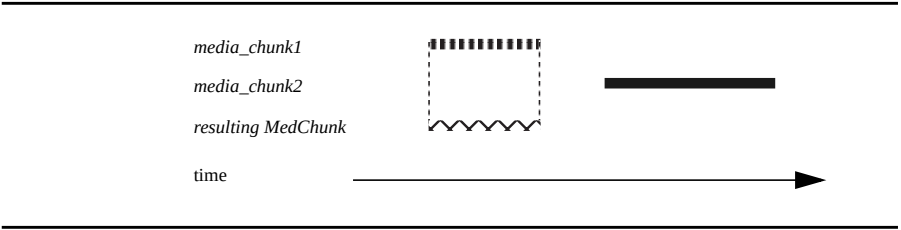
For the MedChunk data type (see “[MedChunk Data Type](#)” on page 2-12), the result is made up of those points of *media_chunk1* not shared with *media_chunk2*.



If there are two pieces of *media_chunk1* not shared with *media_chunk2*, only the first piece is returned.



If no points are shared, *media_chunk1* is returned.



Example

This section presents an example of using the **Minus** function to subtract MedPtr values and MedChunk values.

MedPtr Data Type

This example references the **medptrtest** table ([page 5-3](#)). The statement selects the row and returns a MedPtr value that is the result of subtracting the MedPtr value '40-00:00:01:10' from the row's MedPtr value.

```
insert into medptrtest values (1, '40-00:00:01:10');
select id, 'Minus', val - '10-00:00:00:10' from medptrtest;
```

This is the result:

id	1
(constant)	Minus
(expression)	30000000N

MedChunk Data Type

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns a MedChunk value that is the result of subtracting the MedChunk value '25-00:00:00:25|30-00:00:01:00' from the row's MedChunk value.

```
select id, 'Minus',
       val - '25-00:00:00:25|30-00:00:01:00' from medchunktest;
```

These are the results:

id	1
(constant)	Minus
(expression)	0N-00:00:00:00 10000000N-00:00:00:10
id	2
(constant)	Minus
(expression)	20000000N-00:00:00:20 25000000N-00:00:00:25
id	3
(constant)	Minus
(expression)	50000000N-00:00:01:20 60000000N-00:00:02:00

See Also

Server Functions:

[“Plus” on page 5-89](#)

Client API Functions:

[“medChunkMinus” on page 6-15](#)

[“medPtrMinus” on page 6-36](#)

[“medTCMinus” on page 6-62](#)

Minute

Returns the *MM* element (the minutes) of a MedTC value.

Syntax

```
Minute(timecode)
```

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the <i>MM</i> element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an INTEGER value that specifies the minutes of the timecode.

Example

This example references the **medtctest** table ([page 5-3](#)). The example selects the row from the table where the *MM* element of the MedTC value, when divided by 30.0, results in a value greater than 1.0.

```
select vidID,tc from medtctest where minute(tc) / 30.0 > 1.0;
```

This is the result (dividing the *MM* element of 53 by 30.0 results in 1.766):

```
vidid    1000
tc       00:53:12:19-F
```

See Also

Server Functions:

[“Fps” on page 5-32](#)

[“Frame” on page 5-34](#)

[“Hour” on page 5-40](#)

[“Second” on page 5-92](#)

[“Type \(MedTC\)” on page 5-112](#)

Client API Functions:

[“medTCGetMinute” on page 6-56](#)

Native

Returns the *Native* element of a MedPtr value.

Syntax

```
Native(media_point)
```

Argument	Purpose	Restrictions
media_point	Specifies the media point for which the caller wants the <i>Native</i> element.	Must be a MedPtr value; see “MedPtr Data Type” on page 2-3 .

Returns

Returns an LVARCHAR value that specifies the *Native* element.

Example

This example references the **medptrtest** table ([page 5-3](#)). The statement selects the row and returns the *Native* element of its MedPtr value.

```
insert into medptrtest values (1, '40-00:00:01:10');
select id, 'Native', Native(val) from medptrtest;
```

This is the result:

```
id          1
(constant)  Native
(expression) 00:00:01:10
```

See Also

Server Functions:

[“Time” on page 5-108](#)

Client API Functions:

[“medPtrGetNative” on page 6-32](#)

New

Returns a MedLoc value that tells Dynamic Server where to create a new, empty, large media object on the specified storage server. (The actual creating operation is handled internally.)

Syntax

`New(storage_server)`

Argument	Purpose	Restrictions
<i>storage_server</i>	Specifies the storage server on which to create an empty large media object.	Must be a MedStorage row value; see “MedStorage Data Type” on page 3-4 .

Returns

Returns a MedLoc value. For information on this data type, see [“MedLoc Data Type” on page 3-7](#).

Example

This example references the **medloctest** table ([page 5-4](#)). The **Storage** function ([“Storage” on page 5-104](#)) returns the row from the **MedVSIRegister** table ([“MedVSIRegister Table” on page 3-6](#)) for the storage server of type VSI_FILE, #1. Then the **New** function returns a MedLoc value informing Dynamic Server where to create a new large media object on VSI_FILE#1. Because the **New** function was called from an INSERT statement, the creating operation occurs, and the returned MedLoc value is inserted into the **medloctest** table as a row identified by the string 'Empty Media'.

```
insert into medloctest values ('Empty Media', new(storage('VSI_FILE#1:')));
```

See Also

Server Functions:

[“Copy” on page 5-26](#)

Client API Functions:

N.A.

NotContained

Tests whether the first media chunk is not contained in the second media chunk.

Syntax

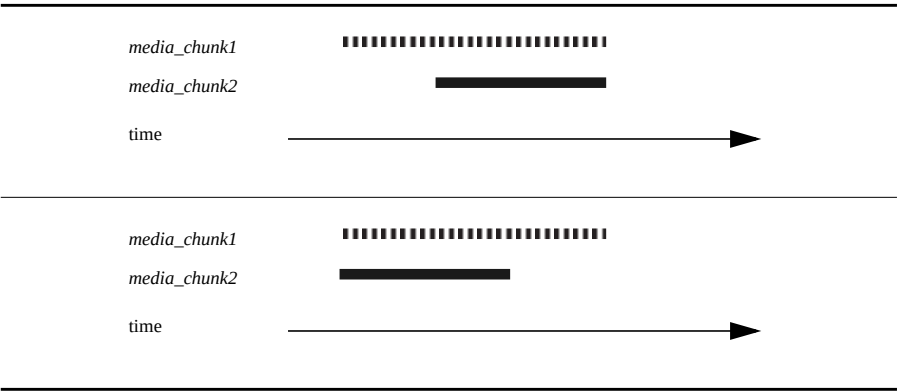
```
NotContained(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element is less than the second *StartP*, or if the first *EndP* element is greater than the second *EndP*, or both. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row’s MedChunk value is not contained in the MedChunk value of '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'NotContained',
       NotContained(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result; the MedChunk values in Rows 1 and 3 are not contained in the input media chunk:

id	(constant)	(expression)
1	NotContained	t
2	NotContained	f
3	NotContained	t

See Also

Server Functions:

[“Contained” on page 5-20](#)

[“Contains” on page 5-23](#)

[“NotContains” on page 5-77](#)

Client API Functions:

N.A.

NotContains

Tests whether the first media chunk does not contain the second media chunk.

Syntax

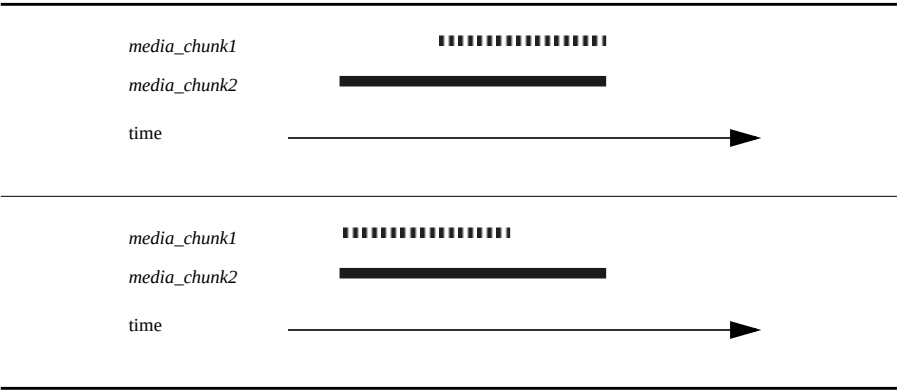
```
NotContains(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element is greater than the second *StartP*, or if the first *EndP* element is less than the second *EndP*, or both. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value does not contain the MedChunk value of '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'NotContains',
       NotContains(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result; the MedChunk values in rows 1 and 3 do not contain the input media chunk:

id	(constant)	(expression)
1	NotContains	t
2	NotContains	f
3	NotContains	t

See Also

Server Functions:

[“Contained” on page 5-20](#)

[“Contains” on page 5-23](#)

[“NotContained” on page 5-75](#)

Client API Functions:

N.A.

NotEqual

Tests whether two values of the same data type are not equal: that is, whether they do not reference the same media point, timecode, media object location, or media chunk.

Syntax

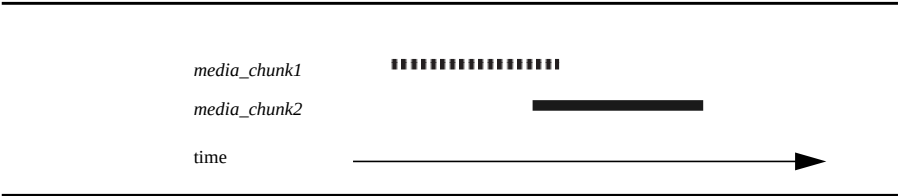
```
NotEqual(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies the data type values to compare.	Both values must be of the same data type.

Returns

Returns a Boolean value: *t* if *data_type1* does not equal *data_type2*; otherwise, *f*.

The following illustration shows a true condition for the MedChunk data type. Although the media chunks are the same size, they are not equal because they do not reference the *same* media chunk.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and tests whether that row's MedChunk value is not equal to the MedChunk value '20-00:00:00:20|40-00:00:01:10'.

```
select id, 'NotEqual',
       NotEqual(val, '20-00:00:00:20|40-00:00:01:10') from medchunktest;
```

This is the result; the MedChunk values in Rows 1 and 3 are not equal to the input media chunk:

```
id (constant) (expression)
1 NotEqual          t
2 NotEqual          f
3 NotEqual          t
```

See Also

Server Functions:

[“Compare” on page 5-16](#)

[“Equal” on page 5-30](#)

Client API Functions:

N.A.

NotMeets

Tests whether the first media chunk does not meet the second media chunk, either at its start or its end.

Syntax

```
NotMeets(media_chunk1, media_chunk2)
```

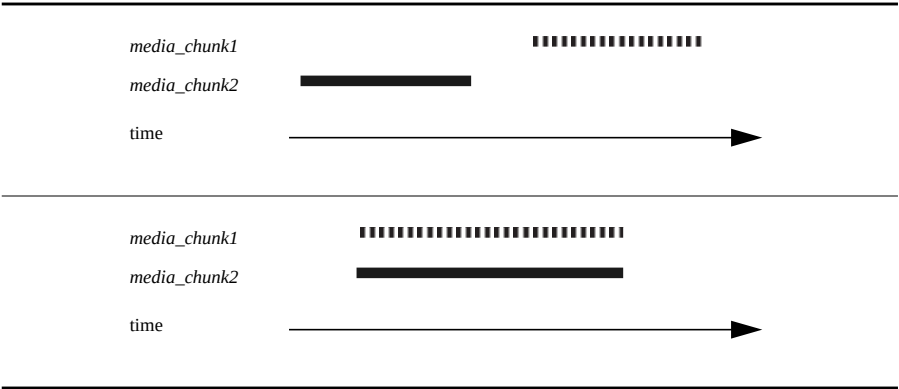
Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if the first *StartP* element does not equal the second *EndP* element, *and* if the first *EndP* element does not equal the second *StartP* element. Otherwise, it returns *f*.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value does not meet the MedChunk value of '10-00:00:00:10|40-00:00:01:10', either at its start or end.

```
select id, 'NotMeets',  
       NotMeets(val, '10-00:00:00:10|40-00:00:01:10') from medchunktest;
```

This is the result; the MedChunk values in Rows 2 and 3 do not meet the input media chunk:

id (constant) (expression)		
1	NotMeets	f
2	NotMeets	t
3	NotMeets	t

See Also

Server Functions:

[“Intersection” on page 5-42](#)

[“Meets” on page 5-62](#)

[“Overlap” on page 5-86](#)

Client API Functions:

N.A.

NotOverlap

Tests whether the first media chunk has no points in common with the second media chunk.

Syntax

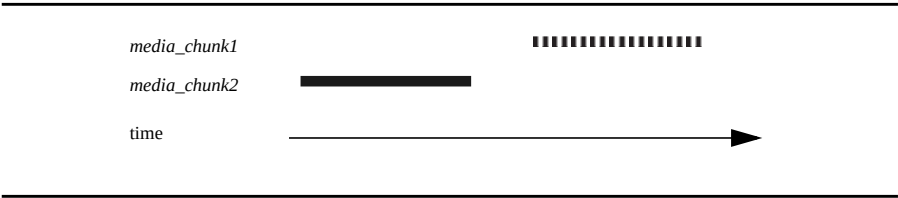
```
NotOverlap(media_chunk1, media_chunk2)
```

Argument	Purpose	Restrictions
<i>media_chunk1, media_chunk2</i>	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: *t* if *media_chunk1* does not overlap *media_chunk2*; otherwise, *f*.

The following illustration shows a true condition.



Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value does not overlap the MedChunk value of '10-00:00:00:10|40-00:00:01:10'.

```
select id, 'NotOverlap',
       NotOverlap(val, '10-00:00:00:10|40-00:00:01:10') from medchunktest;
```

This is the result; only the MedChunk value in Row 3 does not overlap the input media chunk:

id (constant) (expression)		
1	NotOverlap	f
2	NotOverlap	f
3	NotOverlap	t

See Also

Server Functions:

[“Intersection” on page 5-42](#)

[“Meets” on page 5-62](#)

[“Overlap” on page 5-86](#)

Client API Functions:

N.A.

Overlap

Tests whether the first media chunk has any points in common with the second media chunk.

Syntax

```
Overlap(media_chunk1, media_chunk2)
```

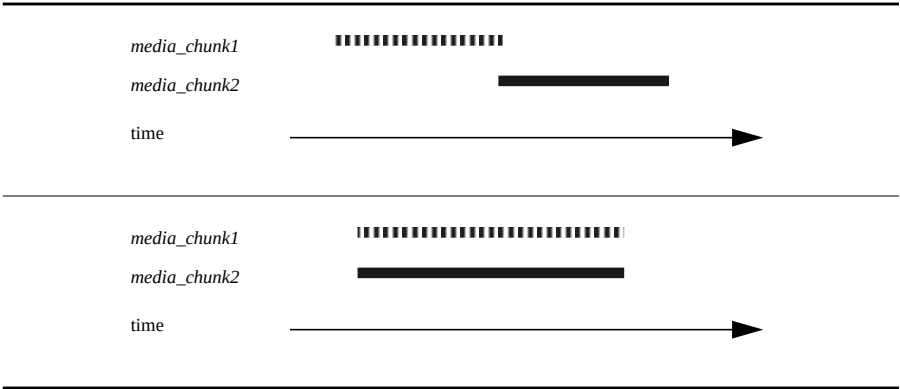
Argument	Purpose	Restrictions
media_chunk1, media_chunk2	Specifies the media chunks to compare.	Both values must be MedChunk values; see “MedChunk Data Type” on page 2-12.

Returns

Returns a Boolean value: `t` if *media_chunk1* overlaps *media_chunk2*, including if the first *StartP* element equals the second *EndP* element, or if the first *EndP* element equals the second *StartP* element. Otherwise, it returns `f`.

Each of the following illustrations shows a true condition.





Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row from the table and tests whether that row's MedChunk value overlaps the MedChunk value of '10-00:00:00:10|40-00:00:01:10'.

```
select id, 'Overlap',
       Overlap(val, '10-00:00:00:10|40-00:00:01:10') from medchunktest;
```

This is the result; the MedChunk values in Rows 1 and 2 overlap the input media chunk:

id (constant) (expression)		
1	Overlap	t
2	Overlap	t
3	Overlap	f

See Also

Server Functions:

[“Intersection” on page 5-42](#)

[“Meets” on page 5-62](#)

[“NotOverlap” on page 5-84](#)

Client API Functions:

N.A.



Plus

Takes two values of the same data type and adds them.

Tip: You cannot use this function with the *MedLoc* data type.

Syntax

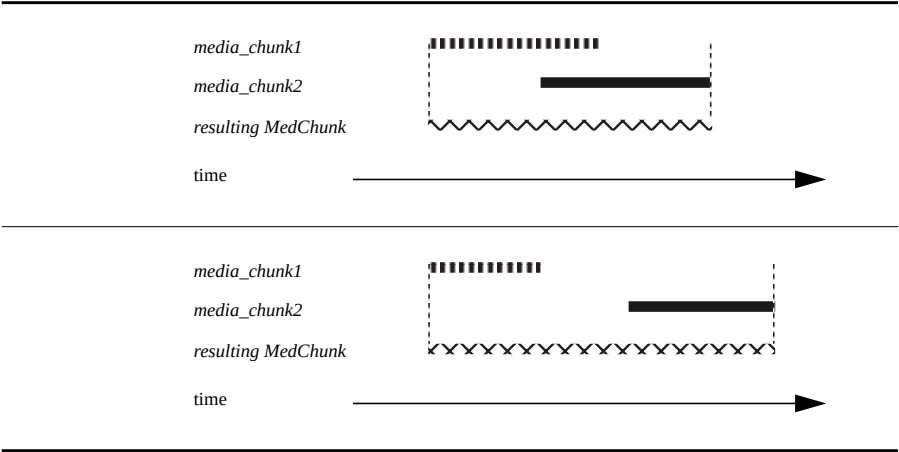
```
Plus(data_type1, data_type2)
```

Argument	Purpose	Restrictions
<i>data_type1</i> , <i>data_type2</i>	Specifies which data types are added.	Both values must be of the same data type.

Returns

The returned result is of the same data type as the inputs.

For the *MedChunk* data type (see [“MedChunk Data Type” on page 2-12](#)), the result is made up of the lesser of the two *StartP* elements and the greater of the two *EndP* elements. If the two inputs have no points in common, the constructed *MedChunk* value might not represent an existing piece of media.



Example

This section presents examples of using the **Plus** function to add MedPtr values, MedTC values, and MedChunk values.

MedPtr Data Type

This example references the **medptrtest** table ([page 5-3](#)). The statement selects the row and returns a MedPtr value that is the result of adding the row's MedPtr value to the MedPtr value '10-00:00:00:10'.

```
insert into medptrtest values (1, '10-00:00:00:10');
select id, 'Plus', val + '40-00:00:01:10' from medptrtest;
```

This is the result:

id	1
(constant)	Plus
(expression)	50000000N

MedTC Data Type

This example references the **medtctest** table ([page 5-3](#)). The statement returns the *HH* element of the MedTC value that resulted from adding '00:53:12:19-F' to '01:34:25:16-F'.

```
select hour( tc + '01:34:25:16-F' ) from medtctest where
fps(tc)=24;
```

The result of the addition is '02:27:38:11-F'; the result of the statement is 2.

MedChunk Data Type

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns a MedChunk value that is the result of adding the row’s MedChunk value to the MedChunk value '15-00:00:00:15|40-00:00:01:10'.

```
select id, 'Plus',  
       val + '15-00:00:00:15|40-00:00:01:10' from medchunktest;
```

These are the results:

id	1
(constant)	Plus
(expression)	0N-00:00:00:00 40000000N-00:00:01:10
id	2
(constant)	Plus
(expression)	15000000N-00:00:00:15 40000000N-00:00:01:10
id	3
(constant)	Plus
(expression)	15000000N-00:00:00:15 60000000N-00:00:02:00

See Also

Server Functions:

[“Minus” on page 5-64](#)

Client API Functions:

[“medChunkPlus” on page 6-19](#)

[“medPtrPlus” on page 6-38](#)

[“medTCPlus” on page 6-64](#)

Second

Returns the SS element (the seconds) of a MedTC value.

Syntax

```
Second(timecode)
```

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the SS element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an INTEGER value that specifies the seconds of the timecode.

Example

The following example references the **medtctest** table ([page 5-3](#)). The example uses the **Second** and **Type** ([“Type \(MedTC\)” on page 5-112](#)) functions to select the row where the SS element is not 0 and the video standard is film.

```
select vidID, tc from medtctest where second(tc) <> 0 and type(tc)='F';
```

This is the result:

```
vidid  1000
tc      00:53:12:19-F
```

See Also

Server Functions:

[“Fps” on page 5-32](#)

[“Frame” on page 5-34](#)

[“Hour” on page 5-40](#)

[“Minute” on page 5-69](#)

[“Type \(MedTC\)” on page 5-112](#)

Client API Functions:

[“medTCGetSecond” on page 6-58](#)

Shutdown

Closes the connection between Informix Dynamic Server and the specified storage server and releases any resources associated with the connection.

Syntax

```
Shutdown(storage_server)
```

Argument	Purpose	Restrictions
<i>storage_server</i>	Specifies the storage server from which to disconnect.	Must be a MedStorage row value; see “External Representation” on page 3-5 .

Returns

Returns an INTEGER value that specifies the status of the connection. See [“Status” on page 5-102](#) for information on this value’s bit settings.

Example

This example references the Informix Video Foundation DataBlade module’s **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)). The statement shuts down all storage servers in the media management system of type VSI_FILE.

```
select shutdown(storage) from MedVSIRegister where storage.Type='VSI_FILE';
```

See Also

Server Functions:

[“Startup” on page 5-100](#)

[“Status” on page 5-102](#)

Client API Functions:

N.A.

Size

Returns the size of a media chunk, where *size* is the difference between the MedChunk’s *StartP* and *EndP* elements, in nanoseconds.

Syntax

`Size(media_chunk)`

Argument	Purpose	Restrictions
<i>media_chunk</i>	Specifies the media chunk for which the caller wants the size.	Must be a MedChunk value; see “ MedChunk Data Type ” on page 2-12 .

Returns

Returns an INT8 value that specifies the size of the media chunk in nanoseconds.

Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns the size of that row’s MedChunk value, in nanoseconds.

```
select id, 'Size', Size(val) from medchunktest;
```

These are the results:

id (constant)		(expression)
1	Size	10000000
2	Size	20000000
3	Size	10000000

See Also

Server Functions:

[“EndP” on page 5-28](#)

[“StartP” on page 5-98](#)

Client API Functions:

[“medChunkGetSize” on page 6-11](#)

StartP

Returns the *StartP* element (the starting media point) of a MedChunk value.

Syntax

```
StartP(media_chunk)
```

Argument	Purpose	Restrictions
<i>media_chunk</i>	Specifies the media chunk for which the caller wants the starting media point.	Must be a MedChunk value; see “MedChunk Data Type” on page 2-12 .

Returns

Returns a MedPtr value. For information on this data type, see [“MedPtr Data Type” on page 2-3](#).

Example

This example references the **medchunktest** table ([page 5-4](#)). The statement selects each row in the table and returns the starting media point of that row’s MedChunk value.

```
select id, 'StartP', StartP(val) from medchunktest;
```

These are the results:

id	1
(constant)	StartP
(expression)	0N-00:00:00:00
id	2
(constant)	StartP
(expression)	20000000N-00:00:00:20
id	3
(constant)	StartP
(expression)	50000000N-00:00:01:20

See Also

Server Functions:

[“EndP” on page 5-28](#)

[“Size” on page 5-96](#)

Client API Functions:

[“medChunkGetStartPtr” on page 6-13](#)

Startup

Initializes the connection between Informix Dynamic Server and the specified storage server.

Syntax

```
Startup(storage_server)
```

Argument	Purpose	Restrictions
<i>storage_server</i>	Specifies the storage server with which to connect.	Must be a MedStorage row value; see “MedStorage Data Type” on page 3-4 .

Returns

Returns an INTEGER value that specifies the status of the connection. See [“Status” on page 5-102](#) for information on this value’s bit settings.

Example

This example references the Informix Video Foundation DataBlade module’s **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)). The statement initializes all storage servers in the media management system of type VSI_FILE.

