GRASS VALLEY GROUP

SOFTWARE DEVELOPMENT KIT

User Manual

software version **5.0**

071-8017-XX
PRELIMINARY - JULY 2001

PROFILE & PROFILE XP FAMILY

VIDEO FILE SERVERS AND MEDIA PLATFORMS

Revision Status

Rev date	Description
January 1995	Original issue. Manual part number 070-9187-00.
March 1995	Updated to support version 1.1. Manual part number changed to 070-9187-01.
November 1995	Updated to support version 1.3. Manual part number changed to 070-9187-02.
May 1996	Updated to support version 1.4. Reference material removed. Manual part number changed to 070-9187-03.
May 1999	Updated to support version 3.0. Manual part number changed to 070-9187-04.
April 2000	Updated to support version 4.0. Manual part number changed to 071-8017-00.
July 12 2001	Updated to support version 5.0. Preliminary version

Third-party License Agreements

Independent JPEG software license agreement

The authors make NO WARRANTY or representation, either express or implied, with respect to this software, its quality, accuracy, merchantability, or fitness for a particular purpose. This software is provided “AS IS”, and you, its user, assume the entire risk as to its quality and accuracy.

This software is copyright © 1991, 1992, 1993, 1994, Thomas G. Lane. All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this software (or portions thereof) for any purpose, without fee, subject to these conditions: (1) If any part of the source code for this software is distributed, then this README file must be included, with this copyright and no-warranty notice unaltered; and any additions, deletions, or changes to the original files must be clearly indicated in accompanying documentation. (2) If only executable code is distributed, then the accompanying documentation must state that “this software is based in part on the work of the Independent JPEG Group”. (3) Permission for use of this software is granted only if the user accepts full responsibility for any undesirable consequences; the authors accept NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code, not just to the unmodified library. If you use our work, you ought to acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name in advertising or publicity relating to this software or products derived from it. This software may be referred to only as “the Independent JPEG Group's software”.

We specifically permit and encourage the use of this software as the basis of commercial products, provided that all warranty or liability claims are assumed by the product vendor.

Intel GNU general public license agreement

The following listed PDR 100 tools are based on tools from the Intel GNU/960 Tools, some of which were developed and/or distributed by an organization called the Free Software Foundation (FSF): `gdb960.exe` and `objcopy.exe`.

These tools are covered by the GNU General Public License and have no warranty of any kind. The text of this license is built into gdb960.exe, and can be viewed by typing “info copying’ at the `gdb960` prompt. Source code for the above listed tools is available under the terms of this license by contacting Grass Valley Group.

Contents

List of tables

List of figures

List of examples

Overview

About the Profile Software Development Kit

The Profile Software Development Kit (SDK) includes two manuals:

- The *Profile SDK User Manual* which includes:
  - a conceptual overview of the Profile system;
  - a general discussion of the Profile programming libraries; and
  - a series of sample programs that demonstrate Profile functionality.
- The *Profile SDK Reference Manual* which includes:
  - application programming interface (API) and serial command summary tables;
  - a functional and alphabetical listing of all available API commands; and
  - a functional and numerical listing of all serial commands.

About this manual

The *Profile SDK User Manual* describes programming applications for the Profile video server:

Chapter 1, *Introduction* is a general overview of Profile application development. This chapter offers a brief introduction to basic Profile concepts and the core API libraries.

Chapter 2, *Programming the Profile Video Server* provides a more complete conceptual overview of the API libraries and what it takes to make them work.

Chapter 3, *Recording and Playing Movies* explores the basic work of recording and playing media.

Chapter 4, *Using the Profile Media File System* shows you how to write programs that inventory media on the Profile file system.

Chapter 5, *Using Events* furnishes a sample program that demonstrates the Profile event model.

Chapter 6, *Transferring Media with Fibre Channel* provides you with the code samples you need to write applications that implement Fibre Channel transfers, including streaming operations.

Chapter 7, *Programming the Profile Library System* helps you write programs for the Profile library system, a device that caches media on digital cartridge tapes.

Chapter 8, *Programming with MPEG* provides a sample program to help you record and play MPEG video.

Chapter 11, *Profile RS-422 Serial Control* covers programs that perform serial communication by using wrapper functions that implement the RS-422-based Profile protocol.