```
select startup(storage) from MedVSIRegister where storage.Type='VSI_FILE';
```

See Also

Server Functions:

[“Shutdown” on page 5-94](#)

[“Status” on page 5-102](#)

Client API Functions:

N.A.

Status

Returns the status of the connection between Informix Dynamic Server and the specified storage server.

Syntax

```
Status(storage_server)
```

Argument	Purpose	Restrictions
<i>storage_server</i>	Specifies the storage server from which to get status information.	Must be a MedStorage row value; see “MedStorage Data Type” on page 3-4.

Returns

Returns an INTEGER value that specifies the status of the connection. If this value is 0, communication can occur between the caller, Dynamic Server, and the storage server.

The following table shows the 16-bit integer’s settings.

Bit	Settings
0	0 = on-line 1 = off-line
1–5	0 = no error Nonzero = vendor-specific errors
6–10	0 = not busy Nonzero = vendor-specific busy signals
11–15	Reserved

Example

This example references the Informix Video Foundation DataBlade module's **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)). The statement returns the status for all storage servers in the media management system of type VSI_FILE.

```
select status(storage) from MedVSIRegister where storage.Type='VSI_FILE';
```

See Also

Server Functions:

[“Shutdown” on page 5-94](#)

[“Startup” on page 5-100](#)

Client API Functions:

N.A.

Storage

Returns the MedStorage row value of the **MedVSIRegister** table associated with a MedLoc value.

Syntax

```
Storage(media_object)
```

Argument	Purpose	Restrictions
media_object	Specifies the large media object for which the caller wants the associated MedStorage row value.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 . If the MedLoc’s Type and StorageID elements are valid, the MedLoc value need not represent an existing large media object.

Returns

Returns a MedStorage row value.

If the MedLoc type is VSI_CLIENT, the returned MedLoc row value’s **ConnectionString** field is set to the MedLoc’s location.

Example

This example references the **medloctest** table ([page 5-4](#)). The **Storage** function ([“Storage” on page 5-104](#)) returns the row from the **MedVSIRegister** table ([“MedVSIRegister Table” on page 3-6](#)) for the storage server of type VSI_FILE, #1. Then the **New** function requests that an empty MedLoc value be created on that storage server. The statement inserts that new MedLoc value into the table as a row identified by the string 'New Media'.

```
insert into medloctest values ('New Media', new(storage('VSI_FILE#1:')));
```


See Also

Server Functions:

[“Location” on page 5-48](#)

[“StorageID” on page 5-106](#)

[“Type \(MedLoc\)” on page 5-110](#)

Client API Functions:

N.A.

StorageID

Returns the *StorageID* element of a MedLoc value.

Syntax

```
StorageID(media_object)
```

Argument	Purpose	Restrictions
<i>media_object</i>	Specifies the large media object for which the caller wants the <i>StorageID</i> element.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 .

Returns

Returns an INTEGER value that specifies the storage ID number.

Example

This example references the **medloctest** table ([page 5-4](#)) and the row that was inserted into the table in the **Copy** function’s example ([“Copy” on page 5-26](#)). The statement returns the storage server’s *StorageID* element from the row identified with the string 'My Video'.

```
select storageID(loc) from medloctest where name='My Video';
```

This is the result:

```
(expression)
1
```

See Also

Server Functions:

[“Location” on page 5-48](#)

[“Storage” on page 5-104](#)

[“Type \(MedLoc\)” on page 5-110](#)

Client API Functions:

[“medLocGetStorageID” on page 6-28](#)

Time

Returns the *Time* element, in nanoseconds, of a MedPtr value.

Syntax

```
Time(media_point)
```

Argument	Purpose	Restrictions
media_point	Specifies the media point for which the caller wants the <i>Time</i> element.	Must be a MedPtr value; see “MedPtr Data Type” on page 2-3 .

Returns

Returns an INT8 value that specifies the *Time* element in nanoseconds.

Example

This example references the **medptrtest** table ([page 5-3](#)). The statement selects the row and returns the *Time* element of its MedPtr value.

```
insert into medptrtest values (1, '40-00:00:01:10');
select id, 'Time', Time(val) from medptrtest;
```

This is the result:

id (constant)	(expression)
1	Time 40000000

See Also

Server Functions:

[“Native” on page 5-71](#)

Client API Functions:

[“medPtrGetTime” on page 6-34](#)

Type (MedLoc)

Returns the *Type* element (the storage server type) of a MedLoc value.

Syntax

Type(*media_object*)

Argument	Purpose	Restrictions
<i>media_object</i>	Specifies the large media object for which the caller wants the <i>Type</i> element.	Must be a MedLoc value; see “MedLoc Data Type” on page 3-7 .

Returns

Returns an LVARCHAR value that specifies the *Type* element.

Example

This example references the **medloctest** table ([page 5-4](#)) and the row that was inserted into the table in the **Copy** function’s example ([“Copy” on page 5-26](#)). The statement returns the storage server’s *Type* element from the row identified with the string 'My Video'.

```
select type(loc) from medloctest where name='My Video';
```

This is the result:

```
(expression)  VSI_FILE
```

See Also

Server Functions:

[“Location” on page 5-48](#)

[“Storage” on page 5-104](#)

[“StorageID” on page 5-106](#)

Client API Functions:

[“medLocGetType” on page 6-30](#)

Type (MedTC)

Returns the *-Standard_flag* element (the video standard type) of a MedTC value.

Syntax

Type(*timecode*)

Argument	Purpose	Restrictions
<i>timecode</i>	Specifies the timecode for which the caller wants the <i>-Standard_flag</i> element.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an LVARCHAR value that specifies the video standard (and thus the valid frame numbers) of this timecode, as follows.

Standard Flag	Video Standard	Valid Frame Numbers
D	Drop-frame NTSC	02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.
F	Film	00 through 23.
N	NTSC	00 through 29.
P	PAL	00 through 24.

Example

The following example references the **medtctest** table ([page 5-3](#)). It uses the **Second** (“[Second](#)” on [page 5-92](#)) and **Type** functions to select the row where the SS element is not 0 and the video standard is film.

```
select vidID, tc from medtctest where second(tc) <> 0 and type(tc)='F';
```

This is the result:

```
vidid  1000
tc      00:53:12:19-F
```

See Also

Server Functions:

[“Fps” on page 5-32](#)

[“Frame” on page 5-34](#)

[“Hour” on page 5-40](#)

[“Minute” on page 5-69](#)

[“Second” on page 5-92](#)

Client API Functions:

[“medTCGetType” on page 6-60](#)



VidFrameNumberFrom

Converts one or more input arguments of various types into an integer value that represents an absolute frame number.

Tip: The first frame number in a piece of media is 0. Thus, to calculate the number of frames in the media, you must add 1 to the result of this function.

Syntax

To convert a MedPtr value and a frames-per-second FLOAT value to a frame number, use:

```
VidFrameNumberFrom(media_point, fps)
```

To convert a MedTC value to a frame number, use:

```
VidFrameNumberFrom(timecode)
```

Argument	Purpose	Restrictions
<i>media_point</i>	Specifies the media point used (with <i>fps</i> , next row) in calculating the frame number.	Must be a MedPtr value; see “MedPtr Data Type” on page 2-3 .
<i>fps</i>	Specifies the frames-per-second rate used (with <i>media_point</i> , previous row) in calculating the frame number.	Must be a positive number.
<i>timecode</i>	Specifies the timecode to convert to a frame number.	Must be a MedTC value; see “MedTC Data Type” on page 2-7 .

Returns

Returns an INTEGER value that specifies the frame number.

Example

This section presents examples of using the **VidFrameNumberFrom** function to convert input values of various data types into frame numbers.

Input of MedPtr, Fps Values

This example references the **medptrtest** table ([page 5-3](#)). The **MedPtrFrom** function converts a MedPtr value and an *Fps* value (FLOAT) into a frame number integer. The statement inserts that value into the table and then selects it.

```
insert into medptrtest values (1, '1334668001N');
select id, 'VidFrameNumberFrom',
       VidFrameNumberFrom(val, 29.97) from medptrtest;
```

This is the result:

id (constant)	(expression)
1	VidFrameNumberFrom 40

Input of MedTC Value

This example references the **medtctest** table ([page 5-3](#)) and the **medtctest3** table (created here). The **VidFrameNumberFrom** function is used to convert the entries in the **medtctest** table into integer values. Both the integer frame numbers and the original MedTC values are inserted into **medtctest3**. Finally, the entries of the **medtctest3** table are selected.

```
create table medtctest3 (frameNum integer, tc medtc);
insert into medtctest3 select vidframenumberfrom(tc),tc from medtctest;
select * from medtctest3;
```

This is the result:

```
framenum  76627
tc         00:53:12:19-F

framenum  125874
tc         01:10:00:00-D
```

See Also

Server Functions:

[“MedChunkFrom” on page 5-50](#)

[“MedPtrFrom” on page 5-53](#)

[“MedTCFrom” on page 5-59](#)

Client API Functions:

[“medPtrToFrameNumber” on page 6-40](#)

[“medTCToFrameNumber” on page 6-66](#)

Client API

In This Chapter	6-3
The Client API Library	6-3
Video Foundation Class	6-4
Client Functions	6-6
medChunkGetEndPtr	6-9
medChunkGetSize	6-11
medChunkGetStartPtr	6-13
medChunkMinus	6-15
medChunkPlus	6-19
medFrameNumberToMedPtr	6-22
medFrameNumberToMedTC	6-24
medLocGetLocation	6-26
medLocGetStorageID	6-28
medLocGetType	6-30
medPtrGetNative	6-32
medPtrGetTime	6-34
medPtrMinus	6-36
medPtrPlus	6-38
medPtrToFrameNumber	6-40
medPtrToMedChunk	6-43
medPtrToMedTC	6-45
medPtrsToMedChunk	6-47
medTCGetFPS	6-49
medTCGetFrame	6-52
medTCGetHour	6-54
medTCGetMinute	6-56

medTCGetSecond	6-58
medTCGetType	6-60
medTCMinus	6-62
medTCPlus	6-64
medTCToFrameNumber.	6-66
medTCToMedPtr	6-68
medTimeNativeToMedPtr	6-70

In This Chapter

This chapter describes the Informix Video Foundation DataBlade module's client API. This API provides the following services to developers who create video applications:

- The Video Foundation Class, which consists of the Informix Video Foundation DataBlade module's opaque data types as value objects that expose an **ITValue** interface, for developers using Informix's C++ Interface.
- A subset of this DataBlade module's server functions as client-side functions. They provide the same services as their server-side counterparts but execute on the client system rather than being processed by Dynamic Server. These functions are available to developers using Informix's ESQL/C, DataBlade API, or C++ interfaces.

This chapter first describes how to include these services in your application and describes each of them.

The Client API Library

To install the client API library, first copy the **\$INFORMIXDIR/extend/Video.1.10/lib/client/windows/vid_win32.exe** file to your Windows 95 or Windows NT client computer. Invoke this executable file to install the **\$INFORMIXDIR\bin\vidfc.dll** file on your client computer and then follow the instructions in this section to include the library in your application.

Add the appropriate include statement to your application. If you are using ESQL/C or DataBlade API, include the **\$INFORMIXDIR\incl\video\convert.h** file:

```
#include "convert.h"
```

If you are using C++, include the **\$INFORMIXDIR\incl\video\videodb.h** file (which includes the **convert.h** file):

```
#include "videodb.h"
```

Next, link in the **\$INFORMIXDIR\lib\vidfc.lib** file, which contains the conversion functions and the link points for the Video Foundation Class.

For more information and guidelines, consult one of these Informix guides:

- [C++ Interface Programmer's Guide](#)
- [DataBlade API Programmer's Manual](#)
- [INFORMIX-ESQL/C Programmer's Manual](#)

Video Foundation Class

The Video Foundation Class provides the Informix Video Foundation DataBlade module's opaque data types as value objects, as follows.

Opaque Data Type	Value Object
MedChunk	ITMedChunk
MedLoc	ITMedLoc
MedPtr	ITMedPtr
MedTC	ITMedTC

For each Video Foundation Class value object, all of the **ITEssential** and **ITValue** methods are provided. The following is a list of those methods for your convenience; for details, refer to the [C++ Interface Programmer's Guide](#):

- ITOpErrorCode QueryInterface(const ITInterfaceID &iid, void **pointer_to_interface)
- unsigned long AddRef()
- unsigned long Release()
- const ITString &Printable()
- const ITTypeInfo &TypeOf()
- ITBool IsNull()

- `ITBool SameType(ITValue *)`
- `ITBool Equal(ITValue *)`
- `ITBool LessThan(ITValue *)`
- `ITBool CompatibleType(ITValue *)`
- `ITBool IsUpdated()`
- `ITBool FromPrintable(const ITString &printable)`
- `ITBool SetNull()`

In addition, the following functions are provided for each value object, where *ITValueName* is the name of the opaque data type, such as *ITMedChunk*:

- `ITValueName(ITMVDesc *)`
- `~ITValueName()`
- `static ITValueName *MakeValueName(ITMVDesc *)`

Important: *There is no default constructor function for these value objects.*

The following query interfaces are not exposed for the Video Foundation Class value objects:

- **ITRow**
- **ITConversions**
- **ITLargeObject**
- **ITSet**

The following reminders are from the [C++ Interface Programmer's Guide](#); for details, refer to that guide:

- Informix's C++ Interface does not support "object persistence" for application classes; it does not automatically map instances of database data types to application classes, or vice versa.
- To modify the values in the database that correspond to the value objects returned to client applications, you must issue SQL queries. You cannot directly update the database data by modifying the value objects.
- You cannot develop Dynamic Server functions using the C++ Interface.
- Do not mix database access through the C++ Interface and lower-level interfaces (such as LibDMI) in the same application.



Client Functions

The client functions provided in this API provide the same services as their server-side counterparts, but they execute on your client system rather than being processed by Dynamic Server.

The following types of client functions are provided for each of the Video Foundation DataBlade module's opaque data types (MedChunk, MedLoc, MedPtr, and MedTC), except where noted:

- The **Minus** and **Plus** functions (although they do not support the MedLoc data type).
- All functions that return an element of a data type value (such as the **Time** function to return the *Time* element of a MedPtr value).
- Data type conversion functions, including client-side versions of **VidFrameNumberFrom**. No such functions are provided for MedLoc.



Important: *MedStorage is a named row data type, not an opaque type, so no client-side functions are provided for it.*

The client functions test all input and return values to make sure the opaque types are valid, according to the definitions provided in [Chapter 2, “Temporal Component,”](#) and [Chapter 3, “Storage Component,”](#) of this guide.

The following table shows the server-side functions and their corresponding client-side functions. For details on the server functions, see [Chapter 5, “Server Functions.”](#) For details on the client functions, see the reference section starting with [“medChunkGetEndPtr” on page 6-9.](#)

Server-Side Function	Client-Side Function
EndP(MedChunk)	medChunkGetEndPtr
Fps(MedTC)	medTCGetFPS
Frame(MedTC)	medTCGetFrame
Hour(MedTC)	medTCGetHour
Location(MedLoc)	medLocGetLocation

(1 of 3)

Server-Side Function	Client-Side Function
MedChunkFrom(MedPtr)	medPtrToMedChunk
MedChunkFrom(MedPtr, MedPtr)	medPtrsToMedChunk
MedPtrFrom(<i>frame_number, frame_rate</i>)	medFrameNumberToMedPtr
MedPtrFrom(MedTC)	medTCToMedPtr
MedPtrFrom(<i>time, native</i>)	medTimeNativeToMedPtr
MedTCFrom(<i>frame_number, frame_rate</i>)	medFrameNumberToMedTC
MedTCFrom(MedPtr)	medPtrToTime
MedTCFrom(MedPtr, <i>frame_rate</i>)	medPtrToMedTC
Minus(MedChunk, MedChunk)	medChunkMinus
Minus(MedPtr, MedPtr)	medPtrMinus
Minus(MedTC, MedTC)	medTCMinus
Minute(MedTC)	medTCGetMinute
Native(MedPtr)	medPtrGetNative
Plus(MedChunk, MedChunk)	medChunkPlus
Plus(MedPtr, MedPtr)	medPtrPlus
Plus(MedTC, MedTC)	medTCPlus
Second(MedTC)	medTCGetSecond
Size(MedChunk)	medChunkGetSize
StartP(MedChunk)	medChunkGetStartPtr
StorageID(MedLoc)	medLocGetStorageID
Time(MedPtr)	medPtrGetTime
Type(MedLoc)	medLocGetType
Type(MedTC)	medTCGetType

(2 of 3)

Server-Side Function	Client-Side Function
VidFrameNumberFrom(MedPtr, <i>frame_rate</i>)	medPtrToFrameNumber
VidFrameNumberFrom(MedTC)	medTCToFrameNumber

(3 of 3)

The rest of this chapter provides the following information for each client function:

- A brief description of the task it performs
- Its syntax
- A brief description of each of its input arguments
- A brief description of the value it returns
- Example code that includes the function
- A list of other client API functions and server functions (see [Chapter 5, “Server Functions”](#)) that might be of interest to you

medChunkGetEndPtr

Returns the *EndP* element (the ending media point) of a *MedChunk* value (see “[MedChunk Data Type](#)” on page 2-12).

Syntax

```
MedPtr *medChunkGetEndPtr(MedChunk *media_chunk)
```

Argument	Purpose	Restrictions
<i>*media_chunk</i>	Specifies the media chunk for which the caller wants the ending media point.	N.A.

Returns

Returns a pointer to a *MedPtr* value (“[MedPtr Data Type](#)” on page 2-3). On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8Time;
MedPtr *mpMedPtrStart, *mpMedPtrEnd, *mpResultPtr;
MedChunk *mcMedChunk;

/* The MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
```

```
mcMedChunk = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

mpResultPtr = medChunkGetEndPtr(mcMedChunk);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout, "medChunkGetEndPtr result:\nTime %s, Native
%s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medChunkGetEndPtr result: Time 108000000000000, Native
 * 00:03:00:00
 */

/* Clean up after ourselves */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);
mi_free(mcMedChunk);
mi_free(mpResultPtr);
```

See Also

Client API Functions:

[“medChunkGetSize” on page 6-11](#)

[“medChunkGetStartPtr” on page 6-13](#)

Server Functions:

[“EndP” on page 5-28](#)

medChunkGetSize

Returns the size of a MedChunk value (see “MedChunk Data Type” on [page 2-12](#)), where *size* is the difference between the MedChunk value’s *StartP* and *EndP* elements, in nanoseconds.

Syntax

```
mi_int8 *medChunkGetSize(MedChunk *media_chunk)
```

Argument	Purpose	Restrictions
<i>*media_chunk</i>	Specifies the media chunk for which the caller wants the size.	N.A.

Returns

Returns a pointer to an *mi_int8* value that specifies the size of the media chunk in nanoseconds. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8Time;
mi_int8 *i8TheChunkDuration;
MedPtr *mpMedPtrStart, *mpMedPtrEnd;
MedChunk *mcMedChunk;

/* The MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
```

```
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

i8TheChunkDuration = medChunkGetSize(mcMedChunk);

/* Look at the results, it should be 2 hours */
ifx_int8toasc(i8TheChunkDuration, szNumString, STRING_SIZE);
fprintf (stdout, "medChunkGetSize result: Time %s\n",
szNumString);

/* Output is:
 * medChunkGetSize result: Time 7200000000000
 */

/* Clean up after ourselves */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);
mi_free(mcMedChunk);
```

See Also

Client API Functions:

[“medChunkGetEndPtr” on page 6-9](#)

[“medChunkGetStartPtr” on page 6-13](#)

Server Functions:

[“Size” on page 5-96](#)

medChunkGetStartPtr

Returns the *StartP* element (the starting media point) of a MedChunk value (see “MedChunk Data Type” on page 2-12).

Syntax

```
MedPtr *medChunkGetStartPtr(MedChunk *media_chunk)
```

Argument	Purpose	Restrictions
<i>*media_chunk</i>	Specifies the media chunk for which the caller wants the starting media point.	N.A.

Returns

Returns a pointer to a MedPtr value (“MedPtr Data Type” on page 2-3). On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8Time;
MedPtr *mpMedPtrStart, *mpMedPtrEnd, *mpResultPtr;
MedChunk *mcMedChunk;

/* The MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
```

```
mcMedChunk = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

mpResultPtr = medChunkGetStartPtr(mcMedChunk);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf(stdout, "medChunkGetStartPtr result:\nTime %s, Native
%s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medChunkGetStartPtr result: Time 3600000000000, Native
 * 00:01:00:00
 */

/* Clean up after ourselves */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);
mi_free(mcMedChunk);
mi_free(mpResultPtr);
```

See Also

Client API Functions:

[“medChunkGetEndPtr” on page 6-9](#)

[“medChunkGetSize” on page 6-11](#)

Server Functions:

[“StartP” on page 5-98](#)

medChunkMinus

Takes two MedChunk values (see “MedChunk Data Type” on page 2-12) and subtracts the second one from the first.

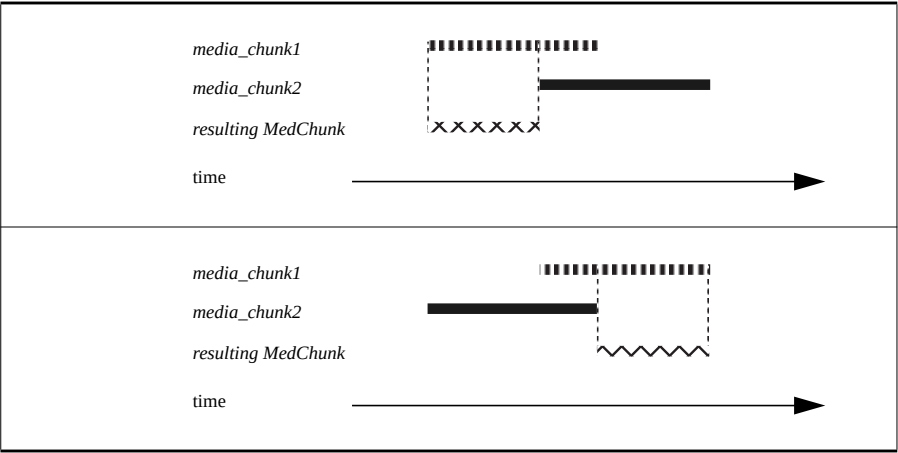
Syntax

```
MedChunk *medChunkMinus(MedChunk *media_chunk1,  
                        MedChunk *media_chunk2)
```

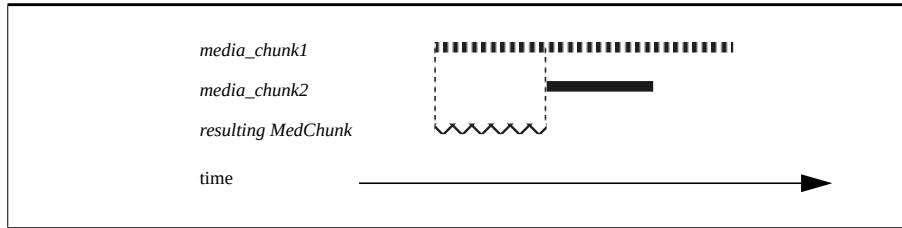
Argument	Purpose	Restrictions
<i>*media_chunk1, *media_chunk2</i>	Specifies which media chunk is subtracted (<i>*media_chunk2</i>) from which (<i>*media_chunk1</i>).	N.A.

Returns

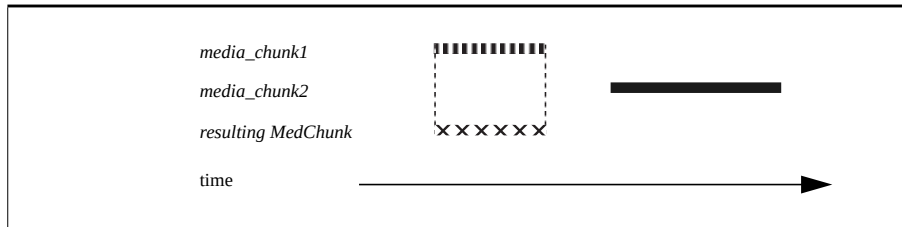
Returns a pointer to a MedChunk value consisting of those points of *media_chunk1* not shared with *media_chunk2*.



If there are two pieces of *media_chunk1* not shared with *media_chunk2*, only the first piece is returned.



If no points are shared, *media_chunk1* is returned:



On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE], szAnotherString[STRING_SIZE];
mi_int8 i8Time;
MedPtr *mpMedPtrStart, *mpMedPtrEnd;
MedChunk *mcMedChunk1, *mcMedChunk2, *mpResultChunk;

/* First MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk1 = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);
```

```

/* A little clean up */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);

/* Second MedChunk */
/* This chunk goes from the 2 hour to the 4 hour mark on the
video */
strcpy(szNativeString, "00:02:00:00");
strcpy(szNumString, "7200000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:04:00:00");
strcpy(szNumString, "14400000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk2 = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

/* A little clean up */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);

/* Do the Math */
mpResultChunk = medChunkMinus(mcMedChunk1, mcMedChunk2);

/* Look at the results */
ifx_int8toasc(&(mpResultChunk->Start), szNumString,
STRING_SIZE);
ifx_int8toasc(&(mpResultChunk->End), szAnotherString,
STRING_SIZE);
fprintf (stdout,
"medChunkMinus result:\nStartTime %s, EndTime %s, Natives
%s\n",
szNumString, szAnotherString, mpResultChunk->Natives);

/* Output is:
* medChunkMinus result: StartTime 3600000000000, EndTime
7200000000000,
* Natives 00:01:00:00
*/

/* Clean up after ourselves */
mi_free(mpResultChunk);

```

See Also

Client API Functions:

[“medChunkPlus” on page 6-19](#)

Server Functions:

[“Minus” on page 5-64](#)

medChunkPlus

Takes two MedChunk values (see “MedChunk Data Type” on page 2-12) and adds them.

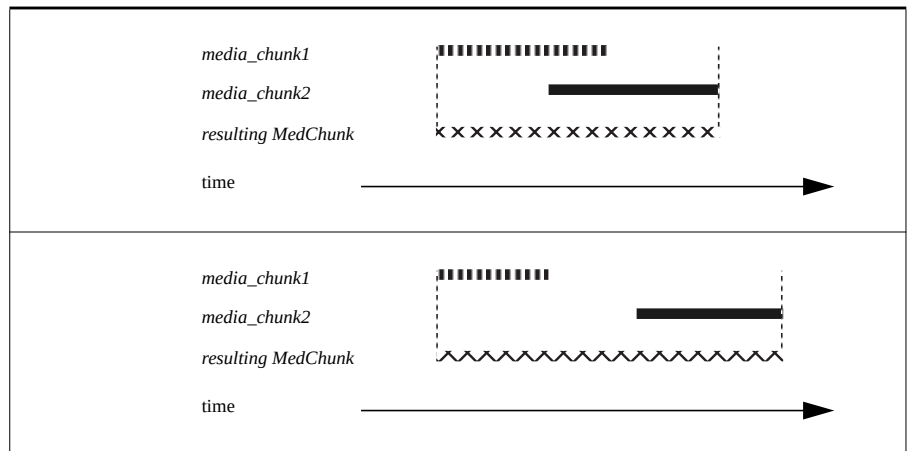
Syntax

```
MedChunk *medChunkPlus(MedChunk *media_chunk1,
                        MedChunk *media_chunk2)
```

Argument	Purpose	Restrictions
<i>*media_chunk1, *media_chunk2</i>	Specifies the media chunks to add.	N.A.

Returns

Returns a pointer to a MedChunk value that consists of the lesser of the two *StartP* elements and the greater of the two *EndP* elements. If the two inputs have no points in common, the constructed MedChunk value might not represent an existing piece of media.



On error, returns a null pointer.

Example

```

char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE], szAnotherString[STRING_SIZE];
mi_int8 i8Time;
MedPtr *mpMedPtrStart, *mpMedPtrEnd;
MedChunk *mcMedChunk1, *mcMedChunk2, *mpResultChunk;

/* First MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk1 = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

/* A little clean up */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);

/* Second MedChunk */
/* This chunk goes from the 2 hour to the 4 hour mark on the
video */
strcpy(szNativeString, "00:02:00:00");
strcpy(szNumString, "7200000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);

strcpy(szNativeString, "00:04:00:00");
strcpy(szNumString, "14400000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk2 = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

/* A little clean up */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);

/* Do the Math */

```



```

mpResultChunk = medChunkPlus(mcMedChunk1, mcMedChunk2);

/* Look at the results */
ifx_int8toasc(&(mpResultChunk->Start), szNumString,
STRING_SIZE);
ifx_int8toasc(&(mpResultChunk->End), szAnotherString,
STRING_SIZE);
fprintf (stdout,
        "medChunkPlus result:\nStartTime %s, EndTime %s, Natives
%s\n",
        szNumString, szAnotherString, mpResultChunk->Natives);

/* Output is:
 * medChunkPlus result: StartTime 3600000000000, EndTime
14400000000000,
 *      Natives 00:01:00:00
 */

/* Clean up after ourselves */
mi_free(mpResultChunk);

```

See Also

Client API Functions:

[“medChunkMinus” on page 6-15](#)

Server Functions:

[“Plus” on page 5-89](#)

medFrameNumberToMedPtr

Converts a frame number and its frames-per-second rate into a MedPtr value. For information on the MedPtr data type, see [“MedPtr Data Type” on page 2-3](#).

Syntax

```
MedPtr *medFrameNumberToMedPtr(mi_integer frame_num,
                                mi_double_precision *fps)
```

Argument	Purpose	Restrictions
<i>frame_num</i>	Specifies the frame number used (with <i>*fps</i> , next row) in calculating the <i>Time</i> element. This same input is used for the <i>Native</i> element.	Must be a positive number.
<i>*fps</i>	Specifies the frames-per-second rate used (with <i>frame_num</i> , previous row) in calculating the <i>Time</i> element.	Must be a positive number.

Returns

Returns a pointer to a MedPtr value. On error, returns a null pointer.

Example

```

MedPtr *mpResultPtr;
mi_double_precision theFrameRate;
char szNumString[STRING_SIZE];

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mpResultPtr = medFrameNumberToMedPtr(2000, &theFrameRate);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout, "medFrameNumberToMedPtr result: Time %s,
Native %s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medFrameNumberToMedPtr result: Time 66733400067, Native
2000
 */

/* Clean up after ourselves */
mi_free(mpResultPtr);

```

See Also

Client API Functions:

[“medPtrToFrameNumber” on page 6-40](#)

[“medTCToMedPtr” on page 6-68](#)

[“medTimeNativeToMedPtr” on page 6-70](#)

Server Functions:

[“MedPtrFrom” on page 5-53](#)

medFrameNumberToMedTC

Converts a frame number and its SMPTE frames-per-second rate into a MedTC value. For information on the MedTC data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
MedTC *medFrameNumberToMedTC(mi_integer frame_num,
                               mi_double_precision *fps)
```

Argument	Purpose	Restrictions
<i>frame_num</i>	Specifies the frame number used (with <i>*fps</i> , next row) in calculating the media timecode.	Must be a positive number.
<i>*fps</i>	Specifies the frames-per-second rate used (with <i>frame_num</i> , previous row) in calculating the media timecode.	Must be a valid SMPTE frame rate: 24.0, 25.0, 29.97, or 30.0.

Returns

Returns a pointer to a MedTC value. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(2000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 2000 at 29.97 fps is %.2d:%.2d:%.2d:%.2d-D\n",
        mtResultTC->hours,
        mtResultTC->minutes,
        mtResultTC->seconds,
        mtResultTC->frames);

/* Output is:
 * medFrameNumberToMedTC result: Frame 2000 at 29.97 fps is
 * 00:01:06:22-D
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medTCToFrameNumber” on page 6-66](#)

[“medPtrToMedTC” on page 6-45](#)

Server Functions:

[“MedTCFrom” on page 5-59](#)

medLocGetLocation

Returns the *Location* element of a MedLoc value. For information on this data type, see “[MedLoc Data Type](#)” on page 3-7.

Syntax

```
mi_string *medLocGetLocation(MedLoc *media_object)
```

Argument	Purpose	Restrictions
*media_object	Specifies the large media object for which the caller wants the <i>Location</i> element.	N.A.

Returns

Returns a pointer to an `mi_string` value that identifies a large media object’s location on a particular storage server. The syntax of this string is storage-server-specific.

On error, returns a null pointer.

Example

```
MedLoc *mlMedLoc;
char *szTypePtr;
char *szLocationPtr;
mi_integer iStorageID;

/*
 * One will usually retrieve a MedLoc from the database
 * After running an mi_exec in MI_QUERY_BINARY mode
 * you get a pointer to the MedLoc
 *
 * The typical way of creating this value would be in the form
 *
 * mi_exec(dbConnection,
 * "select ClientAccess(Location,1,0,'VIDEO') from
 * VidPVideo",
 * MI_QUERY_BINARY);
```

```
* mi_get_result(dbConnection);
* mi_next_row(dbConnection, &error);
* mi_value(row, i, &colval, &collen);
* m1MedLoc = (MedLoc *)colval;
*/

szTypePtr = medLocGetType(m1MedLoc);
szLocationPtr = medLocGetLocation(m1MedLoc);
iStorageID = medLocGetStorage(m1MedLoc);

/* Now look at the pieces of this MedLoc*/
fprintf (stdout,
    "MedLoc results:\n Type %s\n Location %s\n StorageID
%d\n",
    szTypePtr, szLocationPtr, iStorageID);

/* Output is:
* MedLoc results: Type VSI_OTHER Location Beta SP Tape 87723-
1
* StorageID 274961
*/
```

See Also

Client API Functions:

[“medLocGetStorageID” on page 6-28](#)

[“medLocGetType” on page 6-30](#)

Server Functions:

[“Location” on page 5-48](#)

medLocGetStorageID

Returns the *StorageID* element of a MedLoc value (see [“MedLoc Data Type” on page 3-7](#)).

Syntax

```
mi_integer medLocGetStorageID(MedLoc *media_object)
```

Argument	Purpose	Restrictions
*media_object	Specifies the large media object for which the caller wants the <i>StorageID</i> element.	N.A.

Returns

Returns an *mi_integer* value that specifies the storage ID number. On error, returns a null pointer.

Example

```
MedLoc *m1MedLoc;
char *szTypePtr;
char *szLocationPtr;
mi_integer iStorageID;

/*
 * One will usually retrieve a MedLoc from the database
 * After running an mi_exec in MI_QUERY_BINARY mode
 * you get a pointer to the MedLoc
 *
 * The typical way of creating this value would be in the form
 *
 * mi_exec(dbConnection,
 *         "select ClientAccess(Location,1,0,'VIDEO') from
 *         VidPVideo",
 *         MI_QUERY_BINARY);
 * mi_get_result(dbConnection);
 * mi_next_row(dbConnection, &error);
 * mi_value(row, i, &colval, &collen);
 * m1MedLoc = (MedLoc *)colval;
```



```
*/

szTypePtr = medLocGetType(mlMedLoc);
szLocationPtr = medLocGetLocation(mlMedLoc);
iStorageID = medLocGetStorage(mlMedLoc);

/* Now look at the pieces of this MedLoc*/
fprintf (stdout,
        "MedLoc results:\n  Type %s\n  Location %s\n  StorageID\n%d\n",
        szTypePtr, szLocationPtr, iStorageID);

/* Output is:
* MedLoc results: Type VSI_WEB
*      Location http://www.informix.com/video_sample/file.mov
*      StorageID 414770
*/
```

See Also

Client API Functions:

[“medLocGetLocation” on page 6-26](#)

[“medLocGetType” on page 6-30](#)

Server Functions:

[“StorageID” on page 5-106](#)

medLocGetType

Returns the *Type* element (the storage server type) of a MedLoc value. For information on this data type, see “[MedLoc Data Type](#)” on page 3-7.

Syntax

```
mi_string *medLocGetType(MedLoc *media_object)
```

Argument	Purpose	Restrictions
<i>*media_object</i>	Specifies the large media object for which the caller wants the <i>Type</i> element.	N.A.

Returns

Returns a pointer to an *mi_string* value that specifies the *Type* element. On error, returns a null pointer.

Example

```
MedLoc *m1MedLoc;
char *szTypePtr;
char *szLocationPtr;
mi_integer iStorageID;

/*
 * One will usually retrieve a MedLoc from the database
 * After running an mi_exec in MI_QUERY_BINARY mode
 * you get a pointer to the MedLoc
 *
 * The typical way of creating this value would be in the form
 *
 * mi_exec(dbConnection,
 *         "select ClientAccess(Location,1,0,'VIDEO') from
VidPVideo",
 *         MI_QUERY_BINARY);
 * mi_get_resultt(dbConnection);
 * mi_next_row(dbConnection, &error);
 * mi_value(row, i, &colval, &collen);
 * m1MedLoc = (MedLoc *)colval;
```

```
*/

szTypePtr = medLocGetType(m1MedLoc);
szLocationPtr = medLocGetLocation(m1MedLoc);
iStorageID = medLocGetStorage(m1MedLoc);

/* Now look at the pieces of this MedLoc*/
fprintf (stdout,
        "MedLoc results:\n  Type %s\n  Location %s\n  StorageID\n%d\n",
        szTypePtr, szLocationPtr, iStorageID);

/* Output is:
* MedLoc results: Type VSI_FILE
*      Location G:\dir1\dir2\dir3\dir4\filename.avi
*      StorageID 344865
*/
```

See Also

Client API Functions:

[“medLocGetLocation” on page 6-26](#)

[“medLocGetStorageID” on page 6-28](#)

Server Functions:

[“Type \(MedLoc\)” on page 5-110](#)

medPtrGetNative

Returns the *Native* element of a MedPtr value. For information on this data type, see “[MedPtr Data Type](#)” on page 2-3.

Syntax

```
mi_string *medPtrGetNative(MedPtr *media_point)
```

Argument	Purpose	Restrictions
*media_point	Specifies the media point for which the caller wants the <i>Native</i> element.	N.A.

Returns

Returns a pointer to an mi_string value that specifies the *Native* element. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8InputTime;
mi_int8 *i8TheResultTime;
MedPtr *mpMedPtr;

/* Create a MedPtr */
/* There are 3.6 trillion nanoseconds in an hour */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8InputTime);

/* Create the new MedPtr */
mpMedPtr = medTimeNativeToMedPtr(&i8InputTime,
szNativeString);

/* Look at the results */
ifx_int8toasc(i8TheResultTime, szNumString, STRING_SIZE);
```

```
fprintf (stdout, "medPtrMinus result: Time %s\n",
szNumString);

/* Output is:
 * medPtrMinus result: Time 3600000000000, Native
 */

/* Clean up after ourselves */
mi_free(mpMedPtr);
mi_free(i8TheResultTime);
```

See Also

Client API Functions:

[“medPtrGetTime” on page 6-34](#)

Server Functions:

[“Native” on page 5-71](#)

medPtrGetTime

Returns the *Time* element, in nanoseconds, of a MedPtr value. For information on this data type, see [“MedPtr Data Type” on page 2-3](#).

Syntax

```
mi_int8 *medPtrGetTime(MedPtr *media_point)
```

Argument	Purpose	Restrictions
<i>*media_point</i>	Specifies the media point for which the caller wants the <i>Time</i> element.	N.A.

Returns

Returns a pointer to an mi_int8 value that specifies the *Time* element in nanoseconds. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8InputTime;
mi_int8 *i8TheResultTime;
MedPtr *mpMedPtr;

/* Create a MedPtr */
/* There are 3.6 trillion nanoseconds in an hour */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8InputTime);
mpMedPtr = medTimeNativeToMedPtr(&i8InputTime,
szNativeString);

/* Get the time from the MedPtr structure */
i8TheResultTime = medPtrGetTime(mpMedPtr);

/* Look at the results */
ifx_int8toasc(i8TheResultTime, szNumString, STRING_SIZE);
```

```
fprintf (stdout, "medPtrGetTime result: Time %s\n",
szNumString);

/* Output is:
 * medPtrGetTime result: Time 3600000000000
 */

/* Clean up after ourselves */
mi_free(mpMedPtr);
mi_free(i8TheResultTime);
```

See Also

Client API Functions:

[“medPtrGetNative” on page 6-32](#)

Server Functions:

[“Time” on page 5-108](#)

medPtrMinus

Takes two MedPtr values (see “[MedPtr Data Type](#)” on page 2-3) and subtracts the second one from the first.

Syntax

```
MedPtr *medPtrMinus(MedPtr *media_point1, MedPtr *media_point2)
```

Argument	Purpose	Restrictions
<i>*media_point1, *media_point2</i>	Specifies which media point is subtracted (<i>*media_point2</i>) from which (<i>*media_point1</i>).	N.A.

Returns

Returns a pointer to a MedPtr value. On error, returns a null pointer.

Example

```
char szNativeString1[STRING_SIZE],
szNativeString2[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8FirstTime, i8SecondTime;
MedPtr *mpMedPtr1, *mpMedPtr2, *mpResultPtr;

/* Create a first MedPtr */
/* There are 3.6 trillion nanoseconds in an hour */
strcpy(szNativeString1, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8FirstTime);
mpMedPtr1 = medTimeNativeToMedPtr(&i8FirstTime,
szNativeString1);

/* Create the second MedPtr */
/* There are 7.2 trillion nanoseconds in 2 hours */
strcpy(szNativeString2, "00:02:00:00");
strcpy(szNumString, "7200000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8SecondTime);
```



```
mpMedPtr2 = medTimeNativeToMedPtr(&i8SecondTime,
szNativeString2);

/* Do the Math */
mpResultPtr = medPtrMinus(mpMedPtr2, mpMedPtr1);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout,"medPtrMinus result: Time %s, Native %s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medPtrMinus result: Time 3600000000000, Native
 */

/* Clean up after ourselves */
mi_free(mpMedPtr1);
mi_free(mpMedPtr2);
mi_free(mpResultPtr);
```

See Also

Client API Functions:

[“medPtrPlus” on page 6-38](#)

Server Functions:

[“Minus” on page 5-64](#)

medPtrPlus

Takes two MedPtr values (see “MedPtr Data Type” on page 2-3) and adds them.

Syntax

```
MedPtr *medPtrPlus(MedPtr *media_point1, MedPtr *media_point2)
```

Argument	Purpose	Restrictions
<i>*media_point1, *media_point2</i>	Specifies which media points are added.	N.A.

Returns

Returns a pointer to a MedPtr value. On error, returns a null pointer.

Example

```
char szNativeString1[STRING_SIZE],
szNativeString2[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8FirstTime, i8SecondTime;
MedPtr *mpMedPtr1, *mpMedPtr2, *mpResultPtr;

/* First MedPtr */
/* There are 3.6 trillion nanoseconds in an hour */
strcpy(szNativeString1, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8FirstTime);
mpMedPtr1 = medTimeNativeToMedPtr(&i8FirstTime,
szNativeString1);

/* Second MedPtr */
/* There are 7.2 trillion nanoseconds in 2 hours */
strcpy(szNativeString2, "00:02:00:00");
strcpy(szNumString, "7200000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8SecondTime);
mpMedPtr2 = medTimeNativeToMedPtr(&i8SecondTime,
szNativeString2);
```

```

/* Do the Math */
mpResultPtr = medPtrPlus(mpMedPtr1, mpMedPtr2);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout, "medPtrAdd result: Time %s, Native %s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medPtrAdd result: Time 10800000000000, Native
 */

/* Clean up after ourselves */
mi_free(mpMedPtr1);
mi_free(mpMedPtr2);
mi_free(mpResultPtr);

```

See Also

Client API Functions:

[“medPtrMinus” on page 6-36](#)

Server Functions:

[“Plus” on page 5-89](#)



medPtrToFrameNumber

Converts a MedPtr value (see “[MedPtr Data Type](#)” on page 2-3) and a frames-per-second rate into an integer value that represents an absolute frame number.

Tip: The first frame number in a piece of media is 0. Thus, to calculate the number of frames, you must add 1 to the result of this function.

Syntax

```
mi_integer medPtrToFrameNumber(MedPtr *media_point,  
                                mi_double_precision *fps)
```

Argument	Purpose	Restrictions
<i>*media_point</i>	Specifies the media point used (with <i>*fps</i> , next row) in calculating the frame number.	N.A.
<i>*fps</i>	Specifies the frames-per-second rate used (with <i>*media_point</i> , previous row) in calculating the frame number.	Must be a positive number.

Returns

Returns an `mi_integer` value that specifies the frame number. On error, returns a null pointer.

Example

```

MedPtr *mpResultPtr;
mi_integer iFrameNumberResult;
mi_double_precision theFrameRate;
char szNumString[STRING_SIZE];

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mpResultPtr = medFrameNumberToMedPtr(2000, &theFrameRate);

/* Look at the results */
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout, "medFrameNumberToMedPtr result: Time %s,
Native %s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medFrameNumberToMedPtr result: Time 66733400067, Native
2000
 */

/*
 * and now let's get the frame number from the MediaPtr
 * This is the type of thing that occurs when translating from
an
 * original tape in one format and copy in a different format
 */

theFrameRate = 25.0;
iFrameNumberResult = medPtrToFrameNumber(mpResultPtr,
&theFrameRate);

/* Look at the results */
fprintf (stdout,
        "medPtrToFrameNumber result: Frame 2000 at 29.97 fps is %d
at 25.0 fps\n",
        iFrameNumberResult);

/* Output is:
 * medPtrToFrameNumber result: Frame 2000 at 29.97 fps is 1668
at 25.0 fps
 */

/* Clean up after ourselves */
mi_free(mpResultPtr);

```

See Also

Client API Functions:

[“medFrameNumberToMedPtr” on page 6-22](#)

[“medTCToFrameNumber” on page 6-66](#)

Server Functions:

[“VidFrameNumberFrom” on page 5-114](#)

medPtrToMedChunk

Converts a single MedPtr value into a single-point MedChunk value (the *StartP* and *EndP* elements are the same). For information on the MedPtr data type, see “[MedPtr Data Type](#)” on page 2-3; for information on the MedChunk data type, see “[MedChunk Data Type](#)” on page 2-12.

Syntax

```
MedChunk *medPtrToMedChunk(MedPtr *media_point)
```

Argument	Purpose	Restrictions
<i>*media_point</i>	Specifies the media point to convert to a single-point media chunk.	N.A.

Returns

Returns a pointer to a MedChunk value. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8Time;
mi_int8 *i8TheChunkDuration;
MedPtr *mpMedPtr;
MedChunk *mcMedChunk;

/* The MedChunk */
/* This chunk has both start and end points at 1 hour */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtr = medTimeNativeToMedPtr(&i8Time, szNativeString);

/* Notice we only use one MedPtr to define this MedChunk */
mcMedChunk = medPtrToMedChunk(mpMedPtr);

/* Which means chunk size will be 0 */
```

```
i8TheChunkDuration = medChunkGetSize(mcMedChunk);

/* Look at the results, it should be 0 nanoseconds */
ifx_int8toasc(i8TheChunkDuration, szNumString, STRING_SIZE);
fprintf (stdout, "medChunkGetSize result: Time %s\n",
szNumString);

/* Output is:
 * medChunkGetSize result: Time 0
 */

/* Clean up after ourselves */
mi_free(mpMedPtr);
mi_free(mcMedChunk);
```

See Also

Client API Functions:

[“medPtrsToMedChunk” on page 6-47](#)

Server Functions:

[“MedChunkFrom” on page 5-50](#)

medPtrToMedTC

Converts a MedPtr value and a SMPTE frames-per-second rate into a MedTC value. For information on the MedPtr data type, see “[MedPtr Data Type](#)” on page 2-3; for information on the MedTC data type, see “[MedTC Data Type](#)” on page 2-7.

Syntax

```
MedTC *medPtrToMedTC(MedPtr *media_point,  
mi_double_precision *fps)
```

Argument	Purpose	Restrictions
<i>*media_point</i>	Specifies the media point used (with <i>*fps</i> , next row) in calculating the media timecode. The <i>Native</i> element of the MedPtr is not interpreted.	N.A.
<i>*fps</i>	Specifies the frames-per-second rate used (with <i>*media_point</i> , previous row) in calculating the media timecode.	Must be a valid SMPTE frame rate: 24.0, 25.0, 29.97, or 30.0.

Returns

Returns a pointer to a MedTC value. On error, returns a null pointer.

Example

```

MedPtr *mpResultPtr;
MedTC *mtResultTC;
mi_double_precision theFrameRate;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mpResultPtr = medFrameNumberToMedPtr(2000, &theFrameRate);

/* ... and now we get a drop frame Time Code for frame 2000 */
mtResultTC = medPtrToMedTC(mpResultPtr, &theFrameRate);

/* Look at the results */
fprintf (stdout,
"medPtrToMedTC result: Frame 2000 at 29.97 fps is
%.2d:%.2d:%.2d:%.2d-D\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);

/* Output is:
 * medPtrToMedTC result: Frame 2000 at 29.97 fps is
00:01:06:22-D
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
mi_free(mpResultPtr);

```

See Also

Client API Functions:

[“medFrameNumberToMedTC” on page 6-24](#)

[“medTCToMedPtr” on page 6-68](#)

Server Functions:

[“MedTCFrom” on page 5-59](#)

medPtrsToMedChunk

Converts two MedPtr values (a starting media point and an ending media point) into a MedChunk value. For information on the MedChunk data type, see “MedChunk Data Type” on page 2-12; for information on the MedPtr data type, see “MedPtr Data Type” on page 2-3.

Syntax

```
MedChunk *medPtrsToMedChunk(MedPtr *media_point1,
                             MedPtr *media_point2)
```

Argument	Purpose	Restrictions
*media_point1, *media_point2	Specifies the <i>StartP</i> element (<i>*media_point1</i>) and the <i>EndP</i> element (<i>*media_point2</i>) of the media chunk to construct.	N.A.

Returns

Returns a pointer to a MedChunk value. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8Time;
mi_int8 *i8TheChunkDuration;
MedPtr *mpMedPtrStart, *mpMedPtrEnd;
MedChunk *mcMedChunk;

/* The MedChunk */
/* This chunk goes from the 1 hour to the 3 hour mark on the
video */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrStart = medTimeNativeToMedPtr(&i8Time,
szNativeString);
```

```
strcpy(szNativeString, "00:03:00:00");
strcpy(szNumString, "10800000000000");
ifx_int8cvasc(szNumString, strlen(szNumString), &i8Time);
mpMedPtrEnd = medTimeNativeToMedPtr(&i8Time,
szNativeString);
mcMedChunk = medPtrsToMedChunk(mpMedPtrStart, mpMedPtrEnd);

i8TheChunkDuration = medChunkGetSize(mcMedChunk);

/* Look at the results, it should be 2 hours */
ifx_int8toasc(i8TheChunkDuration, szNumString, STRING_SIZE);
fprintf (stdout, "medChunkGetSize result: Time %s\n",
szNumString);

/* Output is:
 * medChunkGetSize result: Time 7200000000000
 */

/* Clean up after ourselves */
mi_free(mpMedPtrStart);
mi_free(mpMedPtrEnd);
mi_free(mcMedChunk);
```

See Also

Client API Functions:

[“medPtrToMedChunk” on page 6-43](#)

Server Functions:

[“MedChunkFrom” on page 5-50](#)

medTCGetFPS

Returns the frames-per-second rate of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_double_precision *medTCGetFPS(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the frames-per-second rate.	N.A.