Getting started

The *Profile SDK User Manual* assumes that you understand basic Profile operations and that you are already familiar with the following resources:

- The *Profile Installation Manual*, containing essential information for the installation of your Profile server and the proper cabling for the installed cards.
- The *Profile Family User Manual*, containing essential information on properly configuring a Profile server for your environment.
- The *VdrPanel* application, which utilizes much of the Profile API functionality. Playing and recording test clips from VdrPanel is a useful exercise for the Profile developer.

If you are new to Profile programming

If this is your first introduction to Profile programming, we recommend you acquaint yourself with the libraries, commands, and examples before writing your own code.

- Read Chapter 1 of this manual for a general overview of Profile application development, an introduction to the core API libraries, and other basic Profile concepts.
- Study the sample programs that follow in all the chapters (complete code samples are available in the `c:\profile_sdk\src` directory). Focus on the API or serial code as appropriate to your development environment. Compile and run the sample applications.
- Once you are comfortable with the basic sample programs, you will be ready to create your own. Use the sample code as component building blocks. (You may want to cut-and-paste this code to create or customize your own work.)

If you are an experienced Profile programmer

We've made many changes to the SDK since the last release. We've added parameters to many existing functions and created some new functions which implement new features and provide convenient shortcuts.

- Read the release notes for an overview of the changes.
- You may want to work with your Profile administrator and/or other Profile developers to coordinate the upgrade to the current software.
- If you have existing applications written for an earlier version of the API, you may want to convert these as well.

Chapter **1**

Introduction

The Profile Software Development Kit provides an Application Programming Interface (API) for libraries of Profile functions that control the Profile video server. The software models supported by the API use a set of handles or identifiers to reference objects to which the application has requested access, or has reserved. (For a detailed explanation of the eight libraries, see ***Chapter 2,*** *Programming the Profile Video Server*.)

There are two programming models involved. The first programming model supports direct function calls that are issued by programs developed under Microsoft Visual C++. This model will also function across Ethernet, from a remote Win32 computer to the Profile. A second programming model supports a byte-stream serial protocol used over RS-422 communication lines.

All functions in a particular library share a three-letter identifying prefix. For example, all functions in the TekCfg library have the Cfg prefix. This is intended to help clarify the differences between functions of similar names and to alphabetize function listings in the companion volume to this user manual, the *Profile SDK Reference Manual*.

Louth and Odetics RS-422 protocols are supported, although there is not a one-to-one correspondence between these protocols and the Profile API. Louth and Odetics protocols do not allow access to the full functionality of the Profile system.

Basic concepts

The software model presented by the integrated libraries of the Profile separate functions according to the area of responsibility. Each library is associated with a major object of the system. At the top is the TekVdr library which provides transport control for recording and playing movies. This library provides access to the port clock and physical resources. Below this library is the TekPdr library which maintains the inventory of movies, their Common Movie Format descriptions, the logical requirements of the movies for resources, and how they relate to the media files.

A movie's description might be changed either while being associated with the physical resources of the system and being prepared for playing, or while in the inventory of movies. This means that there are two distinct, yet similar sets of functions in the Profile API: one is for manipulating the movie associated with the physical resources, the other one is associated with the movie in the inventory. A function with a title prefixed by *Vdr* is used to manipulate the movie associated with the physical resource. A function with a title prefixed by *Pdr* is used to manipulate the movie in the inventory. For example, **PdrSetMediaOut** sets the mark-out point for a media file which will be stored permanently with the movie. **VdrSetMediaMarkOut**, on the other hand, overrides any stored out-points, setting a mark-out position which applies to the current session only. It is essential for Profile developers to correctly distinguish which functionality they want.

A second aid to distinguishing the objects being manipulated is the naming convention for various API datatypes: objects associated with the physical resources are represented by *handles*, while objects associated with the movie inventory are represented by *tokens*. For example, a MovieHandle datatype represents a movie associated with physical resources, while a MovieToken represents a movie associated with the inventory.

A Profile system overview

Please refer to the Profile XP System Guide for a complete description of the Profile XP Media Platform. The following sections describe the PDR100, PDR200, PDR300, PDR400 and Profile PRO Series Professional Disk Recorders and Video File Servers.