Returns

Returns a pointer to an `mi_double_precision` value that identifies the frames-per-second rate (and thus the video standard and valid frame numbers) of this timecode, as follows.

Fps Value	Video Standard	Valid Frame Numbers
24	Film	00 through 23.
25	PAL	00 through 24.
29.97	Drop-frame NTSC	02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.
30	NTSC	00 through 29.

On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_double_precision *theFrameResult;

/* Start with a film frame rate */
theFrameRate = 24.0;
mtResultTC = medFrameNumberToMedTC(410000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 410000 at 24.0 fps is %.2d:%.2d:%.2d:%.2d-F\n",
        mtResultTC->hours,
        mtResultTC->minutes,
        mtResultTC->seconds,
        mtResultTC->frames);

/* Output is:
 * medFrameNumberToMedTC result: Frame 410000 at 24.0 fps is
 * 04:44:43:08-F
 */

theFrameResult = medTCGetFPS(mtResultTC);

/* Display the frame rate */
fprintf (stdout, "medTCGetFPS result:");
fprintf (stdout,
    " The film frame rate is %f", *theFrameResult);

/* Output is:
 * medTCGetFPS result: The film frame rate is 24.000000
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medTCGetFrame” on page 6-52](#)

[“medTCGetHour” on page 6-54](#)

[“medTCGetMinute” on page 6-56](#)

[“medTCGetSecond” on page 6-58](#)

[“medTCGetType” on page 6-60](#)

Server Functions:

[“Fps” on page 5-32](#)

medTCGetFrame

Returns the *FF* element (the frame number) of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_integer medTCGetFrame(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the <i>FF</i> element.	N.A.

Returns

Returns an `mi_integer` value that specifies the frame number. (This number is determined by the video standard in use; see [“MedTC Data Type” on page 2-7](#).)

On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_integer iHour, iMinute, iSecond, iFrame;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(320000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 320000 at 29.97 fps is %.2d:%.2d:%.2d:%.2d-D\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);
```



```
/* Output is:
 * medFrameNumberToMedTC result: Frame 320000 at 29.97 fps is
 * 02:57:57:10-D
 */

iHour = medTCGetHour(mtResultTC);
iMinute = medTCGetMinute(mtResultTC);
iSecond = medTCGetSecond(mtResultTC);
iFrame = medTCGetFrame(mtResultTC);

/* ... now look at the break down */
fprintf (stdout, "Break down of results:");
fprintf (stdout,
         "Hours (%d), Minutes(%d), Seconds(%d), Frames(%d)\n",
         iHour, iMinute, iSecond, iFrame);

/* Output is:
 * Break down of results:Hours (2), Minutes(57), Seconds(57),
 * Frames(10)
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

[“medTCGetHour” on page 6-54](#)

[“medTCGetMinute” on page 6-56](#)

[“medTCGetSecond” on page 6-58](#)

[“medTCGetType” on page 6-60](#)

Server Functions:

[“Frame” on page 5-34](#)

medTCGetHour

Returns the *HH* element (the hours) of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_integer medTCGetHour(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the <i>HH</i> element.	N.A.

Returns

Returns an `mi_integer` value that specifies the hours of the timecode. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_integer iHour, iMinute, iSecond, iFrame;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(320000, &theFrameRate);

/* Look at the results */
fprintf(stdout, "medFrameNumberToMedTC result:");
fprintf(stdout,
    " Frame 320000 at 29.97 fps is %.2d:%.2d:%.2d:%.2d-D\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);

/* Output is:
 * medFrameNumberToMedTC result: Frame 320000 at 29.97 fps is
 * 02:57:57:10-D
 */
```

```

iHour = medTCGetHour(mtResultTC);
iMinute = medTCGetMinute(mtResultTC);
iSecond = medTCGetSecond(mtResultTC);
iFrame = medTCGetFrame(mtResultTC);

/* ... now look at the break down */
fprintf (stdout, "Break down of results:");
fprintf (stdout,
    "Hours (%d), Minutes(%d), Seconds(%d), Frames(%d)\n",
        iHour, iMinute, iSecond, iFrame);

/* Output is:
 * Break down of results:Hours (2), Minutes(57), Seconds(57),
 * Frames(10)
 */

/* Clean up after ourselves */
mi_free(mtResultTC);

```

See Also

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

[“medTCGetFrame” on page 6-52](#)

[“medTCGetMinute” on page 6-56](#)

[“medTCGetSecond” on page 6-58](#)

[“medTCGetType” on page 6-60](#)

Server Functions:

[“Hour” on page 5-40](#)

medTCGetMinute

Returns the *MM* element (the minutes) of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_integer medTCGetMinute(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the <i>MM</i> element.	N.A.

Returns

Returns an `mi_integer` value that specifies the minutes of the timecode. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_integer iHour, iMinute, iSecond, iFrame;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(320000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 320000 at 29.97 fps is %.2d:%.2d:%.2d:%.2d-D\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);

/* Output is:
 * medFrameNumberToMedTC result: Frame 320000 at 29.97 fps is
 02:57:57:10-D
 */
```

```
iHour = medTCGetHour(mtResultTC);
iMinute = medTCGetMinute(mtResultTC);
iSecond = medTCGetSecond(mtResultTC);
iFrame = medTCGetFrame(mtResultTC);

/* ... now look at the break down */
fprintf (stdout, "Break down of results:");
fprintf (stdout,
    "Hours (%d), Minutes(%d), Seconds(%d), Frames(%d)\n",
        iHour, iMinute, iSecond, iFrame);

/* Output is:
 * Break down of results:Hours (2), Minutes(57), Seconds(57),
 * Frames(10)
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

[“medTCGetFrame” on page 6-52](#)

[“medTCGetHour” on page 6-54](#)

[“medTCGetSecond” on page 6-58](#)

[“medTCGetType” on page 6-60](#)

Server Functions:

[“Minute” on page 5-69](#)

medTCGetSecond

Returns the SS element (the seconds) of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_integer medTCGetSecond(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the SS element.	N.A.

Returns

Returns an `mi_integer` value that specifies the seconds of the timecode. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_integer iHour, iMinute, iSecond, iFrame;

/* Start with a SMPTE drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(320000, &theFrameRate);

/* Look at the results */
fprintf(stdout, "medFrameNumberToMedTC result:");
fprintf(stdout,
    " Frame 320000 at 29.97 fps is %.2d:%.2d:%.2d:%.2d-D\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);

/* Output is:
 * medFrameNumberToMedTC result: Frame 320000 at 29.97 fps is
 * 02:57:57:10-D
 */
```

```
iHour = medTCGetHour(mtResultTC);
iMinute = medTCGetMinute(mtResultTC);
iSecond = medTCGetSecond(mtResultTC);
iFrame = medTCGetFrame(mtResultTC);

/* ... now look at the break down */
fprintf (stdout, "Break down of results:");
fprintf (stdout,
    "Hours (%d), Minutes(%d), Seconds(%d), Frames(%d)\n",
        iHour, iMinute, iSecond, iFrame);

/* Output is:
 * Break down of results:Hours (2), Minutes(57), Seconds(57),
 * Frames(10)
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

[“medTCGetFrame” on page 6-52](#)

[“medTCGetHour” on page 6-54](#)

[“medTCGetMinute” on page 6-56](#)

[“medTCGetType” on page 6-60](#)

Server Functions:

[“Second” on page 5-92](#)

medTCGetType

Returns the *-Standard_flag* element (the video standard type) of a MedTC value. For information on this data type, see [“MedTC Data Type” on page 2-7](#).

Syntax

```
mi_lvarchar *medTCGetType(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode for which the caller wants the <i>-Standard_flag</i> element.	N.A.

Returns

Returns a pointer to an *mi_lvarchar* value that specifies the video standard (and thus the valid frame numbers) of this timecode, as follows.

Optional Flag	Video Standard	Valid Frame Numbers
D	Drop-frame NTSC	02 through 29. For minutes divisible by 10, valid frame numbers are 00 through 29.
F	Film	00 through 23.
N	NTSC	00 through 29.
P	PAL	00 through 24.

On error, returns a null pointer.

Example

```

MedTC *mtResultTC;
char *szTimeCodeType;
mi_double_precision theFrameRate;

/* Start with an NTSC drop frame rate */
theFrameRate = 29.97;
mtResultTC = medFrameNumberToMedTC(300000, &theFrameRate);

szTimeCodeType = medTCGetType(mtResultTC);

/* Look at the results */
fprintf (stdout, "medTCGetType result: Type found '%s'\n",
szTimeCodeType);

/* Output is:
 * medTCGetType result: Type found 'D'
 */

/* Clean up after ourselves */
mi_free(mtResultTC);

```

See Also

Client API Functions:

[“medTCGetFPS” on page 6-49](#)

[“medTCGetFrame” on page 6-52](#)

[“medTCGetHour” on page 6-54](#)

[“medTCGetMinute” on page 6-56](#)

[“medTCGetSecond” on page 6-58](#)

Server Functions:

[“Type \(MedTC\)” on page 5-112](#)

medTCMinus

Takes two MedTC values (see “[MedTC Data Type](#)” on page 2-7) and subtracts the second one from the first.

Syntax

```
MedTC *medTCMinus(MedTC *timecode1, MedTC *timecode2)
```

Argument	Purpose	Restrictions
<i>*timecode1, *timecode2</i>	Specifies which timecode is subtracted (<i>*timecode2</i>) from which (<i>*timecode1</i>).	N.A.

Returns

Returns a pointer to an MedTC value. On error, returns a null pointer.

Example

```
MedPtr *mpResultPtr;
MedTC *mtTimeCode1, *mtTimeCode2, *mtResultTC;
mi_double_precision theFrameRate;

/* Create the first Time Code */
/* Start with a PAL or SECAM frame rate */
theFrameRate = 25.0;
mpResultPtr = medFrameNumberToMedPtr(10000, &theFrameRate);

/* ... and now we get a Time Code for frame 3000 */
mtTimeCode1 = medPtrToMedTC(mpResultPtr, &theFrameRate);
mi_free(mpResultPtr); /* Free this memory before reusing the
pointer */

/* Now Create the second Time Code */
/* Note: frame rate must be the same */
mpResultPtr = medFrameNumberToMedPtr(3500, &theFrameRate);

/* ... and now we get a Time Code for frame 6500 */
mtTimeCode2 = medPtrToMedTC(mpResultPtr, &theFrameRate);
```

```

/* Do the math */
mtResulttTC = medTCMinus(mtTimeCode1, mtTimeCode2);

/* Look at the results */
fprintf (stdout,
"medTCMinus result: Frame 6500 at 25.0 fps is
%.2d:%.2d:%.2d:%.2d-P\n",
    mtResulttTC->hours,
    mtResulttTC->minutes,
    mtResulttTC->seconds,
    mtResulttTC->frames);

/* Output is:
 * medTCMinus result: Frame 6500 at 25.0 fps is 00:04:20:00-P
 */

/* Clean up after ourselves */
mi_free(mtTimeCode1);
mi_free(mtTimeCode2);
mi_free(mtResulttTC);
mi_free(mpResulttPtr);

```

See Also

Client API Functions:

[“medTCPlus” on page 6-64](#)

Server Functions:

[“Minus” on page 5-64](#)

medTCPlus

Takes two MedTC values (see “MedTC Data Type” on page 2-7) and adds them.

Syntax

```
MedTC *medTCPlus(MedTC *timecode1, MedTC *timecode2)
```

Argument	Purpose	Restrictions
<i>*timecode1, *timecode2</i>	Specifies which timecodes to add.	N.A.

Returns

Returns a pointer to a MedTC value. On error, returns a null pointer.

Example

```
MedPtr *mpResultPtr;
MedTC *mtTimeCode1, *mtTimeCode2, *mtResultTC;
mi_double_precision theFrameRate;

/* Create the first Time Code */
/* Start with a standard film frame rate */
theFrameRate = 24.0;
mpResultPtr = medFrameNumberToMedPtr(3000, &theFrameRate);

/* ... and now we get a Time Code for frame 3000 */
mtTimeCode1 = medPtrToMedTC(mpResultPtr, &theFrameRate);
mi_free(mpResultPtr); /* Free this memory before reusing the
pointer */

/* Now Create the second Time Code */
/* Note: frame rate must be the same */
mpResultPtr = medFrameNumberToMedPtr(5500, &theFrameRate);

/* ... and now we get a Time Code for frame 5500 */
mtTimeCode2 = medPtrToMedTC(mpResultPtr, &theFrameRate);

/* Do the math */
mtResultTC = medTCPlus(mtTimeCode1, mtTimeCode2);
```

```

/* Look at the results */
fprintf (stdout,
    "medTCPlus result: Frame 8500 at 24.0 fps is
    %.2d:%.2d:%.2d:%.2d-F\n",
        mtResultTC->hours,
        mtResultTC->minutes,
        mtResultTC->seconds,
        mtResultTC->frames);

/* Output is:
 * medTCPlus result: Frame 8500 at 24.0 fps is 00:05:54:04-F
 */

/* Clean up after ourselves */
mi_free(mtTimeCode1);
mi_free(mtTimeCode2);
mi_free(mtResultTC);
mi_free(mpResultPtr);

```

See Also

Client API Functions:

[“medTCMinus” on page 6-62](#)

Server Functions:

[“Plus” on page 5-89](#)



medTCToFrameNumber

Converts a MedTC value (see [“MedTC Data Type” on page 2-7](#)) into an integer value representing an absolute frame number.

Tip: The first frame number in a piece of media is 0. Thus, to calculate the number of frames, you must add 1 to the result of this function.

Syntax

```
mi_integer medTCToFrameNumber(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode to convert to a frame number.	N.A.

Returns

Returns an `mi_integer` value that specifies the frame number. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
mi_double_precision theFrameRate;
mi_integer iFrameNumber;

/* Start with an NTSC frame rate */
theFrameRate = 30.0;
mtResultTC = medFrameNumberToMedTC(15000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 15000 at 30.0 fps is %.2d:%.2d:%.2d:%.2d-N\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);
```

```
/* Output is:
 * medFrameNumberToMedTC result: Frame 15000 at 30.0 fps is
 * 00:08:20:00-N
 */

/* Now let's convert it back */
iFrameNumber = medTCToFrameNumber(mtResultTC);
fprintf (stdout, "medTCToFrameNumber result:Frame number %d
found\n",
        iFrameNumber);

/* Output is:
 * medTCToFrameNumber result: Frame number 15000 found
 */

/* Clean up after ourselves */
mi_free(mtResultTC);
```

See Also

Client API Functions:

[“medFrameNumberToMedTC” on page 6-24](#)

Server Functions:

[“VidFrameNumberFrom” on page 5-114](#)

medTCtoMedPtr

Converts a MedTC value into a MedPtr value. For information on the MedTC data type, see “[MedTC Data Type](#)” on page 2-7; for information on the MedPtr data type, see “[MedPtr Data Type](#)” on page 2-3.

Syntax

```
MedPtr *medTCtoMedPtr(MedTC *timecode)
```

Argument	Purpose	Restrictions
<i>*timecode</i>	Specifies the timecode to convert. The single input is first converted into the <i>Time</i> element, and then the <i>same input</i> is used for the <i>Native</i> element.	N.A.

Returns

Returns a pointer to a MedPtr value. On error, returns a null pointer.

Example

```
MedTC *mtResultTC;
MedPtr *mpResultPtr;
mi_double_precision theFrameRate;
char szNumString[STRING_SIZE];

/* Start with a PAL or SECAM frame rate */
theFrameRate = 25.0;
mtResultTC = medFrameNumberToMedTC(280000, &theFrameRate);

/* Look at the results */
fprintf (stdout, "medFrameNumberToMedTC result:");
fprintf (stdout,
    " Frame 280000 at 25.0 fps is %.2d:%.2d:%.2d:%.2d-P\n",
    mtResultTC->hours,
    mtResultTC->minutes,
    mtResultTC->seconds,
    mtResultTC->frames);
```



```

/* Output is:
 * medFrameNumberToMedTC result: Frame 280000 at 25.0 fps is
03:06:40:00-P
 */

mpResultPtr = medTCToMedPtr(mtResultTC);

/* Look at the results of converting to MedPtr*/
ifx_int8toasc(&(mpResultPtr->Time), szNumString,
STRING_SIZE);
fprintf (stdout, "medTCToMedPtr result:\nTime %s, Native
%s\n",
        szNumString, mpResultPtr->Native);

/* Output is:
 * medTCToMedPtr result: Time 11200000000000, Native
03:06:40:00-P
 */

/* Clean up after ourselves */
mi_free(mpResultPtr);
mi_free(mtResultTC);

```

See Also

Client API Functions:

[“medFrameNumberToMedPtr” on page 6-22](#)

[“medPtrToMedTC” on page 6-45](#)

[“medTimeNativeToMedPtr” on page 6-70](#)

Server Functions:

[“MedPtrFrom” on page 5-53](#)

medTimeNativeToMedPtr

Converts a *Time* integer and a *Native* string into a MedPtr value. For information on the MedPtr data type, see [“MedPtr Data Type” on page 2-3](#).

Syntax

```
MedPtr *medTimeNativeToMedPtr(mi_int8 *time,
                               mi_string *native)
```

Argument	Purpose	Restrictions
<i>*time</i>	Specifies the value to use as the <i>Time</i> element.	Must be a positive number.
<i>*native</i>	Specifies the variable-length character string to use as the <i>Native</i> element.	N.A.

Returns

Returns a pointer to a MedPtr value. On error, returns a null pointer.

Example

```
char szNativeString[STRING_SIZE];
char szNumString[STRING_SIZE];
mi_int8 i8InputTime;
MedPtr *mpMedPtr;

/* Create a MedPtr */
/* There are 3.6 trillion nanoseconds in an hour */
strcpy(szNativeString, "00:01:00:00");
strcpy(szNumString, "3600000000000");
ifx_int8cvasc(szNumString, strlen(szNumString),
&i8InputTime);

/* Which gives us our MedPtr*/
mpMedPtr = medTimeNativeToMedPtr(&i8InputTime,
szNativeString);

/* Clean up after ourselves */
mi_free(mpMedPtr);
```

See Also

Client API Functions:

[“medFrameNumberToMedPtr” on page 6-22](#)

[“medTCToMedPtr” on page 6-68](#)

Server Functions:

[“MedPtrFrom” on page 5-53](#)

Examples

In This Chapter	7-3
Preparing to Run the Examples	7-4
Registering the Video Storage Server Component	7-4
Inserting Physical Attributes	7-5
Copy 1: BetaSP Tape	7-6
Copy 2: MPEG File	7-6
Inserting Abstract Descriptions	7-8
Inserting Explicit Values.	7-9
Inserting SELECT Statement Results	7-11
Issuing Queries	7-12
Selecting the MediaID Value of a Media Instance	7-12
Selecting Media Chunks in a Single Strata	7-12
Selecting Media Chunks that Overlap with a Strata	7-13
Selecting a Newly Constructed Media Chunk	7-14
Requesting Access to a Segment	7-15
Tracking Nondigital Multimedia Assets.	7-15

In This Chapter

This chapter provides a multipart example that shows how the fictitious Sample Company incorporates the Informix Video Foundation DataBlade module into its media management system. Specifically, the example shows how Sample Company manages, queries, and accesses one of its multimedia assets, the classic movie *Metropolis*.

This chapter first describes how to prepare to run the examples yourself; then it shows how Sample Company performs the following multimedia-management tasks using the Informix Video Foundation DataBlade module:

- Registering Sample Company's video storage server component
- Registering the physical attributes of *Metropolis*, its media instances, and the files that constitute those instances into the inventory tables
- Inserting abstract descriptions of *Metropolis* into the subject matter tables
- Issuing queries about *Metropolis* based on the physical and abstract metadata entered into the video database schema tables
- Requesting playback access to *Metropolis*
- Creating a company-specific schema table for Sample Company's nondigital multimedia assets and issuing a query about the nondigital copy of a media chunk

Tip: The concepts, schema tables, and functions mentioned in this chapter are defined in other chapters of this user's guide. For information, see the index.



Preparing to Run the Examples

The examples shown in this chapter are provided with the Informix Video Foundation DataBlade module so that you can try them out for yourself. The code for the examples is in the **dcexmpls.sql** file of this DataBlade module's **\$INFORMIXDIR/extend** directory.

To prepare to run the examples

1. Install and configure the Informix Video Foundation DataBlade module.
2. Create a file with the path **/tmp/video/dcmetrop.mpeg**. This file can have any content, or it can be empty.

The rest of this chapter describes what occurs when you run the examples.

Registering the Video Storage Server Component

Sample Company hosts Informix Dynamic Server on a UNIX computer and stores its media component files (including those that constitute *Metropolis*) in the Dynamic Server file system. Specifically, the files are stored on the fifth storage server of type **VSI_FILE** in this media management system, in the subdirectories of the **/tmp/** directory.

The following example shows the inserts into the **MedVSISRegister** and **MedVSI_File** table to register Sample Company's storage server component with the Informix Video Foundation DataBlade module. The **MedStorage** row value that defines this video storage server is inserted into the **MedVSISRegister** table using the **MedStorage** function.


```

INSERT INTO MedVSIRegister (storage) VALUES ( MedStorage(
    'VSI_FILE',                                -- Type
    5,                                           -- StorageID
    NULL,                                       -- ConnectionName
    NULL,                                       -- UserID
    NULL,                                       -- Password
    '/tmp/',                                   -- ConnectionString
    NULL,                                       -- Vendor
    NULL));                                    -- Model

INSERT INTO MedVSI_File Values
(
    (SELECT storage.ConnectionString            -- Path
     FROM MedVSIRegister
     WHERE storage.StorageID = 5),
    0                                           -- LastNum
);

```

Inserting Physical Attributes

To keep track of its physical multimedia assets, Sample Company inserts rows into the inventory tables.

First, because *Metropolis* is a single video program, one row is inserted into the **VidMDGeneral** table:

```

INSERT INTO VidMDGeneral
(VideoID, Title, Description)
VALUES (
    0,
    'Metropolis',
    'American UFA release - with soundtrack added'
);

```

Next, because Sample Company owns two copies (or *media instances*) of *Metropolis*, two rows are inserted into the **VidMedia** table. These INSERT statements are shown and described in the following subsections.

Copy 1: BetaSP Tape

The first copy of *Metropolis* is on BetaSP tape. In the following example, information about this copy is inserted into the **VidMedia** table.

Sample Company assigns a **MediaID** value of 1001 to this copy. The values for the **VideoID**, **StandardID**, and **FormatID** columns are selected from the **VidMDGeneral**, **VidStandard**, and **VidFormat** tables, respectively.

The **MediaDerived** column has a value of 0, indicating that this BetaSP copy is the source timeline for all other copies of *Metropolis*. This copy of *Metropolis* is a little over one hour long (**Duration**) and starts at timecode 00:20:00:00 (**Offset**). The timecodes for **Duration** and **Offset** are converted to **MedPtr** data type values using the **MedPtrFrom** function.

```
INSERT INTO VidMedia
(MediaID, VideoID, StandardID, FormatID,
MediaDerived, Duration, Offset, Description)
VALUES
(
    1001,    -- Could have been 0 because it's a serial type
column          -- Server will assign the next integer automatically
    (SELECT VideoID FROM VidMDGeneral WHERE Title =
'Metropolis'),
    (SELECT StandardID FROM VidStandard WHERE Name = 'NTSC'),
    (SELECT FormatID FROM VidFormat WHERE Name = 'BetaSP-30'),
    0,      -- 0 means that this is the original source
    MedPtrFrom('01:03:13:00-D'::MedTC),
    MedPtrFrom('00:20:00:00'::MedTC),
    'Source Tape - BetaSP'
);
```

Copy 2: MPEG File

Sample Company converted the BetaSP tape to MPEG format to create the company's second copy of *Metropolis*. The MPEG copy has a **MediaID** value of 1002, while the **MediaDerived** column has a value of 1001, referencing the BetaSP source copy.

The starting timecode (**Offset**) for any copy that starts at the beginning of the video program is usually 00:00:00:00. However, because a 10-second leader was added to the MPEG copy, its starting timecode is 00:00:10:00. Thus, the starting BetaSP timecode of 00:20:00:00 (see [“Copy 1: BetaSP Tape” on page 7-6](#)) is equal to the starting MPEG timecode of 00:00:10:00.

The **Duration** column of the MPEG copy has the same value as the **Duration** column of the BetaSP source copy.

```
INSERT INTO VidMedia
(MediaID, VideoID, StandardID, FormatID, CaptureRate,
MediaDerived,
Duration, Offset, Description)
VALUES
(
    1002,
    (SELECT VideoID FROM VidMDGeneral WHERE Title =
'Metropolis'),
    (SELECT StandardID FROM VidStandard WHERE Name = 'NTSC'),
    (SELECT FormatID FROM VidFormat WHERE Name =
'Compressed'),
    29.97,
    1001, -- The MediaID of the instance that was used as
source
    MedPtrFrom('01:03:13:00-D'::MedTC),
    MedPtrFrom('00:00:10:00'::MedTC),
    'MPEG-1'
);
```

The MPEG copy of *Metropolis* is encoded, implying that there is at least one media component file. For this video program, it is the single video output file **/tmp/video/dcmetrop.mpeg**. In the following example, Sample Company inserts a row into the **VidPVideo** table for this file.

The following example also shows that the values for the **MediaID** and **VCodecID** columns are selected from the **VidMedia** and **VidVCodec** tables, respectively. At the end of the INSERT statement, the **Location** column is assigned the MedLoc value returned by the **Copy** function after it copies the video file from the client directory to VSI_FILE#5 (the video storage system set up in [“Registering the Video Storage Server Component” on page 7-4](#)).

```
INSERT INTO VidPVideo
(MediaId, VCodecID, BitRate, FWidth, FHeight, Speed,
Location)
VALUES (
  (SELECT MediaId FROM VidMedia WHERE Description = 'MPEG-
1'),
  (SELECT VCodecID FROM VidVCodec WHERE Name = 'MPEG-1-
System'),
  2100000,    -- bit rate in mega bits per second
  320,
  240,
  1000,      -- signifies normal forward play version of
this video
  Copy('VSI_CLIENT:/tmp/video/dcmetro.p.mpeg',
Storage('VSI_FILE#5:'))
);
```

Inserting Abstract Descriptions

Because Sample Company wants to reference segments (or *media chunks*) of *Metropolis* by abstract descriptions of those chunks, the company inserts rows into the subject matter tables.

First, Sample Company creates the **Who** strata and the **What** strata by inserting rows into the **VidStrata** table, as follows:

```
INSERT INTO VidStrata (StrataID, Name, StrataType,
Description)
VALUES (100, 'Who', 'editor', 'People who appear on
Screen');

INSERT INTO VidStrata (StrataID, Name, StrataType,
Description)
VALUES (101, 'What', 'editor', 'Things that appear');
```

Next, the company identifies media chunks and then associates those chunks with abstract text descriptions. Each chunk+description pair is called a *stratum*.

This section contains two sets of INSERT statements that create stratum: one set inserts explicit values into the database tables; the second inserts the results of SELECT statements into the tables.

Inserting Explicit Values

The INSERT statements shown in this section insert explicit column values into the **VidChunkMD** and **VidTextDesc** tables. The exceptions are the **Chunk** column values of the **VidChunkMD** table. For these values, the media chunk definitions are formed by timecodes being converted to media pointers (using the **MedPtrFrom** function), which are then converted to media chunks (using the **MedChunkFrom** function).

The **StrataID** column of the **VidTextDesc** table has a value of 100 if this stratum is in the **Who** strata and a value of 101 if the stratum is in the **What** strata.

```
INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2000,
MedChunkFrom(MedPtrFrom('00:43:22:12-D'::MedTC),
MedPtrFrom('00:46:01:03-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1,-- first text annotation for this chunk
2000,
100, -- stratum for people who appaer in this chunk
CURRENT year to second,
'Maria/The Robot');

INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2001,
MedChunkFrom(MedPtrFrom('00:40:01:27-D'::MedTC),
MedPtrFrom('00:44:55:14-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2001, 100, CURRENT year to second, 'Georg');

INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2002,
MedChunkFrom(MedPtrFrom('00:56:42:17-D'::MedTC),
MedPtrFrom('01:01:01:21-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2002, 100, CURRENT year to second, 'Workers
in the machine');
```

```
INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2003,
MedChunkFrom(MedPtrFrom('00:47:06:06-D'::MedTC),
MedPtrFrom('00:47:58:02-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2003, 100, CURRENT year to second, 'Freder');

INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2004,
MedChunkFrom(MedPtrFrom('00:47:06:06-D'::MedTC),
MedPtrFrom('00:49:12:25-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2004, 100, CURRENT year to second, 'Georg');

INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2005,
MedChunkFrom(MedPtrFrom('00:13:26:17-D'::MedTC),
MedPtrFrom('00:16:55:01-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2005, 101, CURRENT year to second, 'The
Skyline');

INSERT INTO VidChunkMD (MediaID, ChunkID, Chunk)
VALUES ( 1001, 2006,
MedChunkFrom(MedPtrFrom('00:59:40:03-D'::MedTC),
MedPtrFrom('01:02:22:28-D'::MedTC)) );
INSERT INTO VidTextDesc (SeqNum, ChunkID, StrataID, When,
Description)
VALUES ( 1, 2006, 101, CURRENT year to second, 'Revolution
Starts');
```

Inserting SELECT Statement Results

The INSERT statements in this section insert column values that are the results of SELECT statements into the **VidChunkMD** and **VidTextDesc** tables. With fewer “hard-coded” values, the following example is more flexible and portable—but more complex—than the example shown in [“Inserting Explicit Values” on page 7-9](#).

```

INSERT INTO  VidChunkMD (ChunkID, MediaID, Chunk)
VALUES
(
  -- ChunkID
    0,
  -- MediaID
    (SELECT m.MediaID
     FROM VidMedia m, VidMDGeneral v
     WHERE v.Title = 'Metropolis' AND
           m.VideoID = v.VideoID AND
           m.MediaDerived = 0),
  -- Chunk
    MedChunkFrom(MedPtrFrom('00:28:23:14-D'::MedTC),
                 MedPtrFrom('00:31:05:04-D'::MedTC))
);
INSERT INTO  VidTextDesc (ChunkID, StrataID, SeqNum, When,
Description)
VALUES
(
  -- ChunkID
    ( SELECT ChunkID
      FROM VidChunkMD
      WHERE MediaID = (SELECT m.MediaID
                       FROM VidMedia m, VidMDGeneral v
                       WHERE v.Title = 'Metropolis' AND
                             m.VideoID = v.VideoID AND
                             m.MediaDerived = 0)
      AND Chunk = MedChunkFrom(MedPtrFrom('00:28:23:14-
D'::MedTC),
                               MedPtrFrom('00:31:05:04-D'::MedTC)) ),
  -- StrataID
    (SELECT StrataID
     FROM VidStrata
     WHERE Name = 'Who' AND
           StrataType = 'editor' ),
  -- SeqNum
    1,
  -- When
    CURRENT year to second,
  -- Description

```

```
'Joh'  
);
```

Issuing Queries

After Sample Company has entered both physical and abstract metadata into its database, it can issue queries to select segments of interest from *Metropolis*.

This section provides examples of the following types of queries:

- Selecting the **MediaID** value of a particular media instance
- Selecting all media chunks in a single strata
- Selecting all media chunks that overlap with those in a strata
- Selecting a newly constructed media chunk

Selecting the MediaID Value of a Media Instance

The following query selects the **MediaID** value of the source copy (**Media-Derived** value of 0) of the video program with a title containing the string Metro. The value 1001 is returned.

```
SELECT m.MediaID  
FROM VidMedia m, VidMDGeneral v  
WHERE  
v.Title LIKE '%Metro%'  
AND m.VideoID = v.VideoID  
AND m.MediaDerived = 0;
```

Selecting Media Chunks in a Single Strata

The following query selects the media chunk definition and text description of all stratums in the source copy that are in the **Who** strata. These rows are retrieved:

- Maria / The Robot
- Georg

- Workers in the machine
- Freder
- Georg
- Joh

```
SELECT c.Chunk, t.Description
FROM VidChunkMD c, VidTextDesc t, VidStrata s
WHERE
    c.MediaID = 1001
    AND s.Name = 'Who'
    AND c.ChunkID = t.ChunkID
    AND t.StrataID = s.StrataID;
```

Selecting Media Chunks that Overlap with a Strata

After stratoms that are in the **Who** strata have been selected, they can be compared with other media chunks in the video program. For example, the following query uses the **Overlap** function to select all media chunks that overlap those in the **Who** strata. The following rows are returned:

- Maria/The Robot + Georg
- Georg + Maria/The Robot
- Freder + George
- George + Freder

To find other relationships among the media chunks, Sample Company can substitute one of the other MedChunk comparison functions (such as **After**, **Before**, **Contains**, **Meets**, and so on) for the **Overlap** function.

```
SELECT c1.Chunk, t1.Description, c2.Chunk, t2.Description
FROM VidChunkMD c1, VidChunkMD c2, VidTextDesc t1,
VidTextDesc t2,
VidStrata s
WHERE
    c1.MediaID = 1001
    AND c2.MediaID = 1001
    AND s.Name = 'Who'
    AND c1.ChunkID = t1.ChunkID
    AND t1.StrataID = s.StrataID
    AND c2.ChunkID = t2.ChunkID
    AND t2.StrataID = s.StrataID
    AND c1.ChunkID != c2.ChunkID
    AND Overlap(c1.Chunk, c2.Chunk);
```

Selecting a Newly Constructed Media Chunk

Sample Company can dynamically construct media chunks using the **Plus**, **Minus**, and **Intersection** functions. For example, the following query uses the **Plus** and **Overlap** functions to select a new media chunk that starts at the media chunk with the description Maria/The Robot, ends at the media chunk with the description Georg, and contains all points in between. (The returned expression is 2401901901902N-00:40:01:27-D|2761061061061N-00:46:01:03-D.)

To construct and select a media chunk that contains all the points in the first media chunk not included in the second, Sample Company can use the **Minus** function. To construct or select a media chunk that contains only the points in common between the two described chunks, the company can use the **Intersection** function.

```
SELECT c1.Chunk + c2.Chunk
FROM VidChunkMD c1, VidChunkMD c2, VidTextDesc t1,
VidTextDesc t2
WHERE
    c1.MediaID = 1001
    AND c2.MediaID = 1001
    AND c1.ChunkID = t1.ChunkID
    AND c2.ChunkID = t2.ChunkID
    AND t1.Description = 'Maria/The Robot'
    AND t2.Description = 'Georg'
    AND Overlap(c1.Chunk, c2.Chunk);
```

Requesting Access to a Segment

To request access to a segment of the movie (called a *media chunk*), Sample Company's client application issues the **ClientAccess** function. The following query uses the **ClientAccess** and **Location** functions to request the **Location** element of a MedLoc value granting permission to play back the media chunk containing the entire MPEG copy of *Metropolis*.

```
SELECT Location(ClientAccess(pvid.Location,1,0,''))
FROM   VidMDGeneral md, VidMedia vid,
       VidPVideo pvid
WHERE  md.Title = 'Metropolis' AND
       vid.VideoID = md.VideoID AND
       vid.FormatID = 13 AND -- Format ID for compressed video
       pvid.MediaID = vid.MediaID AND
       pvid.VCodecID = 2 AND -- Video codec ID for MPEG1-System
       pvid.Speed = 1000;    -- The normal forward play file
```

Tracking Nondigital Multimedia Assets

Sample Company has decided to also track the contents of its video tape library (which it keeps in numbered boxes in the store room) in its media management system. It follows these steps, shown in the next example:

- Creates the **TapeLocation** schema table to store information on each tape in the tape library
- Inserts rows into the table for each tape in the library, using its own numbering system and taking advantage of the VSI_OTHER implementation provided by the Informix Video Foundation DataBlade module (in the example, only one INSERT statement is shown)
- Selects the location and other information about a media chunk in the tape copy of *Metropolis*; specifically:
 - The tape is located at TA1632.C44
 - The text description of the media chunk is Maria/The Robot
 - The starting timecode of the media chunk is 01:03:22:12-D
 - The ending timecode of the media chunk is 01:06:01:03-D

```
CREATE TABLE TapeLocation (  
    MediaID int, -- Used for joining with VidMedia table  
    RoomNum char(25),  
    TapeLocator MedLoc, -- Using VSI_OTHER and storing the  
    locator  
        -- in 'location' part of MedLoc  
    CopyNum int, -- May have multiple copies of the tape  
    Status char(1), -- Tape checked out ?  
  
    PRIMARY KEY (MediaID, CopyNum),  
    FOREIGN KEY (MediaID) REFERENCES VidMedia  
);  
  
INSERT INTO TapeLocation VALUES (  
    1001,  
    'ENS 601',  
    'VSI_OTHER#1:TA1632.C44',  
    1,  
    'N'  
);  
  
SELECT Location(t.TapeLocator), dsc.Description,  
    MedTCFrom(StartP(md.chunk) + vid.offset, 29.97),  
    MedTCFrom(EndP(md.chunk) + vid.offset, 29.97)  
FROM VidChunkMD md, VidTextDesc dsc, VidMedia vid, VidStrata  
s,  
    TapeLocation t  
WHERE md.ChunkID = 2000 AND  
    md.MediaID = t.MediaID AND  
    dsc.ChunkID = md.ChunkID AND  
    s.StrataID = dsc.StrataID AND  
    s.StrataType = 'editor' AND  
    s.Name = 'Who' AND  
    vid.MediaID = md.MediaID;
```

Writing a Virtual Storage DataBlade Module

In This Chapter	8-3
Registering the DataBlade Module with Informix	8-4
Implementing a Media Point UDT	8-4
Defining, Writing, and Registering the VSI UDRs	8-5
Device-Independent VSI UDRs	8-6
Defining VSI UDR Objects	8-7
Writing VSI UDRs	8-9
ClientAccess Function	8-11
MedLoc Data Type	8-12
Assign, Copy, and New Functions	8-12
Destroy Procedure	8-14
MedLoc_fd Data Type	8-14
Registering VSI UDRs with the Informix Video Foundation Module	8-14
Documenting the Virtual Storage DataBlade Module	8-17

In This Chapter

This chapter describes how to write a storage-server-specific Virtual Storage Interface (VSI) implementation, referred to as a *virtual storage DataBlade module* or as a *Video Foundation Interface*. For information about the VSI and what it provides, see [Chapter 3, “Storage Component.”](#)

The process of writing any DataBlade module involves setting up a Blade-Smith project environment and using it and other DataBlade Developers Kit tools to organize the project, create data types and functions, generate files, and build an installation package. The specifics of using these tools are described in the [DataBlade Developers Kit User’s Guide](#); you will also need the [DataBlade API Programmer’s Manual](#) for information on writing functions.

To write a virtual storage DataBlade module, you must complete these steps:

1. Register your DataBlade module with Informix; see [“Registering the DataBlade Module with Informix” on page 8-4.](#)
2. Decide if you need to supply a storage-server-specific media point user-defined data type (UDT). If you do, you must also supply two conversion functions for that UDT; see [“Implementing a Media Point UDT” on page 8-4.](#)
3. Write storage-server-specific VSI user-defined routines (UDRs) and register them with the Informix Video Foundation DataBlade module; see [“Defining, Writing, and Registering the VSI UDRs” on page 8-5.](#)
4. Document your virtual storage DataBlade module for your users. Some suggestions for topics to address are listed starting in [“Documenting the Virtual Storage DataBlade Module” on page 8-17.](#)

Examples in this chapter are based on the fictitious Sample Virtual Storage DataBlade module, which implements the VSI for the fictitious Sample Company's SAM 4000-PRO video server. You can also use the Informix VSI implementations for popular storage servers as examples; see the *Informix Video Foundation Interfaces Supplement* for information.

Registering the DataBlade Module with Informix

Before you set up a BladeSmith project environment, you must register your virtual storage DataBlade module with Informix. Do this by making a request for a DataBlade module name, one or more DataBlade module object prefixes, and a range of error message codes. (For details, refer to the [DataBlade Developers Kit User's Guide](#) or contact the Informix DataBlade Developers Program by sending electronic mail to registry@informix.com.)

As an example, Sample Company might register the prefix *SAM* with Informix.

Important: Informix reserves the *VSI*, *VID*, and *MED* prefixes for use in its own video and multimedia products.



Implementing a Media Point UDT

After you have set up your BladeSmith project environment, import any other DataBlade modules that your DataBlade module depends on. At a minimum, this includes the Informix Video Foundation DataBlade module's interfaces and data types.

The Informix Video Foundation DataBlade module provides the *MedPtr* and *MedTC* user-defined data types (UDTs) to describe precise locations in time within a video program (*media points* and *timecodes*). It also provides functions that convert values among these data types, integer frame numbers, and built-in data types. (For information, see [Chapter 2, "Temporal Component."](#))



You can use one of these UDTs to represent media points stored on your storage server, or you can supply your own. If you supply your own, you must also supply two conversion functions: a **MedPtrFrom** function that takes your UDT as input and returns a MedPtr, and another function that takes a MedPtr as input and returns your UDT. You do this by defining new objects in your BladeSmith project.

Important: The **MedPtrFrom** function object must be named “**MedPtrFrom**.” The names of your media point UDT and its associated conversion function, however, must start with one of your DataBlade module object prefixes (see the previous section), but otherwise can be any valid SQL object identifier.

For example, Sample Company’s VSI implementation represents media point data in seconds from the beginning of a video program. Sample Company is providing the SamTimeUnit UDT, which is a distinct type with a source type of INT8. The company is also providing the following functions:

- MedPtrFrom(SamTimeUnit) returns MedPtr
- SamTimeUnitFrom(MedPtr) returns SamTimeUnit

Defining, Writing, and Registering the VSI UDRs

The VSI includes the user-defined routines (UDRs) listed in the next section. Each of these routines is device *independent*; they can be called with MedLoc and MedStorage values for different types of storage servers. Internally, however, the *Type* element of the MedLoc or MedStorage value is inspected and used to select the correct *storage-server-specific* VSI UDR implementation.

In writing your virtual storage DataBlade module, you must provide VSI UDRs for your storage server. To help with this task, this section provides the following information:

- A listing of the device-independent VSI UDRs
- Information on how to define these objects in your BladeSmith project
- Information on how to write storage-server-specific VSI UDRs
- Information on how to register your VSI UDRs with the Informix Video Foundation DataBlade module



Device-Independent VSI UDRs

The following table is an alphabetical listing of each device-independent VSI UDR. It shows the external interface through which each function can be called and lists the page number on which the function is described.

Important: The **Assign** function and the **Destroy** procedure do not have external interfaces. They are internal UDRs that are called automatically by Dynamic Server as a result of SQL statements that insert, update, or delete a column of type MedLoc.

VSI Name	SQL Call	VSI Programming Interface	Page
Assign	—	—	8-12
ClientAccess	function ClientAccess (MedLoc, INTEGER, INTEGER, LVARCHAR) returns MedLoc	—	5-13
Close	—	mi_integer MedLocClose(mi_bitvarying *loc_fd, MI_FPARAM *fp)	9-7
Copy	function Copy (MedLoc, MedStorage) returns MedLoc	—	5-26
Destroy	—	—	8-14
New	function New (MedStorage) returns MedLoc	—	5-73
Open	—	mi_bitvarying * MedLocOpen(mi_bitvarying *medloc, mi_integer flags, MI_FPARAM *fp)	9-8
Read	—	mi_integer MedLocRead(mi_bitvarying *loc_fd, mi_char *buf, mi_integer bytes, MI_FPARAM *fp)	9-10

(1 of 2)

VSI Name	SQL Call	VSI Programming Interface	Page
Seek	—	void MedLocSeek(mi_bitvarying *loc_fd, mi_int8 *offset, mi_integer whence, MI_FPARAM *fp)	9-11
Shutdown	function Shutdown (MedStorage) returns INTEGER	—	5-94
Startup	function Startup (MedStorage) returns INTEGER	—	5-100
Status	function Status (MedStorage) returns INTEGER	—	5-102
Tell	—	void MedLocTell(mi_bitvarying *loc_fd, mi_int8 *offset, MI_FPARAM *fp)	9-13
Write	—	mi_integer MedLocWrite(mi_bitvarying *loc_fd, mi_char *buf, mi_integer bytes, MI_FPARAM *fp)	9-15

(2 of 2)

Defining VSI UDR Objects

Using BladeSmith, define a new object for each of your VSI UDRs. Your UDR names must begin with one of your DataBlade module object prefixes (see [“Registering the DataBlade Module with Informix” on page 8-4](#)), but otherwise they can be any valid SQL object identifier. For example, Sample Company named its functions following the syntax *SamVsiName*, so its implementation of the **Write** function is named **SamWrite**.

Defining VSI UDR Objects

The following table lists how each BladeSmith object property should be set for each of your VSI UDRs. The UDR objects are listed by the names Sample Company used to define its UDR objects.

UDR Object	Parameters	Returns	Internal	Language	Variant	handlesNulls
SamAssign	(MedLoc)	MedLoc	TRUE	C	TRUE	FALSE
SamClientAccess	(MedLoc, INTEGER, INTEGER, LVARCHAR)	MedLoc	TRUE	C	TRUE	FALSE
SamClose	(MedLoc_fd)	INTEGER	TRUE	C	TRUE	FALSE
SamCopy	(MedLoc, MedStorage)	MedLoc	TRUE	C	TRUE	FALSE
SamDestroy	(MedLoc)	none ^a	TRUE	C	TRUE	FALSE
SamNew	(MedStorage)	MedLoc	TRUE	C	TRUE	FALSE
SamOpen	(MedLoc, INTEGER)	MedLoc_fd	TRUE	C	TRUE	FALSE
SamRead	(MedLoc_fd, INTEGER, INTEGER)	INTEGER	TRUE	C	TRUE	FALSE
SamSeek	(MedLoc_fd, INTEGER, INTEGER)	none ^a	TRUE	C	TRUE	FALSE
SamShutdown	(MedStorage)	INTEGER	TRUE	C	TRUE	FALSE
SamStartup	(MedStorage)	INTEGER	TRUE	C	TRUE	FALSE
SamStatus	(MedStorage)	INTEGER	TRUE	C	TRUE	FALSE

UDR Object	Parameters	Returns	Internal	Language	Variant	handlesNulls
SamTell	(MedLoc_fd, INTEGER)	none ^a	TRUE	C	TRUE	FALSE
SamWrite	(MedLoc_fd, INTEGER, INTEGER)	INT8	TRUE	C	TRUE	FALSE

a. UDRs that do not return any values are procedures, not functions.