The PDR series of video servers are multi-channel digital disk recorders. Both the PDR100 and PDR200 are capable of supporting up to four play/record channels of video; the video is compressed using the JPEG algorithm for storage on disk. The PDR200 and all later (higher numbered) models support Fiber Channel networking between systems so that content material can be moved among networked Profiles without decompression and recompression of the material. The PDR300 uses MPEG2 compression of the video for storage; both 4:2:0 and 4:2:2 video compression is supported. The PDR300 can support two record channels and eight playback channels. The PDR400 supports the DV compression algorithm for video. It can support up to sixchannels of DVCPRO25 or three channels of DVCPRO50 compression and decompression; all channels can both record and playback.

The Profile system has an EISA motherboard with an internal digital video routing system. There are sixteen EISA slots and one ISA slot used for interface cards and routing audio data. The server also uses a PCI bus for routing data between the master and slave enhanced disk recorder (EDR) boards, Fibre Channel boards, and boards for compressing and decompressing the video.

A video router chip set is integrated on the mother board. It routes video signals between the video compression boards, video mix effects cards, and video I/O cards. The video router is a 32x32 crosspoint matrix capable of full bandwidth 4:2:2 CCIR-601 8-bit digital video. The video router allows real-time transfer of video throughout the system without impacting overall system performance. The video router also makes possible simultaneous record and playback on separate channels.

Figure 1, The PDR200/300/400 block diagram on page 18 shows a block diagram of the hardware layout of a PDR200 or above.

Video disk storage

In the video disk subsystem, video data is compressed and written to up to eight 4-gigabyte (PDR100 only), 9-gigabyte, or 18-gigabyte disks, and then read from these disks and decompressed. This video data is read from and written to the video router in 8-bit, parallel component digital video format. The video disk subsystem has disk recorder boards (PDR100) and enhanced disk recorder boards (PDR200 and above), with an Intel i960 real-time processor and a SCSI-2 interface to the disks.

The video disk subsystem uses master and slave disk recorder or enhanced disk recorder (EDR) boards with two SCSI-2 channels on each board. The master disk recorder board comes standard with a two-channel JPEG codec. Bidirectional codec channels allow channels to be configured for recording or playback. Adding a slave disk recorder board makes a Profile unit a four-channel JPEG system. The master board has an Intel i960 real-time processor which controls compression and the data flows on SCSI-2 channels and JPEG codecs. Master and slave EDR boards also control MPEG encoder and decoder boards and DVCPRO codec boards, which are connected to the master and slave via a PCI interconnect board.

Video compression

The processor on the master enhanced disk recorder board is used to control the flow of data and load the coefficients of the compression encoder and decoder hardware. The amount of video compression varies according to the setting of the compression coefficients. Higher compression ratios store more video, but the result is lower quality video. On the other hand, lower compression ratios result in higher quality video and less storage capacity.

The compression coefficients are expressed in several ways; one of the more common ways of expressing the compression is in the resulting bitrate of the video stream. In addition, with MPEG compression the structure of the Group of Pix (GOP) will greatly effect the compression; long GOP structures with many P and B frames will be more compressed than an I-frame only compression.

Audio is not compressed on the Profile system.

Since the video compression ratio can be varied to change the video quality given available storage time, the amount of storage depends on your choice of compression ratio. A quick rule of thumb is that five minutes of JPEG video—plus four channels of audio and two channels of timecode—is roughly equal to one gigabyte of disk storage at 75,000 bytes per field in 525-line video. For example, a PDX208 Disk Expansion unit expands storage up to twelve hours and a PRS200 RAID Storage System can bring it up to approximately 96 hours. For video stored in the MPEG format at an average 24Mbps, you can just about double these capacities.

In addition to video compression, the disk recorder boards also integrate the digital audio data coming from the EISA bus, with typically four channels of audio per channel of video (up to 32). These recorder boards communicate with the SCSI-2 interface using a Direct Memory Access (DMA) interface. The PDR200 also supports the audio signal processing board (ASPB). This board is capable of delivering 16 channels of analog, embedded digital, or AES/EBU digital audio. The PDR200 can be equipped with two of these boards, for a total of 32 channels of audio.

Video and audio boards

Video and audio interface boards receive incoming and send outgoing video and audio data. These boards are responsible for converting the video and audio to internal formats used by the video server.