Writing VSI UDRs

After you have defined the VSI UDR objects, use BladeSmith to generate the C source code for those objects. Next, edit the `.c` file to add your code to the function declarations, following the guidelines in the [DataBlade Developers Kit User's Guide](#) and the [DataBlade API Programmer's Manual](#).

In particular, for your storage-server-specific implementation of the VSI UDRs, follow these basic guidelines:

- The MedStorage row values (defined in “[External Representation](#)” on page 3-5) are specified as DataBlade API `MI_ROW` types. Use the API's row and column access functions, such as `mi_value`, to access their data.
- The MedLoc (defined in “[MedLoc Data Type](#)” on page 3-7) and MedLoc_fd values (defined in “[MedLoc_fd Data Type](#)” on page 9-3) are wrapped in DataBlade API `mi_bitvarying` structures as required for variable-length data types. Use the API's variable-length data type access functions, such as `mi_get_vardata`, to access their data.
- The DataBlade API `MI_FPARAM` argument is standard for all UDRs.

- Except for the function names, your UDRs must have the same prototypes that Sample Company's do; these prototypes are shown in the following table.

VSI Name	Sample Company's Implementation
Assign	<code>mi_bitvarying * SamAssign(mi_bitvarying *medloc, MI_FPARAM *fp)</code>
ClientAccess	<code>mi_bitvarying * SamClientAccess(mi_bitvarying *medloc, int4 service1, int4 service2, mi_lvarchar *svcars, MI_FPARAM *fp)</code>
Close	<code>mi_integer SamClose(mi_bitvarying *loc_fd, MI_FPARAM *fp)</code>
Copy	<code>mi_bitvarying * SamCopy(mi_bitvarying *src_loc, MI_ROW *dest_store, MI_FPARAM *fp)</code>
Destroy	<code>void SamDestroy(mi_bitvarying *medloc, MI_FPARAM *fp)</code>
New	<code>mi_bitvarying * SamNew(MI_ROW *storage, MI_FPARAM *fp)</code>
Open	<code>mi_bitvarying * SamOpen(mi_bitvarying *medloc, mi_integer flags, MI_FPARAM *fp)</code>
Read	<code>mi_integer SamRead(mi_bitvarying *loc_fd, mi_char *buf, mi_integer bytes, MI_FPARAM *fp)</code>
Seek	<code>void SamSeek(mi_bitvarying *loc_fd, mi_int8 *offset, mi_integer whence, MI_FPARAM *fp)</code>
Shutdown	<code>mi_integer SamShutdown(MI_ROW *storage, MI_FPARAM *fp)</code>
Startup	<code>mi_integer SamStartup(MI_ROW *storage, MI_FPARAM *fp)</code>
Status	<code>mi_integer SamStatus(MI_ROW *storage, MI_FPARAM *fp)</code>
Tell	<code>void SamTell(mi_bitvarying *loc_fd, mi_int8 *offset, MI_FPARAM *fp)</code>
Write	<code>mi_integer SamWrite(mi_bitvarying *loc_fd, mi_char *buf, mi_integer bytes, MI_FPARAM *fp)</code>

For most storage server types, the UDRs you must supply are straightforward; see their reference pages (listed in [“Device-Independent VSI UDRs” on page 8-6](#)) for information on the tasks they must perform. The following UDRs and data types, however, can raise additional issues, which are addressed in the next sections:

- **ClientAccess** function
- MedLoc data type
- **Assign**, **Copy**, and **New** functions
- **Destroy** procedure
- MedLoc_fd data type

ClientAccess Function

The **ClientAccess** function returns a handle to be used by the client application to access MedLoc data directly from a storage server. The client must request access operation permissions using the arguments to **ClientAccess**. You can handle the request for services as is appropriate for your storage server and the needs of your users.

The **ClientAccess** function returns a MedLoc value. Again, depending on the capabilities of your storage server and the needs of your users, this value can be one of the following:

- The same as the input MedLoc value. This is appropriate if your implementation does not enforce any access restrictions.
- A MedLoc value with encoded or encrypted access- or service-control information in the Location element. This is appropriate if your storage server can restrict access to only those users with properly encoded MedLoc values.

MedLoc Data Type

The MedLoc data type is defined in [“MedLoc Data Type” on page 3-7](#). The *Type* element—used to distinguish your MedLoc values from other storage server MedLoc values—must start with one of your DataBlade module object prefixes; see [“Registering the DataBlade Module with Informix” on page 8-4](#). In addition, your users must specify this same *Type* string in entries for your storage server type in the **MedVSIRegister** table; see [“MedVSIRegister Table” on page 3-6](#).

The *Location* element of the MedLoc data type is storage-server-specific; Informix places no restrictions on this element other than it be NULL-terminated. Use this element to encode location information of large media objects on your storage server type or to hold other information, such as physical attributes of the video data that must be communicated with the location.



Important: *MedLoc values can be allocated only by SQL INSERT and UPDATE statements and deallocated only by UPDATE and DELETE statements. See the next two sections for more information.*

Assign, Copy, and New Functions

The **Assign** function is called by Dynamic Server in response to an SQL INSERT or UPDATE statement that affects a table column of type MedLoc. The input MedLoc value represents the large media object that the caller wants to insert into the database. Your implementation of the **Assign** UDR must allocate storage for this large media object, copy the media data, and return a new MedLoc value representing the new large media object on your storage server.

Your implementation of **Assign** is called only when the *Type* element of the input MedLoc value matches the type of your storage server. Your **Assign** function must handle the following sources of input MedLoc values:

- An input MedLoc value from a previous call to your implementation of the **Assign** function

The caller is inserting a copy of the data into another row in the database. Your **Assign** function must copy the data from the existing large media object so that updates to the original object do not affect the copy.

- An input MedLoc value from your implementation of the **New** function

The caller is inserting a MedLoc value for a new large media object into the database. To maintain a one-to-one correspondence between MedLoc values in the database and the storage server objects they represent, your **New** function must return a MedLoc value with a request string (for example, “new”) instead of a valid object name in the MedLoc value’s *Location* element. Your **Assign** function then allocates the large media object on your storage server and returns a MedLoc value with the new object’s location as part of a single, atomic insert operation.

- An input MedLoc value from your implementation of the **Copy** function

The caller is copying an existing large media object to your storage server and inserting a MedLoc value that represents the copy into the database. To maintain a one-to-one correspondence between MedLoc values in the database and the storage server objects they represent, your **Copy** function must return a MedLoc value with a request string (for example, “copy”) instead of a valid object name in the MedLoc value’s *Location* element. The copy request must include the source MedLoc value, for example:

```
'SAM#2:copy from VSI_CLIENT:C:\videos\myfile.mpg'
```

Your **Assign** function then allocates the large media object on your storage server and uses the VSI programming interface (see [Chapter 9](#)) to copy data from the source MedLoc value.



Important: Your **Copy** and **New** implementations do not allocate MedLoc values on your storage server. Instead, they call your **Assign** implementation to handle the creation of MedLoc values and the copying of data from a source MedLoc value.

In an UPDATE operation, Dynamic Server first calls the **Destroy** procedure (see the next section, “[Destroy Procedure](#)”) to delete the original large media object; then it calls the **Assign** function to write the new value.

For more information on the **Assign** function, consult the [DataBlade Developers Kit User’s Guide](#).

Destroy Procedure

The **Destroy** procedure is called by Dynamic Server in response to an SQL DELETE or UPDATE statement that affects a table column of type MedLoc. Your implementation of the **Destroy** UDR must delete the specified large media object from the storage server and free all resources associated with that object. Because it is a procedure, the **Destroy** UDR does not return a value.

In an UPDATE operation, Dynamic Server first calls the **Destroy** procedure to delete the original large media object; then it calls the **Assign** function (see the previous section, “[Assign, Copy, and New Functions](#)”) to write the new value.

For more information on the **Destroy** procedure, consult the [DataBlade Developers Kit User’s Guide](#).

MedLoc_fd Data Type

The MedLoc_fd data type is defined in “[MedLoc_fd Data Type](#)” on page 9-3. The device-independent functions access only the *Type* element. Your UDRs can access both the *Type* and *Data* elements, as appropriate for your storage server. For example, the *Data* element might contain a file or object descriptor, a cast to a structure required by this storage server, or the name of a local file in which a remote storage server is cached.

Registering VSI UDRs with the Informix Video Foundation Module

Before using BladeSmith to generate SQL scripts, you must add an SQL file object to your project. This object contains SQL commands that register your VSI UDRs with the Informix Video Foundation DataBlade module.

To register your virtual storage DataBlade module’s VSI UDRs with the video foundation, you insert one row per UDR into the video foundation’s **MedVSIFunctions** table. This table allows Dynamic Server to call the appropriate storage-server-specific VSI UDR in response to a device-independent VSI function. Functions that are not implemented must have a **Udr_signature** column value of NULL.

The layout of the **MedVSIFunctions** table is shown here. The **Type** and **Vsi_name** columns together form a primary key; values in the **Vsi_name** and **Udr_signature** columns must be in lowercase letters.

Column Name	Type	Sample Data	Explanation
Type	CHAR (18)	'SAM'	Storage server type, used with Vsi_name to select the storage-server-specific UDR for MedLoc and MedStorage values.
Vsi_name	CHAR (30)	'startup'	One of the VSI function names (see “Device-Independent VSI UDRs” on page 8-6), in lowercase. This value is used with Type to select the storage-server-specific UDR for MedLoc and MedStorage values.
Udr_signature	LVARCHAR	'function samstartup(medstorage)'	The signature of the storage-server-specific UDR that implements the VSI UDR specified in Vsi_name , in lowercase. If Type does not support Vsi_name , set this column to NULL.

The following example shows Sample Company's inserts into the **MedVSI-Functions** table for its VSI UDRs:

```
insert into MedVSIFunctions values('SAM', 'assign',
    'function samassign(medloc)');

insert into MedVSIFunctions values('SAM', 'clientaccess',
    'function samclientaccess(medloc, integer, integer, lvarchar)');

insert into MedVSIFunctions values('SAM', 'close',
    'function samclose(medstorage)');

insert into MedVSIFunctions values('SAM', 'copy',
    'function samcopy(medloc, medstorage)');

insert into MedVSIFunctions values('SAM', 'destroy',
    'procedure samdestroy(medloc)');

insert into MedVSIFunctions values('SAM', 'new',
    'function samnew(medstorage)');

insert into MedVSIFunctions values('SAM', 'open',
    'function samopen(medloc, integer)');

insert into MedVSIFunctions values('SAM', 'read',
    'function samread(medloc_fd, integer, integer)');

insert into MedVSIFunctions values('SAM', 'seek',
    'procedure samseek(medloc_fd, integer, integer)');

insert into MedVSIFunctions values('SAM', 'shutdown',
    'function samshutdown(medstorage)');

insert into MedVSIFunctions values('SAM', 'startup',
    'function samstartup(medstorage)');

insert into MedVSIFunctions values('SAM', 'status',
    'function samstatus(medstorage)');

insert into MedVSIFunctions values('SAM', 'tell',
    'procedure samtell(medloc_fd, integer)');

insert into MedVSIFunctions values('SAM', 'write',
    'function samwrite(medloc_fd, integer, integer)');
```

If Sample Company decided not to implement the **Seek** and **Tell** procedures, the entries for those UDRs would look like this:

```
insert into MedVSIFunctions values('SAM', 'seek', 'NULL');

insert into MedVSIFunctions values('SAM', 'tell', 'NULL');
```

Documenting the Virtual Storage DataBlade Module

The details of the documentation that you provide to your users depend on your and your users' needs and the capabilities of your DataBlade module. Consider addressing the following topics in your documentation:

- Describe the elements of a media management system that includes your storage server and your virtual storage DataBlade module. In particular, list any hardware or software requirements. (For a description of the basic media management system, see [“Definition of Terms” on page 1-3.](#))
- If you have provided any UDTs, document them and their associated functions.
- Document the DataBlade module object prefix that you registered with Informix (see [“Registering the DataBlade Module with Informix” on page 8-4](#)) and the valid *Type* element used to distinguish your storage server's MedLoc and MedStorage values. This *Type* element string must start with your registered prefix. For example, if Sample Company registered *SAM* with Informix, its *Type* element could also be *SAM*, or it could be *Sample* or *SampleCo*.
- Document the valid format of your storage server's MedLoc values, especially the syntax of the *Location* element (for a review of the MedLoc data type, see [“MedLoc Data Type” on page 3-7](#)).
- Describe any differences between the VSI functions documented in this guide and your storage-server-specific implementations; for example:
 - Does **ClientAccess** provide access to only a few services? Does it return a specially encoded MedLoc value?
 - Does **Status** return storage-server-specific status information? Or have you provided another method for retrieving status information?
 - Do your functions raise error messages in addition to the ones raised by the Informix Video Foundation DataBlade module?

- For the database administrator, document the following:
 - Because the VSI UDRs for each storage server type invoke network calls in an Informix Extension Virtual Processor (EVP), tell your users how to set up a dedicated EVP for your DataBlade module. They must insert a line like the following one into the **\$INFORMIXDIR/etc/\$ONCONFIG** file:

```
VPCLASS      prefix,num=1
```

For example, Sample Company might inform its customers to insert the following line:

```
VPCLASS      sam,num=1
```
 - Document any mandatory storage-server-specific values or formats of your MedStorage row values' fields. For example, to reference a SAM 4000-PRO storage server, a MedStorage row value's **Type** field must be 'SAM'. Another storage server's **ConnectionString** field might require site-specific data in a storage-server-specific format.
 - Remind your users that for each separate storage server in their media management system (whether they be separate computers or separate processes), they must insert a single row in the **MedVSIRegister** table (see [“MedVSIRegister Table” on page 3-6](#)). In addition, if they are using the VSI_FILE implementation, they must insert a corresponding row in the **MedVSI_File** table (see [“VSI_FILE” on page 3-10](#)).
 - Consider providing a list of the UDRs that your DataBlade module **register.sql** file will add to your users' **MedVSIFunctions** table.

VSI Programming Interface Reference

In This Chapter	9-3
The VSI Programming Interface Library	9-3
MedLoc_fd Data Type	9-3
Functions	9-4
MedLocClose	9-7
MedLocOpen	9-8
MedLocRead	9-10
MedLocSeek	9-11
MedLocTell	9-13
MedLocWrite	9-15

In This Chapter

This chapter describes the Informix Video Foundation DataBlade module's VSI programming interface. This interface is a device-independent way for the UDRs in your DataBlade module to directly access MedLoc data. (For a definition of the MedLoc data type, see [“MedLoc Data Type” on page 3-7](#).) Situations where such access is useful include copying a MedLoc value from one storage server type to another or opening and reading a MedLoc value for processing.

This chapter describes how this programming interface is supplied; then it describes the data type and functions that make up the interface.

The VSI Programming Interface Library

To use the VSI programming interface library, include the following header file in your source code: `$INFORMIXDIR/extend/Video.1.10/incl/vsi.h`.

Then add the following library file to your link list: `$INFORMIXDIR/extend/Video.1.10/lib/server/vsi.a`.

MedLoc_fd Data Type

MedLoc_fd is an opaque data type that the Informix Video Foundation DataBlade module uses to maintain the state of an open large media object between access calls. It is returned by the **MedLocOpen** function, and it must be used as input to the other VSI programming interface functions.

This data type is represented by a C structure with a maximum length of 1024 characters; it has the following members:

<i>Type</i>	A NULL-terminated character string, with a maximum length of 19 characters (including the NULL character). This string must match the value in the Type field of the MedStorage row value for the open MedLoc value's storage server (see “MedStorage Data Type” on page 3-4).
<i>Data</i>	A storage-server-specific byte array, with a maximum length of 1,005 characters.

Functions

The VSI programming interface functions let you directly access MedLoc data. These functions take a MedLoc or MedLoc_fd value as input; using the *Type* information in that value, they look up the storage-server-specific VSI function to call in the **MedVSIFunctions** table. (For more information on this process and the **MedVSIFunctions** table, see [Chapter 8, “Writing a Virtual Storage DataBlade Module.”](#))

To read, write, seek, tell, or close a MedLoc value, you must use a valid MedLoc_fd value. These are returned from the **MedLocOpen** function.

Each of these functions takes an MI_FPARAM argument; pass it the MI_FPARAM argument that the UDR you are writing received as input. For information on this function parameter structure, consult the [DataBlade API Programmer's Manual](#).

The rest of this chapter provides the following reference information for these functions:

- A brief description of the task it performs
- Its syntax
- A brief description of each of its input arguments
- A brief description of the value it returns

Individual examples are not provided. Instead, the following C program shows the VSI programming interface functions being used to implement Sample Company's **SamLength**, **SamExtract**, and **SamUpdate** functions.

```
#include <string.h>
#include <mi.h>
#include <vsi.h>

mi_integer SamLength(/* return length of storage object */
    mi_bitvarying* wloc, /* wrapped MedLoc struct */
    MI_FPARAM * fp) /* standard UDR parameter */
{
    mi_bitvarying* fd; /* MedLoc_fd wrapped */
    mi_int8 offset; /* file read/write position */
    mi_integer retval; /* regular integer for UDR return value */

    fd = MedLocOpen( wloc, MED_READ | MED_SEEK, fp );
    ifx_int8cvint(0, &offset);
    MedLocSeek( fd, &offset, MED_SEEK_END, fp ); /* seek to end */
    MedLocTell( fd, &offset, fp ); /* read current position */
    MedLocClose( fd, fp );
    if (ifx_int8toint( &offset, &retval ) >= 0)
        return( retval );
    else
        return( -1 ); /* int8 could not be converted */
}

mi_lvarchar* SamExtract(/* returns bytes read from source at offset */
    mi_bitvarying* wloc, /* wrapped MedLoc source for read */
    mi_int8 *offset, /* position for start of read */
    mi_integer bytes, /* number of bytes to read */
    MI_FPARAM * fp) /* standard UDR parameter */
{
    mi_bitvarying* fd; /* MedLoc_fd wrapped */
    mi_lvarchar* wtxt; /* wrapped char array for return value */

    fd = MedLocOpen( wloc, MED_READ, fp );
    MedLocSeek( fd, offset, MED_SEEK_SET, fp );
    wtxt = mi_new_var( bytes );
    MedLocRead( fd, mi_get_vardata(wtxt), mi_get_varlen(wtxt), fp );
    MedLocClose( fd, fp );
    return( wtxt );
}

mi_integer SamUpdate(
    mi_bitvarying* wloc, /* wrapped MedLoc destination for write */
    mi_int8 *offset, /* position for start of write */
    mi_lvarchar* wtxt, /* wrapped char array to write */
    MI_FPARAM * fp) /* standard UDR parameter */
{
    mi_bitvarying* fd; /* MedLoc_fd wrapped */
    mi_integer bytes; /* return bytes written to destination */

    fd = MedLocOpen( wloc, MED_WRITE, fp );
```

Functions

```
MedLocSeek( fd, offset, MED_SEEK_SET, fp );
bytes = MedLocWrite( fd, mi_get_vardata(wtxt), mi_get_varlen(wtxt), fp );
MedLocClose( fd, fp );
return( bytes );
}
```

MedLocClose

Closes a large media object and releases any resources allocated by **MedLocOpen** (see [“MedLocOpen” on page 9-8](#)).

Syntax

```
mi_integer MedLocClose(mi_bitvarying *wfd, MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wfd</i>	The MedLoc_fd value specifying the large media object to access. The MedLoc_fd value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.

Returns

Returns an MI_INTEGER value of 0 if the close was successful or a value of -1 if an error occurred.

Example

See [page 9-5](#).

MedLocOpen

Opens a MedLoc value (see [“MedLoc Data Type” on page 3-7](#)) and returns a MedLoc_fd value to be used as input to the other VSI programming interface functions. The file position for reads and writes is set to the beginning (0) of the large media object; see [“MedLocTell” on page 9-13](#) for more information.

Syntax

```
mi_bitvarying * MedLocOpen(mi_bitvarying *wloc,
                           mi_integer flags, MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wloc</i>	The MedLoc value specifying the large media object to open. The MedLoc value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.
<i>flags</i>	Specifies one or more access permissions for the returned MedLoc_fd.	Must be one or more of the values shown in the next table; can be combined with the OR operator.

The *flags* argument must have one or more of the following values. For information on the *file position*, see [“MedLocTell” on page 9-13](#).

Permission	Meaning
MED_READ	Open the MedLoc value for reading.
MED_WRITE	Open the MedLoc value for writing.

Permission	Meaning
MED_SEEK	Open the MedLoc value and allow setting of the file position with the MedLocSeek function (see “MedLocSeek” on page 9-11).
MED_APPEND	If combined with MED_WRITE with the OR operator, the file position is set to the end of any existing data before a write operation.
MED_TRUNC	Open the MedLoc value and delete any existing data.

Returns

Returns a pointer to an **mi_bitvarying** structure that contains a MedLoc_fd value (see [“MedLoc_fd Data Type” on page 9-3](#)).

Example

See [page 9-5](#).

MedLocRead

Reads data from a large media object into a buffer and returns the number of bytes read. The data is read starting at the current file position; see [“MedLocTell” on page 9-13](#).

Syntax

```
mi_integer MedLocRead(mi_bitvarying *wfd, mi_char *buffer,
                      mi_integer bytes, MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wfd</i>	The MedLoc_fd value specifying the large media object to access. The MedLoc_fd value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.
<i>*buffer</i>	Specifies the buffer into which to read the data.	N.A.
<i>bytes</i>	Specifies the number of bytes to read.	Must be a positive integer.

Returns

Returns an `mi_integer` value specifying the number of bytes read. This value is less than the input *bytes* if the end of the large media object occurs before that number of bytes is read.

Example

See [page 9-5](#).

MedLocSeek

Sets the current file position in a large media object for read and write operations. (See [“MedLocTell” on page 9-13](#) for information on the file position.)

Syntax

```
void MedLocSeek(mi_bitvarying *wfd, mi_int8 *offset,
                mi_integer whence, MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wfd</i>	The MedLoc_fd value specifying the large media object to access. The MedLoc_fd value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.
<i>*offset</i>	On input, specifies the file position to set. On return, specifies the file position that was set (see “Returns,” below).	Must be a positive integer.
<i>whence</i>	Specifies where the file position is set.	Must be one of the values listed in the next table.

The *whence* argument must have one of the following values.

Attribute	Meaning
MED_SEEK_SET	Set the file position relative to the beginning (0) of the MedLoc value.
MED_SEEK_CUR	Set the file position relative to the current position within the MedLoc value.
MED_SEEK_END	Set the file position relative to the end of the MedLoc value (the offset is subtracted from the end).

Returns

Returns no value. However, if *offset* precedes the beginning or exceeds the end of the large media object, the closest valid file position is used and is returned in the *offset* argument.

Example

See [page 9-5](#).

MedLocTell

Retrieves the current file position of a large media object.

The *current file position* is the location in the large media object where the next read or write operation starts. The position can be affected by any of the following factors:

- If the **MedLocOpen** function was invoked with the *flags* argument set to MED_APPEND (see [“MedLocOpen” on page 9-8](#))
- If the **MedLocOpen** function was invoked with the *flags* argument set to MED_SEEK (see [“MedLocOpen” on page 9-8](#)), and the **MedLocSeek** function was then invoked to set the file position (see [“MedLocSeek” on page 9-11](#))
- If the **MedLocRead** function was invoked, the current file position is set to start where the read ended (see [“MedLocRead” on page 9-10](#))
- If the **MedLocWrite** function was invoked, the current file position is set to start where the write ended (see [“MedLocWrite” on page 9-15](#))

Syntax

```
void MedLocTell(mi_bitvarying *wfd, mi_int8 *offset,
               MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wfd</i>	The MedLoc_fd value specifying the large media object to access. The MedLoc_fd value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.
<i>*offset</i>	The location in which to store the retrieved file position.	Must be a positive value.

Returns

Returns no value. However, the retrieved file position is returned in the *offset* argument.

Example

See [page 9-5](#).

MedLocWrite

Writes data from a buffer to a large media object and returns the number of bytes written. The data is written starting at the current file position; see [“MedLocTell” on page 9-13](#).

Syntax