The PDR200, PDR300 and PDR400come with the audio signal processing board (ASPB). This audio architecture accepts and simultaneously processes sixteen audio inputs and outputs. Internally, all audio is processed with a selectable storage resolution of 16 or 20 bits. Inputs may be individually clocked in groups of four, and any clock group may be referenced to the system reference (house black) or any one of four video inputs. Output clocking is synchronous to system reference. Sample rate conversion is available for all inputs (30 to 50kHz), providing uniform storage at 48kHz.

You can configure the PDR200, PDR300, or PDR400 to operate with analog, AES/EBU digital, or embedded (SMPTE 272M Level A) audio, depending on which options are installed in your system. All three audio formats are supported without external conversion equipment. Analog audio is only available with an optional PAC208 or PAC216 Analog/Digital Interface chassis. You can expand the number of XLR or BNC connectors for AES/EBU audio with an optional XLR216 or BNC 216 digital interface chassis. You can choose an audio format for each video channel. For example, you could enable analog audio on one channel, embedded audio on another, and AES/EBU on the rest.

There are several video boards that allow a Profile server to be used with various standard video formats. Composite analog, serial digital component, or component analog video are all possible. All boards accept 525-line (NTSC) or 625-line (PAL) video standards.

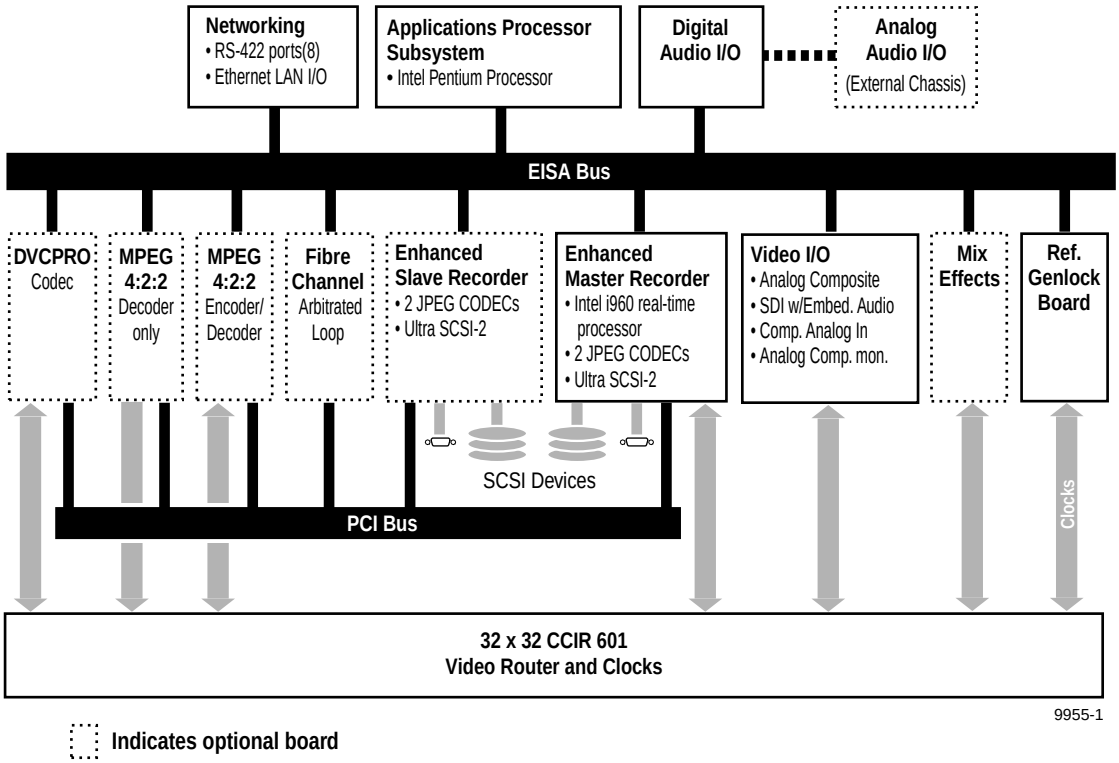
The latest analog composite input and output board offers two input and output channels per board. The two output channels for this board are similar to the output channels of the original analog composite board. An analog composite monitor board allows you to display text and burn-in timecode on an output monitor.

The component analog input allows dithering, auto-timing, and vertical blanking. As with other inputs, you can automate VITC detection. You can adjust input gain and also select an input format such as Betacam.