```
mi_integer MedLocWrite(mi_bitvarying *wfd, mi_char *buffer,
    mi_integer bytes, MI_FPARAM *fp)
```

Argument	Purpose	Restrictions
<i>*wfd</i>	The MedLoc_fd value specifying the large media object to access. The MedLoc_fd value is wrapped in a mi_bitvarying data structure; see “Writing VSI UDRs” on page 8-9 for more information.	N.A.
<i>*buffer</i>	Specifies the buffer from which to write.	N.A.
<i>bytes</i>	Specifies the number of bytes to write.	Must be a positive integer.

Returns

Returns an `mi_integer` value specifying the number of bytes written. This value is less than the input *bytes* if an error occurred.

Example

See [page 9-5](#).

Glossary

8 mm	Eight-millimeter-wide magnetic tape, used for professional video and audio recording. See also HI-8, S-VHS, VHS.
abstract description	A description of video content. The description can be a text annotation, a graphic image, an audio clip, or a video clip. See also proxy.
analog	Physical wave forms etched onto tape or other media. See also digital.
audio	The audible portion of a video program (frequencies between 20 Hz and 20,000 Hz).
AVI file	(Audio / Visual Interleaved) A Microsoft RIFF.
CCIR	(Comite Consultatif International des Radiocommunications) A European committee that creates and approves audio and video standards.
compensated mode	See drop-frame NTSC.
complex data type	A data type that is built from a combination of other data types using an SQL type constructor or the CREATE ROW TYPE statement, and whose components can be accessed through SQL statements. Complex data types include collection data types and row data types.

compression	The process of electronically removing picture data from a video program to decrease its size, to allow it to be sent down a particular channel, or both.
content processing	A term broadly referring to any activity—such as editing, annotating, and parsing—that video content can go through in its life cycle from raw footage, to digitized data, to final video-based product. There is always at least one output from content processing.
D1/D2/D3	Digital video recording and playback formats. D1 uses component video; D2 and D3 use composite video.
DAT	Abbreviation for <i>digital audio tape</i> .
database schema	See video database schema .
digital	Digital coding of data onto media. See also analog .
drop-frame NTSC	(drop-frame National Television Standards Committee) A standard also known as <i>compensated mode</i> , in which frames 00 and 01 are dropped at the start of each minute, except for minutes that are even multiples of 10.
DUB	A duplicate made between recording media types.
DVI	(Digital Video Interface) Computer-generated graphics and text used in video.
EBU	(European Broadcasting Union) The European television standard that includes PAL and SECAM.
encoding	The process of digitizing, then applying some sort of minimal compression, to analog data.
foundation	See video foundation .
foundation timeline	The “master” timeline, with a time unit of nanoseconds, that the Informix Video Foundation DataBlade module creates to reference multiple formats of the same video program. See also source timeline , timeline .
HI-8	A higher-definition version of the 8 mm tape format.
JPEG	(Joint Photographic Experts Group) A compression method for still images.

large media object	Any type of video, audio, or other media data that is to come under database control.
LTC	(longitudinal time code) See SMPTE timecode .
media chunk	A user-defined interval in any continuous medium, such as a video program. The interval is defined by a starting media point and an ending media point.
media component file	The large media objects that constitute a media instance. Some video formats use just one video output file per media instance; others use separate or multiple video, audio, and other output files.
media instance	A physical copy of the video program, in whole or in part, regardless of the copy's format. For example, if you have one copy of a video program on Beta-SP tape and two copies in D-1 format, you have three media instances. Similarly, if you have three segments of the program, you have three media instances.
media management system	A collection of processes, computers, and other machines that enable you to store and manipulate video (and often other media) data along with more traditional data, such as text.
media object	See large media object .
media point	A precise location in time within a video program, defined by a temporal value and an optional indexing string in the media's native format. See also timecode .
metadata	See video metadata .
MJPEG	(motion JPEG) A compression method for motion video in which individual frames are compressed using JPEG.
MPEG	(Motion Picture Experts Group) A compression method for digital video and audio.
named row data type	A row data type that is created with the CREATE ROW TYPE statement and has a name. A named row data type can be used to construct a typed table and can be part of a type or table hierarchy. See also row data type , unnamed row data type .

NTSC	(National Television Standards Committee) Creators of the black-and-white and the color television signals. See also drop-frame NTSC .
opaque data type	A fundamental data type of a predefined fixed or variable length whose internal structure is not accessible through SQL statements. Opaque data types are created with the SQL statement CREATE OPAQUE TYPE. Support functions must always be defined for opaque types.
PAL	(Phase Alternation Line) The European standard for color television and video signals.
physical attributes	Information on the physical properties of the media on which the video data is stored, such as format, frame height and width, and duration.
proxy	A description that “stands in” for a piece of data. Usually, a proxy is much smaller and is of lower resolution than the video data it represents. See also abstract description .
QuickTime (QTM)	Apple Computer’s method for recording, storing, and playing back audio and video data.
repurposing	The process of creating a new video program by recycling and recombining existing video data.
RIFF	(Resource Interchange File Format) Microsoft’s multimedia file format. See also AVI file .
row data type	A complex data type consisting of a group of ordered data elements (fields) of the same or different data types. The fields of a row type can be of any supported built-in or extended data type, including complex data types, except SERIAL and SERIAL8 and, in certain situations, TEXT and BYTE. There are two types of row data type: ■ Named row types, created using the CREATE ROW TYPE statement ■ Unnamed row types, created using the ROW constructor See also named row data type , unnamed row data type .
RTSP	(Real-Time Streaming Protocol) A proposed standard for streaming video.

schema	See video database schema .
SECAM	(Sequential Couleur Avec Memorie) A standard for color television and video signals.
SMPTE timecode	A video synchronization standard, where specific points in a video program are identified by an HH:MM:SS and frame-number timecode, developed by the Society of Motion Picture and Television Engineers. See also timecode .
source timeline	The media instance from which the Informix Video Foundation DataBlade module creates the foundation timeline.
storage component	A collection of data types and functions that provide the device-independent VSI between Informix Dynamic Server, large media objects, and storage servers.
storage server	A repository of large media objects that is under database control, whether it be a file system, a video source, or a physical storage device. Examples of storage servers are video tape recorders, the database server's file system, Web video streamers, and high-performance video servers.
strata	A meaningful group of stratum.
stratification	The process of creating layered relationships between media chunks and abstract descriptions.
stratum	A relationship between a media chunk and an abstract description.
S-VHS	(super VHS) A higher-definition version of the VHS tape format.
temporal component	A collection of data types and functions that enable Informix Dynamic Server to reference, store, and manipulate media points, timecodes, and media chunks.
timecode	A precise location in time within a video program, based on the location's SMPTE-standard timecode. See also media point .
timeline	A horizontal line divided into regularly spaced time units; the logical view of a piece of time. See also foundation timeline , media instance , source timeline .

unnamed row data type	A row type created with the ROW constructor that has no defined name and no inheritance properties. Two unnamed row types are equivalent if they have the same number of fields and if corresponding fields have the same data type, even if the fields have different names.
VCR	(video cassette recorder) A consumer-grade video player/recorder. See also VTR .
VHS	(Video Home System) A consumer-grade videotape format.
video data	Data representing an image that changes continuously in time. Unlike traditional data types that are retrieved and displayed as discrete values, video data is incrementally accessed and is displayed as a real-time stream of frames. See also video metadata .
video database schema	A set of database tables that ties together the temporal and storage components' data, optimized for storage and manipulation of video data and metadata.
video foundation	A base on which new or specialized video technologies can be quickly added or changed.
video metadata	Data about data. Video metadata provides information on the physical attributes of the video's media and abstract descriptions of the content. See also video data .
video program	The format-independent concept of the entire "piece" of video, whether it be a movie, a newscast, a video clip, or a sequence of frames.
virtual storage DataBlade module	A device-specific implementation of the VSI for a particular storage server. The Informix Video Foundation DataBlade module provides three such implementations: VSI_FILE, VSI_CLIENT, and VSI_OTHER.
Virtual Storage Interface (VSI)	The interface between Informix Dynamic Server, large media objects, and storage servers in a media management system.
VTR	(video tape recorder) A studio-grade video player/recorder. See also VCR .

Error Messages

The following table lists the error messages that the Informix Video Foundation DataBlade module can raise. For each error message, the following information is shown:

- The error code
Error codes that start with the string *UMED* are raised by this DataBlade module; those with the *UGEN* string can be raised by any DataBlade module.
- The error string that can be displayed when this error is raised
- An explanation of why or by which type of function this error was raised and, where appropriate, suggestions on what you can do to avoid this error in the future

For information on error handling, consult the *Informix Guide to SQL* manuals and the manual for the Informix programming interface you are using.

Error Code	Error String	Explanation and Suggestions
UGEN1	Connection has failed in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by all VSI UDRs (see Chapter 3, 8, and 9) or by any other DataBlade module.
UGEN2	Memory allocation has failed in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by all VSI UDRs (see Chapter 3, 8, and 9) or by any other DataBlade module.
UGENB	Input data format error in %FUNCTION%	The input value does not conform to the syntax for a MedLoc value's external representation. (Can also be raised by any other DataBlade module.)
UMEDO	Addition of int8	Cannot add two MedPtr values if the resulting <i>Time</i> element is larger than an 8-byte integer.
UMEDO	ASCII to int8	Cannot convert at least one of the MedPtr values' ASCII <i>Time</i> elements (nanoseconds) into an 8-byte integer. Possible reasons are: alpha characters in the <i>Time</i> element or number larger than 2^{64} .
UMEDO	double precision to int8	Cannot convert the double-precision frame number divided by frames-per-second rate into an 8-byte integer for a MedPtr value. A possible reason is that the result is too large.
UMEDO	frame number larger than INT_MAX	Cannot convert a MedPtr value to a frame number: the resulting number exceeds the maximum for this platform.
UMEDO	int8 to ASCII	Cannot convert at least one of the MedPtr values' 8-byte <i>Time</i> elements (nanoseconds) into an ASCII value.
UMEDO	int8 to double precision	Cannot convert the MedPtr value's 8-byte <i>Time</i> element (nanoseconds) into a double-precision value for a frame number.
UMEDO	int8 to double precision	In attempting to get the size of a MedChunk value, cannot convert the difference between the ending MedPtr value and the starting MedPtr value (nanoseconds) into a double-precision value.

(1 of 5)

Error Code	Error String	Explanation and Suggestions
UMEDO	Multiplication of int8	Cannot convert at least one of the MedPtr values' ASCII <i>Time</i> elements into nanoseconds. A possible reason is that the result is greater than an 8-byte integer.
UMEDO	Subtraction of int8	In attempting to get the size of a MedChunk value by subtracting the starting MedPtr value from the ending MedPtr value, a math error occurred.
UMEDO	Subtraction of int8	In attempting to subtract the first MedPtr value from the second, a math error occurred.
UMEDO	frame rates must be the same	MedTC values with different frame rates cannot be added, subtracted, or paired as a MedChunk value. Convert the values to another data type, such as frame numbers, before calling the function.
UMEDO	frame number must be positive	In converting another data type value to a MedTC value, a negative frame number cannot be converted into a SMPTE timecode.
UMEDO	double precision to int8	In converting a MedTC value to another data type value, the timecode has been successfully converted to an integer frame number, but the conversion to an int8 nanoseconds value was <i>not</i> successful.
UMED1	Input string was too short	The input string to create a MedPtr value is a NULL string (0 length).
UMED2	Unexpected character after first component	At least one of the MedPtr values' <i>Time</i> elements is followed either by a character other than the hyphen character (-) or by the string terminator \0.
UMED2	Frame number must be positive	A negative frame number cannot be converted into a MedPtr value.
UMED2	'-' without native string	A hyphen character (-) appears after the MedPtr value's <i>Time</i> element, but no <i>Native</i> element is supplied. (If the hyphen character is supplied, a <i>Native</i> string must also be supplied.)
UMED2	Must start with number	<i>Time</i> elements must start with a positive number.

(2 of 5)

Error Code	Error String	Explanation and Suggestions
UMED2	Start is greater than the end	The ending MedPtr value of the MedChunk value must be greater than the starting MedPtr value. Look at the <i>Unitflag</i> elements of the MedPtr values to see if the C, M, N, or S causes the starting MedPtr value to be greater than the second.
UMED2	expected exactly 11 or 13 characters	One of the MedTC values is invalid: if the <i>Standard_flag</i> element is not supplied, the timecode must be 11 characters long. If a flag is supplied, the timecode must be 13 characters long.
UMED2	expected -	One of the MedTC values is invalid: if the timecode is 13 characters long, character 12 must be the hyphen character (-).
UMED2	expected one of P, D, N, F	One of the MedTC values is invalid: if the timecode is 13 characters long, character 13 must be one of the valid video standard flags.
UMED2	expected one of ; or . or :	One of the MedTC values is invalid: at least one of the delimiter characters is invalid (character 2, 5, or 8).
UMED2	expected a digit	One of the MedTC values is invalid: at least one of the characters between the delimiter characters is not a digit.
UMED2	Time must be positive	To convert a <i>Time</i> and <i>Native</i> string into a MedPtr value, the 8-byte <i>Time</i> element must be a positive number.
UMED3	Resulting time was less than zero	A MedPtr value cannot be negative; thus, the second MedPtr value must be less than the first.
UMED3	num. of frames & fps mismatch	One of the MedTC values is out of range: the frame number is negative or is higher than the video standard allows (as specified by the <i>Standard_flag</i> element).
UMED3	must be between 0 and 99:59:59:ff	One of the MedTC values is out of range: at least one of the <i>HH</i> , <i>MM</i> , or <i>SS</i> elements has an invalid value.
UMED3	second argument must be less than first	A MedTC value cannot be negative; thus, a larger timecode cannot be subtracted from a smaller one.

(3 of 5)

Error Code	Error String	Explanation and Suggestions
UMED3	too many frames for the given fps or invalid fps	In converting another data type value to a MedTC value, the maximum frame number was exceeded. In SMPTE timecodes, valid frames-per-second rates are 24.0, 25.0, 29.97, and 30.0. The maximum cumulative number of frames per frame rate are respectively 8,640,000; 9,000,000; 10,789,200; and 10,800,000. For example, you cannot convert a maximum cumulative number of frames of 9,000,001 to a 25.0 fps timecode because that is an invalid number for that frame rate.
UMED3	result must be less than 100 hours	In converting another data type to a MedTC value, the maximum <i>HH</i> element was exceeded. A valid SMPTE timecode's <i>HH</i> element is in the range 0 to 99.
UMED3	not a valid timecode frame rate for conversion	In SMPTE timecodes, valid frames-per-second rates are 24.0, 25.0, 29.97, and 30.0.
UMED4	00 and 01 frame numbers on every 10th minute only	For drop-frame NTSC MedTC values, only <i>MM</i> elements that are multiples of 10 can have a frame 00 or frame 01.
UMEDE	Result of select is not a single value in %FUNCTION%	The MedVSIRegister table has more than one row with the same Type and StorageID column values.
UMEDF	Function mi_get_result returned error in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by all VSI UDRs that dispatch a storage-server-specific UDR (see Chapter 8) and the Storage function.
UMEDH	Function mi_exec returned error in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by all VSI UDRs that dispatch a storage-server-specific UDR (see Chapter 8), the Assign function (VSI_FILE implementation), and the Storage function.
UMEDI	Function mi_routine_exec failed in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by all VSI UDRs that dispatch a storage-server-specific UDR (see Chapter 8).

(4 of 5)

Error Code	Error String	Explanation and Suggestions
UMEDJ	Attempt to access unmanaged storage via %FUNCTION%	The VSI_OTHER implementation does not allow access to the storage server.
UMEDK	No row in table for %FUNCTION%	The MedVSIRegister table does not have a row with the specified Type and StorageID column values.
UMEDM	File open failed in %FUNCTION%	MedLocOpen source file does not exist or is not readable.
UMEDN	Function mi_file_unlink failed in %FUNCTION%	Diagnostic information for rare, unexpected errors in the database server. Can be raised by the Destroy procedure.
UMEDO	File operation failed in %FUNCTION%	Access to open file failed.
UMEDP	8-byte integer range not supported in %FUNCTION%	Seek offset too large for the specified implementation of the MedLocSeek function.
UMEDR	Illegal argument in %FUNCTION%	The MedLocSeek function's <i>whence</i> argument must be one of MED_SEEK_SET, MED_SEEK_CUR, MED_SEEK_END.
UMEDS	Result of select is NULL in %FUNCTION%	Column value is NULL (in either the MedVSIRegister table, the MedVSI_File table, or both).
UMEDT	Function not supported: %FUNCTION%	Some virtual storage DataBlade modules do not implement all VSI UDRs.
UMEDU	Could not create file in %FUNCTION%	Attempt to create new file failed.

(5 of 5)

Index

A

Abstract descriptions 1-3, 1-9, 1-13, 4-3, 4-20 to 4-31, 7-8
See also Subject matter tables.

Accessing data. *See* ClientAccess function.

Accessing, data type elements
 MedChunk 5-28, 5-98, 6-9, 6-13
 MedLoc 5-48, 5-106, 5-110, 6-26, 6-28, 6-30, 7-15
 MedPtr 5-71, 5-108, 6-32, 6-34
 MedTC 5-34, 5-40, 5-69, 5-92, 5-112, 6-52, 6-54, 6-56, 6-58, 6-60

Adding data type values 5-89, 6-19, 6-38, 6-64

Additional documentation
 Intro-9 to Intro-11

After function 2-17, 5-5

AfterOrEqual function 2-17, 5-7

Annotations, text. *See* VidTextDesc table.

API. *See* Application programming interface (API).

Application developers 1-17

Application programming interface (API)
 Client 1-16, 6-3 to 6-71
 VSI 9-3 to 9-15

Application, client 1-4

Assign function 8-6, 8-12

Attributes, physical 1-3, 1-9, 4-3, 4-7 to 4-19, 7-5

Audience Intro-5

Audio clips 4-21, 4-24

Audio compression formats 4-35

Audio output files. *See* Large media objects; VidPAudio table.

B

Before function 2-17, 5-9

BeforeOrEqual function 2-17, 5-11

Boldface type Intro-6

C

Calculating MedPtr data type value 4-12

Chunks, media 1-7, 2-12, 4-20, 4-22, 7-8, 7-12 to 7-14
See also MedChunk data type; VidChunkMD table.

Client API 1-16, 6-3 to 6-71

Client application 1-4

ClientAccess function 3-9, 5-13, 7-15, 8-6, 8-11

Clips
 audio 4-21, 4-24
 video 4-21, 4-30

Close function 8-6

Closing a large media object 9-7

Compare function 2-6, 2-11, 2-15, 5-16

Comparing
 data type values 5-16, 5-30, 5-36, 5-38, 5-44, 5-46, 5-79
 MedChunk values 5-5, 5-7, 5-9, 5-11, 5-20, 5-23, 5-62, 5-75, 5-77, 5-81, 5-84, 5-86, 7-13

Compensated mode. *See* Drop-frame NTSC video standard.

Complex data types 3-4
Component files, media 1-10, 4-7, 4-14, 7-7
Compression formats 4-35, 4-37, 7-7
ConnectionString field, MedStorage data type 3-5
ConnectionString field, MedStorage data type 3-5
Contained function 2-16, 5-20
Contains function 2-16, 5-23
Content processing 1-18, 4-12
Content, video 1-3, 1-9, 4-3, 7-8
Conventions
 function syntax Intro-9
 icon Intro-8
 naming Intro-8
 typographical Intro-6
Conversion functions 2-18
Converting
 to frame number 2-18, 5-114, 6-40, 6-66
 to MedChunk data type 2-18, 5-50, 6-43, 6-47, 7-9, 7-11
 to MedPtr data type 2-18, 5-53, 6-22, 6-68, 6-70, 7-6, 7-9, 7-11
 to MedTC data type 2-18, 5-59, 6-24, 6-45, 7-16
Copy function 3-9, 3-12, 5-26, 7-7, 8-6, 8-13
Copying data 5-26
Creating
 data types 5-64, 5-89, 6-15, 6-19, 6-36, 6-38, 6-62, 6-64
 MedChunk data type 5-42
 MedLoc value 5-73
 MedStorage data type 5-57
 See also Converting.
Current file position 9-11, 9-13
C++ value objects 6-4

D

Data
 copying 5-26
 format-independent referencing of 1-12
 requesting access to 5-13, 7-15

security 1-8
video 1-3, 1-4
Data element, MedLoc_fd data type 8-14, 9-4
Data types
 comparing values of 5-16, 5-30, 5-36, 5-38, 5-44, 5-46, 5-62, 5-79
 complex 3-4
 converting 2-18, 5-50, 5-53, 5-59, 6-22, 6-24, 6-43, 6-45, 6-47, 6-68, 6-70, 7-6, 7-9, 7-11, 7-16
 creating 5-42, 5-64, 5-89, 6-15, 6-19, 6-36, 6-38, 6-62, 6-64
 MedChunk 5-4, 5-5, 5-7, 5-9, 5-11, 5-20, 5-23, 5-75, 5-77, 5-81, 5-84, 5-86, 7-9, 7-11, 7-13
 in client API 6-4
 client functions for 6-6
 external representation of 2-12
 functions for 2-13 to 2-18
 See also VidChunkMD table.
 in stratums 4-20
MedLoc 5-4, 5-73
 in client API 6-4
 client functions for 6-6
 for copying 5-26
 direct access to 5-13
 functions for 3-9
 in VSI UDRs 8-12
MedLoc_fd 8-14, 9-3, 9-8
MedPtr 4-12, 4-20, 5-3, 7-6, 7-9, 7-11
 in client API 6-4
 client functions for 6-6
 functions for 2-5 to 2-7
 in VSI UDTs 8-4
MedStorage 5-4, 5-94, 5-100, 5-102, 5-104
 for copying 5-26
 functions for 3-7
MedTC 5-3, 5-32, 6-49, 7-16
 in client API 6-4
 client functions for 6-6
 functions for 2-9 to 2-12
 in VSI UDTs 8-4
named row 3-4
opaque 2-3, 6-4, 6-6
prefixes Intro-8
row 3-4

supplying your own 8-5
Data types, accessing
 MedChunk elements 5-28, 5-98, 6-9, 6-13
 MedLoc elements 5-48, 5-106, 5-110, 6-26, 6-28, 6-30, 7-15
 MedPtr elements 5-71, 5-108, 6-32, 6-34
 MedTC elements 5-34, 5-40, 5-69, 5-92, 5-112, 6-52, 6-54, 6-56, 6-58, 6-60
Database 1-4
DataBlade modules
 developers of 1-17, 1-18
 registering with foundation 7-4, 8-14
 registering with Informix 8-4
Delimiter characters
 MedChunk data type 2-12
 MedLoc data type 3-8
 MedPtr data type 2-5
 MedTC data type 2-8
Descriptions, abstract 1-3, 1-9, 1-13, 4-3, 4-20 to 4-31, 7-8
Destroy procedure 8-6, 8-14
Developers
 application 1-17
 content-processing DataBlade module 1-18
 virtual storage DataBlade module 1-17, 8-3 to 8-18
Device-independent VSI
 functions 3-4, 8-6
Device-specific VSI
 functions 8-7 to 8-16
 prototypes 8-10
Device, storage. *See* MedStorage data type.
Documentation notes Intro-10
Documentation,
 additional Intro-9 to Intro-11
Documenting virtual storage
 DataBlade modules 8-17
Drop-frame NTSC video
 standard 2-7, 2-8
Dynamic Server 1-4

E

EndP element, MedChunk data type 5-28, 6-9
 EndP function 2-13, 5-28
 Environment variables Intro-6
 Equal function 2-6, 2-11, 2-14, 5-30
 Examples
 ClientAccess function 7-15
 Copy function 7-7
 Intersection function 7-14
 inventory tables 4-8, 7-5
 Location function 7-15
 MedChunk data type 2-13, 5-4
 MedChunkFrom function 7-9, 7-11
 media management system 7-3 to 7-16
 MedLoc data type 3-8, 5-4
 MedPtr data type 2-5, 5-3
 MedPtrFrom function 7-6, 7-9, 7-11
 MedStorage data type 3-6, 5-4, 7-4
 MedStorage function 7-4
 MedTC data type 2-9, 5-3
 MedTCFrom function 7-16
 MedVSIRegister table 3-6, 3-10, 7-4
 MedVSI_File table 3-11, 7-4
 Minus function 7-14
 nondigital multimedia assets 7-15
 Overlap function 7-13, 7-14
 Plus function 7-14
 Sample Company 3-4 to 3-6, 7-3 to 7-16, 8-4 to 8-18
 setting up 5-3, 5-4, 7-4
 subject matter tables 4-21, 7-8
 TapeLocation example table 7-16
 VidChunkMD table 7-9, 7-11, 7-14, 7-16
 Video Foundation DataBlade module 7-3 to 7-16
 VidFormat table 7-6
 VidMDGeneral table 7-5, 7-6, 7-15
 VidMedia table 7-6, 7-7, 7-15, 7-16
 VidPVideo table 7-7, 7-15

VidStandard table 7-6
 VidStrata table 7-8, 7-14, 7-16
 VidTextDesc table 7-9, 7-11, 7-14, 7-16
 VidVCodec table 7-7
 VSI API 9-5
 VSI_CLIENT MedLoc 3-12
 VSI_FILE MedLoc 3-10, 7-4
 VSI_OTHER MedLoc 3-14, 7-15
 External representation
 MedChunk data type 2-12
 MedLoc data type 3-8
 MedPtr data type 2-4
 MedStorage data type 3-5
 MedTC data type 2-8

F

FF element, MedTC data type 2-8, 5-34, 6-52
 Fields, MedStorage data type 3-5
 File position 9-11, 9-13
 Files
 audio output. *See* Large media objects; VidPAudio table.
 media component 1-10, 4-7, 4-14, 7-7
 video output. *See* Large media objects; VidPVideo table.
 Film 2-9
 Format-independent referencing of video data 1-12
 Formats
 compression 4-35, 4-37, 7-7
 video 4-34, 7-6
 Foundation timeline 1-12, 4-11
 Foundation, video 1-4
 Fps function 2-11, 5-32
 Fps. *See* Frames-per-second (fps) rate.
 Frame function 2-10, 5-34
 Frame number 5-34, 6-52
 converting to 2-18, 5-114, 6-40, 6-66
 See also FF element, MedTC data type.
 Frames-per-second (fps) rate 5-32, 6-49

Functions

After 2-17, 5-5
 AfterOrEqual 2-17, 5-7
 Assign 8-6, 8-12
 Before 2-17, 5-9
 BeforeOrEqual 2-17, 5-11
 Client API 6-3 to 6-71
 ClientAccess 3-9, 5-13, 7-15, 8-6, 8-11
 Close 8-6
 Compare 2-6, 2-11, 2-15, 5-16
 Contained 2-16, 5-20
 Contains 2-16, 5-23
 conversion 2-18
 Copy 3-9, 3-12, 5-26, 7-7, 8-6, 8-13
 device-independent VSI 3-4, 8-6
 device-specific VSI 8-7 to 8-16
 EndP 2-13, 5-28
 Equal 2-6, 2-11, 2-14, 5-30
 Fps 2-11, 5-32
 Frame 2-10, 5-34
 GreaterThan 2-7, 2-11, 5-36
 GreaterThanOrEqual 2-7, 2-11, 5-38
 Hour 2-10, 5-40
 Intersection 2-15, 5-42, 7-14
 LessThan 2-6, 2-11, 5-44
 LessThanOrEqual 2-6, 2-11, 5-46
 Location 3-9, 5-48, 7-15
 MedChunkFrom 2-14, 2-18, 5-50, 7-9, 7-11
 medChunkGetEndPtr 6-9
 medChunkGetSize 6-11
 medChunkGetStartPtr 6-13
 MedChunkIn 2-18
 medChunkMinus 6-15
 MedChunkOut 2-18
 medChunkPlus 6-19
 medFrameNumberToMedPtr 6-22
 medFrameNumberToMedTC 6-24
 MedLocClose 9-7
 medLocGetLocation 6-26
 medLocGetStorageID 6-28
 medLocGetType 6-30
 MedLocOpen 9-8
 MedLocRead 9-10
 MedLocWrite 9-15

MedPtrFrom 2-6, 2-11, 2-18, 5-53,
 7-6, 7-9, 7-11, 8-5
 medPtrGetNative 6-32
 medPtrGetTime 6-34
 MedPtrIn 2-7
 medPtrMinus 6-36
 MedPtrOut 2-7
 medPtrPlus 6-38
 medPtrsToMedChunk 6-47
 medPtrToFrameNumber 6-40
 medPtrToMedChunk 6-43
 medPtrToMedTC 6-45
 MedStorage 3-6, 3-7, 5-57, 7-4
 MedTCFrom 2-18, 5-59, 7-16
 medTCGetFPS 6-49
 medTCGetFrame 6-52
 medTCGetHour 6-54
 medTCGetMinute 6-56
 medTCGetSecond 6-58
 medTCGetType 6-60
 MedTCIn 2-12
 medTCMinus 6-62
 MedTCOut 2-12
 medTCPlus 6-64
 medTCToFrameNumber 6-66
 medTCToMedPtr 6-68
 medTimeNativeToMedPtr 6-70
 Meets 2-15, 5-62
 Minus 2-6, 2-11, 2-14, 5-64, 7-14
 Minute 2-10, 5-69
 Native 2-6, 5-71
 New 3-9, 5-73, 8-6, 8-13
 NotContained 2-17, 5-75
 NotContains 2-16, 5-77
 NotEqual 2-6, 2-11, 2-14, 5-79
 NotMeets 2-15, 5-81
 NotOverlap 2-16, 5-84
 Open 8-6
 Overlap 2-15, 5-86, 7-13, 7-14
 Plus 2-6, 2-11, 2-14, 5-89, 7-14
 prefixes Intro-8
 Read 8-6
 Second 2-10, 5-92
 server 5-3 to 5-116
 Shutdown 3-7, 5-94, 8-7
 Size 2-13, 5-96
 StartP 2-13, 5-98
 Startup 3-7, 5-100, 8-7
 Status 3-7, 5-102, 8-7

Storage 3-9, 5-104
 StorageID 3-9, 5-106
 syntax conventions Intro-9
 Time 2-6, 5-108
 Type (MedLoc) 3-9, 5-110
 Type (MedTC) 2-10, 5-112
 VidFrameNumberFrom 2-18,
 5-114
 VSI 8-5 to 8-16
 VSI API 9-4 to 9-15
 Write 8-7
See also Procedures.

G

Getting current file position 9-13
 GreaterThan function 2-7, 2-11,
 5-36
 GreaterThanOrEqual function 2-7,
 2-11, 5-38

H

HH element, MedTC data type 2-8,
 5-40, 6-54
 Hour function 2-10, 5-40

I

Icon conventions Intro-8
 Image proxies 4-21, 4-26
 Implementations, VSI
 using 3-10 to 3-14
 VSI_CLIENT 3-12
 VSI_FILE 3-10, 7-4
 VSI_OTHER 3-13, 7-15
 Indexing. *See* Native element,
 MedPtr data type.
 Informix Dynamic Server 1-4
 Instances, media 1-10, 4-7, 4-11, 7-5
 See also VidMedia table.
 Intersection function 2-15, 5-42,
 7-14
 Intervals. *See* MedChunk data type.
 Inventory control, video 1-10, 4-7,
 7-5

Inventory tables 1-9, 4-4,
 4-7 to 4-19, 7-5
 examples of 4-8
 ITessential methods 6-4
 ITMedChunk value object 6-4
 ITMedLoc value object 6-4
 ITMedPtr value object 6-4
 ITMedTC value object 6-4
 ITValue methods 6-4

L

Large media objects
 closing 9-7
 as components 4-7, 4-8
 data type for stored 3-7
 defined 1-8, 3-3
 in interfaces 9-3 to 9-15
 as media instances 4-14 to 4-19
 opening 9-8
 reading 9-10
 writing 9-15
 See also VidPAudio table;
 VidPOther table; VidPVideo
 table.
 LastNum column, MedVSI_File
 table 3-11
 LessThan function 2-6, 2-11, 5-44
 LessThanOrEqual function 2-6,
 2-11, 5-46
 Library
 Client API 6-3
 VSI API 9-3
 Location element, MedLoc data
 type 3-8, 5-48, 6-26, 7-15, 8-12
 Location function 3-9, 5-48, 7-15
 Lookup tables 4-4, 4-32 to 4-37

M

Machine notes Intro-10
 Med prefix Intro-8
 MedChunk data type
 in client API 6-4
 client functions for 6-6
 comparing 5-5, 5-7, 5-9, 5-11, 5-20,
 5-23, 5-62, 5-75, 5-77, 5-81, 5-84,
 5-86, 7-13

- converting to 2-18, 5-50, 6-43, 6-47, 7-9, 7-11
- creating 5-42
 - See also* Converting.
- delimiter character 2-12
- EndP element 5-28, 6-9
- examples setup 5-4
- external representation of 2-12
- functions for 2-13 to 2-18
- size 2-13, 5-96
- StartP element 2-12, 5-98, 6-13
- in stratum 4-20
- See also* VidChunkMD table.
- MedChunkFrom function 2-14, 2-18, 5-50, 7-9, 7-11
- medChunkGetEndPtr function 6-9
- medChunkGetSize function 6-11
- medChunkGetStartPtr function 6-13
- MedChunkIn function 2-18
- medChunkMinus function 6-15
- MedChunkOut function 2-18
- medChunkPlus function 6-19
- medFrameNumberToMedPtr function 6-22
- medFrameNumberToMedTC function 6-24
- Media
 - chunks 1-7, 2-12, 4-20, 4-22, 7-8, 7-12 to 7-14
 - See also* MedChunk data type; VidChunkMD table.
 - component files 1-10, 4-7, 4-14, 7-7
 - See also* Large media objects.
 - instances 1-10, 4-7, 4-11, 7-5
 - See also* VidMedia table.
 - management system 1-4, 7-3 to 7-16
 - non-digital example 7-15
 - objects, large
 - data type for stored 3-7
 - defined 1-8, 3-3
 - in instances 4-14 to 4-19
 - in interfaces 9-3 to 9-15
 - See also* VidPAudio table; VidPOther table; VidPVideo table.
 - offset 4-12
 - physical attributes 4-3, 4-7 to 4-19, 7-5
 - points 1-7, 2-3, 5-28, 5-98, 6-9, 6-13, 8-4
 - See also* MedPtr data type.
 - timecode. *See* MedTC data type.
 - objects, large
 - as components 4-7, 4-8
- MedLoc data type
 - in client API 6-4
- client functions for 6-6
- for copying 5-26
- creating 5-73
- delimiter character 3-8
- direct access to 5-13
- examples of 3-8
- examples setup 5-4
- external representation of 3-8
- functions for 3-9
- Location element 3-8, 5-48, 6-26, 7-15, 8-12
- opening 9-8
- StorageID element 3-8, 5-106, 6-28
- Type element 3-8, 5-110, 6-30, 8-12
 - in VSI UDRs 8-12
- MedLocClose function 9-7
- medLocGetLocation function 6-26
- medLocGetStorageID function 6-28
- medLocGetType function 6-30
- MedLocOpen function 9-8
- MedLocRead function 9-10
- MedLocSeek procedure 9-11
- MedLocTell procedure 9-13
- MedLocWrite function 9-15
- MedLoc_fd data type
 - Data element 8-14, 9-4
 - in interfaces 9-3, 9-8
 - Type element 8-14, 9-4
- MedPtr data type
 - calculating value 4-12
 - in client API 6-4
 - client functions for 6-6
 - converting to 2-18, 5-53, 6-22, 6-68, 6-70, 7-6, 7-9, 7-11
 - delimiter character 2-5
 - examples of 2-5
 - examples setup 5-3
 - external representation of 2-4
 - functions for 2-5 to 2-7
 - Native element 2-5, 4-20, 5-71, 6-32
 - pipe character 2-5, 2-12
 - Time element 2-4
 - Unitflag element 2-4
 - in VSI UDTs 8-4
- MedPtrFrom function 2-6, 2-11, 2-18, 5-53, 7-6, 7-9, 7-11, 8-5
- medPtrGetNative function 6-32
- medPtrGetTime function 6-34
- MedPtrIn function 2-7
- medPtrMinus function 6-36
- MedPtrOut function 2-7
- medPtrPlus function 6-38
- medPtrsToMedChunk function 6-47
- medPtrToFrameNumber function 6-40
- medPtrToMedChunk function 6-43
- medPtrToMedTC function 6-45
- MedStorage data type
 - ConnectionName field 3-5
 - ConnectionString field 3-5
 - for copying 5-26
 - creating 5-57
 - example of 7-4
 - examples setup 5-4
 - external representation of 3-5
 - fields 3-5
 - functions for 3-7
 - Model field 3-5
 - Password field 3-5
 - retrieving row value 5-104
 - shutting down 5-94
 - starting 5-100
 - status of 5-102
 - StorageID field 3-5
 - Type field 3-5
 - UserID field 3-5
 - Vendor field 3-5
- MedStorage function 3-6, 3-7, 5-57, 7-4
- MedTC data type
 - in client API 6-4
 - client functions for 6-6

converting to 2-18, 5-59, 6-24, 6-45, 7-16
 delimiter character 2-8
 examples of 2-9
 examples setup 5-3
 external representation of 2-8
 FF element 2-8, 5-34, 6-52
 frames-per-second rate 5-32, 6-49
 functions for 2-9 to 2-12
 HH element 2-8, 5-40, 6-54
 MM element 2-8, 5-69, 6-56
 SS element 2-8, 5-92, 6-58
 Standard_flag element 2-8, 5-112, 6-60
 in VSI UDTs 8-4
 MedTCFrom function 2-18, 5-59, 7-16
 medTCGetFPS function 6-49
 medTCGetFrame function 6-52
 medTCGetHour function 6-54
 medTCGetMinute function 6-56
 medTCGetSecond function 6-58
 medTCGetType function 6-60
 MedTCIn function 2-12
 medTCMinus function 6-62
 MedTCOut function 2-12
 medTCPlus function 6-64
 medTCToFrameNumber function 6-66
 medTCToMedPtr function 6-68
 medTimeNativeToMedPtr function 6-70
 MedVSI prefix Intro-9
 MedVSIFunctions table 8-14
 MedVSISRegister table 3-6, 3-10, 5-104, 7-4
 MedVSI_File table 3-10, 3-11, 7-4
 MED_APPEND flag 9-9
 MED_READ flag 9-8
 MED_SEEK flag 9-9
 MED_SEEK_CUR flag 9-11
 MED_SEEK_END flag 9-11
 MED_SEEK_SET flag 9-11
 MED_TRUNC flag 9-9
 MED_WRITE flag 9-8
 Meets function 2-15, 5-62
 Metadata 1-3, 1-4, 4-3
 Methods
 ITEssential 6-4

ITValue 6-4
 Metropolis movie, example multimedia asset 7-3
 Minus function 2-6, 2-11, 2-14, 5-64, 7-14
 Minute function 2-10, 5-69
 MM element, MedTC data type 2-8, 5-69, 6-56
 Model field, MedStorage data type 3-5

N

Named row data types 3-4
 Naming conventions Intro-8
 National Television Standards Committee (NTSC) video standard 2-7, 2-9
See also Drop-frame NTSC video standard.
 Native element, MedPtr data type 2-5, 4-20, 5-71, 6-32
 Native function 2-6, 5-71
 New function 3-9, 5-73, 8-6, 8-13
 NotContained function 2-17, 5-75
 NotContains function 2-16, 5-77
 NotEqual function 2-6, 2-11, 2-14, 5-79
 NotMeets function 2-15, 5-81
 NotOverlap function 2-16, 5-84

O

Objects, large media
 as components 4-7, 4-8
 data type for stored 3-7
 defined 1-8, 3-3
 in instances 4-14 to 4-19
 in interfaces 9-3 to 9-15
See also VidPAudio table;
 VidPOther table; VidPVideo table
 Objects, value 6-4
 Offset, media 4-12
 On-line files Intro-10
 Opaque data types 2-3, 6-4, 6-6
 Open function 8-6
 Opening a large media object 9-8

Output files
 audio. *See* Large media objects;
 VidPAudio table.
 video. *See* Large media objects;
 VidPVideo table.
 Overlap function 2-15, 5-86, 7-13, 7-14

P

PAL/SECAM video standard 2-9
 Password field, MedStorage data type 3-5
 Path
 MedVSI_File table column 3-11
 Physical attributes 1-3, 1-9, 4-3, 4-7 to 4-19, 7-5
 Pipe character in data types 2-5, 2-12
 Plus function 2-6, 2-11, 2-14, 5-89, 7-14
 Points, media 1-7, 2-3
See also MedPtr data type.
 Prefixes Intro-8, 8-4
 Procedures
 Destroy 8-6, 8-14
 MedLocSeek 9-11
 MedLocTell 9-13
 Seek 8-7
 Tell 8-7
 Program, video 1-10, 4-7, 4-10, 7-5
See also VidMDGeneral table.
 Prototypes, VSI functions 8-10
 Proxies. *See* Abstract descriptions.

Q

Querying video database
 schema 7-12 to 7-14

R

Read function 8-6
 Reading a large media object 9-10
 Referencing video data 1-12
 Registering

DataBlade modules with Informix 8-4
virtual storage DataBlade modules with foundation 7-4, 8-14
Related reading Intro-9 to Intro-11
Release notes Intro-10
Requesting
 data access 5-13, 7-15
 new MedLoc value 5-73
Retrieving MedStorage row value 5-104
Row data types 3-4

S

Sample Company
 examples 3-4 to 3-6, 7-3 to 7-16, 8-4 to 8-18
Sample uses of Video Foundation DataBlade module 1-20 to 1-22
schema. *See* Video database schema.
Second function 2-10, 5-92
Security of data 1-8
Seek procedure 8-7
Seeking in current file 9-11
Segments. *See* MedChunk data type.
Selecting values from video database schema 7-12 to 7-14
Servers
 database 1-4
 functions 5-3 to 5-116
 storage 1-4, 1-8, 3-3, 3-4, 5-110, 6-30, 7-4
 See also MedStorage data type.
 video 1-8
Setting current file position 9-11
Setting up examples 5-3, 5-4, 7-4
Shutdown function 3-7, 5-94, 8-7
Size function 2-13, 5-96
SMPTE timecode. *See* MedTC data type.
Source timeline 1-12, 4-11, 7-6
SS element, MedTC data type 2-8, 5-92, 6-58
Standards, video 5-112
 See also VidStandard table.

Standard_flag element, MedTC data type 2-8, 5-112, 6-60
StartP element, MedChunk data type 2-12, 5-98, 6-13
StartP function 2-13, 5-98
Startup function 3-7, 5-100, 8-7
Status function 3-7, 5-102, 8-7
Storage column, MedVSIRegister table 3-6
Storage component 1-7, 3-3 to 3-14
Storage function 3-9, 5-104
Storage servers
 data type for 3-4
 purpose of 1-4, 1-8, 3-3
 registering 7-4
 shutting down 5-94
 starting 5-100
 status of 5-102
 type of 5-110, 6-30
 See also MedStorage data type.
StorageID
 MedLoc data type element 3-8, 5-106, 6-28
 MedStorage data type field 3-5
StorageID function 3-9, 5-106
Strata 1-13, 4-3, 4-31, 7-8
 See also VidStrata table.
Stratification 1-13, 4-20
 See also Strata.
Stratum 1-13, 4-20
Streamers, Web video 1-8
Subject matter tables 1-13, 4-4, 4-20 to 4-31, 7-8
Subtracting data type values 5-64, 6-15, 6-36, 6-62
Supplying user-defined data types 8-5
System, media management 1-4, 7-3 to 7-16

T

Tables
 inventory 4-4, 4-7 to 4-19, 7-5
 lookup 4-4, 4-32 to 4-37
 MedVSIFunctions 8-14
 MedVSIRegister 3-6, 3-10, 5-104, 7-4

MedVSI_File 3-10, 3-11, 7-4
schema 1-9
subject matter 4-4, 4-20 to 4-31, 7-8
TapeLocation example 7-16
VidACodec 4-35
VidAudDesc 4-21, 4-24
VidChunkMD 4-20, 4-22, 7-9, 7-11, 7-14, 7-16
VidFormat 4-34, 7-6
VidImgDesc 4-21, 4-26
VidMDGeneral 4-7, 4-10, 7-5, 7-6, 7-15
VidMedia 4-7, 4-11, 7-6, 7-7, 7-15, 7-16
VidPAudio 4-8, 4-17
VidPOther 4-8, 4-19
VidPVideo 4-8, 4-14, 7-7, 7-15
VidStandard 4-32, 7-6
VidStrata 4-21, 4-31, 7-8, 7-14, 7-16
VidTextDesc 4-21, 4-28, 7-9, 7-11, 7-14, 7-16
VidVCodec 4-37, 7-7
VidVidDesc 4-21, 4-30
TapeLocation example table 7-16
Tell procedure 8-7
Telling current file position 9-13
Temporal component 1-7, 2-3 to 2-18
Temporal value 2-3
Text annotations. *See* VidTextDesc table.
Time element, MedPtr data type 2-4, 5-108, 6-34
Time function 2-6, 5-108
Timecodes 2-7, 8-4
 See also MedTC data type.
Timelines 1-12, 4-11, 7-6
Type
 MedLoc data type element 3-8, 8-12
 MedLoc_fd data type element 8-14, 9-4
 MedStorage data type field 3-5
Type (MedLoc) function 3-9, 5-110
Type (MedTC) function 2-10, 5-112
Types of users Intro-5
Typographical conventions Intro-6

U

Unitflag element, MedPtr data type 2-4
 User-defined data types (UDTs). *See* Data types.
 User-defined routines (UDRs). *See* Functions.
 UserID field, MedStorage data type 3-5
 Users, types Intro-5, 1-17

V

Value objects 6-4
 Value, temporal 2-3
 Vendor field, MedStorage data type 3-5
 Vid prefix Intro-8
 VidACodec table 4-35
 VidAudDesc table 4-21, 4-24
 VidChunkMD table 4-20, 4-22, 7-9, 7-11, 7-14, 7-16
 Video
 abstract descriptions 1-3, 1-9, 1-13, 4-3, 4-20 to 4-31, 7-8
 application developers 1-17
 clips 4-21, 4-30
 compression formats 4-37, 7-7
 content 1-3, 1-9
 data 1-3, 1-4, 1-12
 data security 1-8
 database schema 4-3 to 4-37, 7-12 to 7-14, 8-4
 See also Tables.
 formats 4-34, 7-6
 foundation 1-4
 inventory control 1-10, 4-7, 7-5
 metadata 1-3, 1-4, 4-3
 output files. *See* Large media objects; VidPVideo table.
 physical attributes 1-3, 1-9, 7-5
 program 1-10, 4-7, 4-10, 7-5
 See also VidMDGeneral table.
 servers. *See* Storage servers.
 standards 4-32, 5-112
 See also VidStandard table.
 Web streamers 1-8

Video Foundation Class 6-4
 Video Foundation DataBlade module
 definition of 1-3
 examples 7-3 to 7-16
 sample uses of 1-20 to 1-22
 Video Foundation Interfaces. *See* Virtual Storage Interface (VSI).
 Video standards
 drop-frame NTSC 2-7, 2-8
 film 2-9
 NTSC 2-9
 PAL/SECAM 2-9
 VidFormat table 4-34, 7-6
 VidFrameNumberFrom function 2-18, 5-114
 VidImgDesc table 4-21, 4-26
 VidMDGeneral table 4-7, 4-10, 7-5, 7-6, 7-15
 VidMedia table 4-7, 4-11, 7-6, 7-7, 7-15, 7-16
 VidPAudio table 4-8, 4-17
 VidPOther table 4-8, 4-19
 VidPVideo table 4-8, 4-14, 7-7, 7-15
 VidStandard table 4-32, 7-6
 VidStrata table 4-21, 4-31, 7-8, 7-14, 7-16
 VidTextDesc table 4-21, 4-28, 7-9, 7-11, 7-14, 7-16
 VidVCodec table 4-37, 7-7
 VidVidDesc table 4-21, 4-30
 Virtual storage DataBlade modules
 developers of 1-17
 documenting 8-17
 registering with foundation 7-4, 8-14
 registering with Informix 8-4
 writing 8-3 to 8-18
 Virtual Storage Interface (VSI)
 API 9-3 to 9-15
 device-independent
 functions 3-4, 8-6
 device-specific
 functions 8-7 to 8-16
 meaning of 1-7
 in storage component 3-3 to 3-14
 supplied
 implementations 3-10 to 3-14

in virtual storage DataBlade modules 8-5 to 8-16
 writing an
 implementation 8-3 to 8-18
 VSI. *See* Virtual Storage Interface (VSI).
 VSI_CLIENT 1-8, 3-12
 example MedLoc 3-12
 VSI_FILE 1-8, 3-10
 example MedLoc 3-10, 7-4
 VSI_OTHER 1-8, 3-13
 example MedLoc 3-14, 7-15

W

Web video streamers 1-8
 Write function 8-7
 Writing a virtual storage DataBlade module 8-3 to 8-18
 Writing to a large media object 9-15