A serial digital component board provides two channels of both input and output, plus embedded audio when used with an ASPB. You can also enable dithering, auto-timing, and automate VITC detection. The board also has error detection and handling.

The standard reference genlock board allows you to time your Profile server to other devices in a broadcast facility. You can lock a Profile unit to a PAL or NTSC reference signal (house black). The genlock board also lets you have LTC inputs and outputs, with four inputs and four outputs possible for each channel.

Figure 1. The PDR200/300/400 block diagram



Chapter **2**

Programming the Profile Video Server

The Profile Software Development Kit (SDK) provides an application programming interface (API) for libraries of functions that control the Profile video server. Software developers can use this API to control the Profile server from third-party hardware devices. The Profile API consists of eight libraries:

- *The TekCfg library* provides an interface for configuring the Profile system. The Profile Configuration Manager, a standard application that comes with system software, implements many TekCfg functions.
- *The TekPdr library* furnishes calls that inventory and manage movies in Common Movie Format (CMF), a file format standard for storing video, audio, and timecode on disk.
- *The TekPls library* supplies function calls for controlling a library of digital tape cartridges that store video, audio, and timecode.
- *The TekRem library* makes it possible for a remote Profile or Windows NT system to control a Profile server over an Ethernet LAN.
- *The TekVdr library* provides an interface for playing and recording video and audio clips, and manipulating resources and their interconnections.
- *The TekVfs library* supports low-level access to individual media files in the media file system, as opposed to movies which consist of sets of media files.
- *The TekVme library* controls the optional video mix effects board which enables you to create various video transitions on a Profile server.
- *The TekXfr library* supports media streaming over Fibre Channel connections.

Eight RS-422 serial ports come standard on a Profile disk recorder. A Profile disk recorder can issue serial commands or receive them from an external device via RS-422 communication lines. The Profile serial protocol associates each API call with a specific number that can be sent over an RS-422 line. The ProLink application monitors Profile protocol calls over an RS-422 link, allowing you to use compatible hardware devices to issue commands to a Profile unit.

NOTE: Louth and Odetics RS-422 protocols are also supported, although there is not a one-to-one correspondence between these protocols and the Profile API. Louth and Odetics protocols do not allow full access to the functionality of the Profile system.

The TekCfg library

The TekCfg library is responsible for providing information about the Profile server's configuration. The major work of this configuration library takes place as the real-time embedded system receives its software load and starts running. At this point, the software discovers what hardware is available; it communicates the hardware information back to the configuration libraries in the host computer. The library formats the information and places it in the Windows NT registry with other hardware start-up information.

At the same time, the libraries check the registry for the preferred settings for the drivers in the real-time embedded system and communicate those settings to the drivers. Any variable parameters that have not been set previously, will be taken from default tables in the libraries. All the settings are placed in the registry.

After the real-time system is started, the configuration libraries interface to the information in the registry for the application. The application does not need to directly access the Windows NT registry and extract the information, but it can obtain the information from the functions in the configuration library.

By using the information available in the registry, an application can create and display lists for use by the user in selecting the components of the embedded system that are to be used by the application. For example, the application can determine the preferred file system onto which the media files are to be recorded, or the media inputs to be recorded. The application is able to use the same names for the components as all other applications on the system since the names are maintained by the configuration libraries. This aids in increasing the consistency in the user interface.

The TekPdr library

The TekPdr library provides an API for enumerating and managing an inventory of stored movies described in a Common Movie Format (CMF), and for editing the descriptions of the movies. A movie consists of multiple, synchronized streams of media, video, audio, and timecode. Within the Profile, the streams of media are held as files in the media file system. The CMF description of the movie specifies the relationship of segments of the stored media files to each other. Using the TekPdr library, the application can determine what movies are in the inventory, create and delete stored movies, and change the specification of the movie so that the relationship of media file segments is changed.

The Common Movie Format provides a structure to the media files that comprise the movies so that a single media file can participate in many movies. Additionally, a single movie track can reference multiple media files and contain segments of black media. By using the TekPdr library, applications can share movies. A movie can be recorded by one application, edited by a second application, and played by yet another application.

Using stored movies

There are two approaches to using stored movies.

In the first approach, you rely primarily on TekVdr calls: Attach empty media files to the port timeline resources with **VdrAttachMovie** and record into the movie with **VdrCueRecord** and **VdrShuttle**. The **VdrAttachMovie** function automatically creates the stored movie representation without further effort on your part. Under this approach, you will still make TekPdr library calls when you need to enumerate the movie database (for example, to select a movie for playback or record.) The enumeration of the stored movies is done in the TekPdr subsystem; the enumeration of movies on the port timeline is done in the TekVdr subsystem. (For enumeration of stored movies, see *Complex movie names* on page 23. Also, refer to the functions in the *SDK Reference Manual* associated with the current dataset and current group, and the functions associated with **FindFirst**, **FindNext** and **CloseFind** for the dataset, group and movie).

With the second approach, you manipulate the stored definition of the movie, directly accessing the lower level elements of the stored movie. There is functionality in the TekPdr subsystem to access the tracks and the individual media that comprise a track. Tracks exist only because there is media on the track, and the track never has to be explicitly created or deleted. Media can be added to existing tracks, or added to a new track causing it to come into existence. Media referenced from one movie can be referenced from a second movie by copying the media token to the second movie. Likewise, a movie can reference the same media multiple times by copying the media token. The actual segment of the media participating in any given reference is controlled by setting in and out points for the media in the media token. Indeed, the structure that represents a movie can be copied to provide a backup version of the movie before changes are made.

The header file *tekpdr.h* contains the function prototypes for the capabilities implemented by the library. The new data types of the library are specified in the header file *pdrtypes.h*. The header file *pdrattribs.h* contains movie attribute definitions. In addition, the header file *pdrerror.h* can be used to decode the extended error information that is returned by the system function **GetLastError**.

Common Movie Format

Common Movie Format (CMF) describes movies by specifying the relationship of stored media files to each other. Most users don't need to know more about CMF beyond this, its key benefit: All Profile applications--those developed by Grass Valley Group and by third-party vendors--share the same media files.

A movie is a combination of several multimedia streams of information (video, audio, timecode) synchronized to form a whole unit. In television one expects video, two or three audio inputs, and a timecode or two to be implicitly synchronized because all three were (historically) recorded together on tape, not on separate channels or tracks. In a digital world where these components can be filed as separate entities on discrete systems, however, synchronization can be a problem. That's what makes CMF so valuable. A CMF movie describes what information is in these various multimedia streams and how the streams are synchronized.

Using library commands with CMF movies

The functionality of the eight Profile API libraries provide notification to applications when the CMF movie inventory changes.

A CMF movie created with **PdrCreateMovie** stores header information about all media files associated with the movie as a whole and is identified with a MovieToken. Once recorded, a media file is referenced as a *media segment*, which is a ***portion*** of a media file (although that portion can comprise up to 100 percent). In order to define a portion of the media file to use, a descriptor is needed. In CMF, the media segment descriptor is the MediaToken, created with **PdrCreateMediaToken**. The MediaToken is a media segment reference. It points to a media file and contains two values (in/out) that specify what part of the file to use.

The movie header then references a list of tracks. Each track is referenced relative to the movie with a TrackToken. It has some information about the track and references a list of media segment references or MediaTokens. For example, suppose you have MovieToken 3. Using that MovieToken, you can discover a movie's name (**PdrGetMovieName**), its group (**PdrGetMovieGroup**), its dataset (**PdrGetMovieDataset**), its length (**PdrGetMovieLength**), when it was created (**PdrGetMovieCreateTime**), when it was last changed (**PdrGetMovieLastChangeTime**), and so on.

From the TrackToken, you can discover the length of the track (**PdrGetTrackLength**) and how many media segments are on the track (**PdrGetNumMediaOnTrack**). You can also request the next TrackToken (**PdrGetNextTrack**) or previous TrackToken (**PdrGetPreviousTrack**). In addition, you can query for information using the MediaToken. The MediaToken will help you determine which file is used for this media segment (**PdrGetMediaPath**), what the in/out points are (**PdrGetMediaIn**, **PdrGetMediaOut**, **PdrGetMediaMarks**), and so on.

You can set movies to read-only or locked mode, and you can open them in either exclusive or shared mode (use **PdrGetMediaAttributes** or **PdrGetMovieAttributes** to test media or move attributes). This format also makes it easier to create a new clip and to capture a feed into separate clips for later use in a complex movie.

A read-only movie's media files are protected against being rerecorded, but a read-only movie can still be edited. A locked movie is protected against any change, including a name change. However, because movies can share media, a movie can acquire the read-only attribute if some of its media files are read-only. This effectively is a warning at the media level that the movie cannot be recorded over.

When a movie is opened, it can be set to exclusive or shared mode. In exclusive mode, only one person can have the movie open, so it is permitted to expand the movie definition by adding media segment references. In fact, the only reason for opening a movie in exclusive mode is because an edit operation is going to add media segment references to the movie. A movie opened in shared mode may be opened by several people at once.

Complex movie names

The TekPdr library uses the *complex movie names* rather than separate group and movie names. A complex movie name consists of three parts: the dataset, group, and movie name. Either one (or both) of the first two parts can be implicit. (See **PdrSetCurrentDataset** and **PdrSetCurrentGroup**.)

The complexMovieName is formed as a string with punctuation separating the three components:

`<dataset:> /<group>/<movieName>`

...where the values between the angle-brackets `<>` are the values supplied by the calling application.

The component substrings must be comprised of characters from the following sets: `A-Z`, `a-z`, `0-9`, the `<space>` character, and the following special characters: `! # $ % & " ( ) + , - . ; = @ [ ] ^ ’ { } ~ " * < > \` and `|`. The slash, colon, and question mark characters (`/`, `:`, and `?`) are *not* allowed because they are used for punctuation of the complex movie name, or in streaming file transfer requests. The slash character is used to indicate the break between the components. The colon character should only be used in the dataset component as the terminating character. The question mark should only be used to specify options to a UML in Fibre Channel streaming transfers. The maximum length of each portion of the complex movie name is specified in the *pdrtypes.h* header file. (See `PDR_MAX_DSET_NAME_LEN`, `PDR_MAX_GROUP_NAME_LEN`, and `PDR_MAX_MOVIE_NAME_LEN`).

The characters `" * < > ? \ | ^` and `<space>` have special meaning at the operating system level. If these characters are included in the complex movie name, they are translated into a special double-character value. This means that every one of these characters used in the complex movie name decreases the maximum length of that portion of the name by one character.

The operating system also reserves some names that cannot be used for movie or group names. These names are: `CON`, `PRN`, `AUX`, `CLOCK$`, `NUL`, `A:-Z:`, `COM1-COM4`, and `LPT1-LPT3`. Using any of these names will cause the creation of a movie to fail.

While names allow the use of both uppercase and lowercase characters for testing whether the name is a duplicate or not, the comparison is not case-sensitive. (For example, the name *Adam* is a duplicate of the name *adam*.)

The dataset portion of the complex name identifies one of the file systems. For that reason, the dataset portion of the name must be the same as one of the file system names returned from **CfgGetFileSystemName**. The default dataset name is the name of file system 0, which is always available. The dataset name will always end with the colon as a terminator. To emphasize that the name is a dataset name, it may be terminated with the slash that starts the group name. If the dataset portion of the name is not explicitly supplied, the value of the current dataset is used for this component.

The group name portion of the complex name is chosen by the application/user to be meaningful and to help group the movies into categories. The default group name is *default*. The group name is identified in the complexMovieName because it is enclosed with slashes;

it is separated from the dataset portion with a slash and from the movieName portion with a slash. If the group portion of the name is not explicitly supplied, the value of the current group is used for this component.

The short movie name is the final portion of the complex name. It cannot be implicit. By default, the complex name is expected to contain a short movie name. To emphasize the movieName portion, the short name of the movie can be prefixed with a slash.

Table 1 shows the punctuation of a complexMovieName, and the resulting components after the name is scanned and values are supplied for the missing components. In the following examples, assume that the default dataset name is `INT:` and the default group name is *default.* Furthermore, assume that the name `EXT:` is a valid dataset name.

Table 1. Punctuation for a complexMovieName

complexMovieName	Dataset	Group	Short Movie Name
aaa	INT:	default	aaa
/mygrp/bbb	INT:	mygrp	bbb
EXT:/grp2/ccc	EXT:	grp2	ccc
EXT:ddd	EXT:	default	ddd
EXT:/eee	EXT:	default	eee
//fff	INT:	default	fff

The last example shows that the name can be formed from component parts even when the dataset and group are empty. The extra punctuation does not invalidate the syntax of the name. The syntax used for `aaa`, `bbb`, `ccc`, and `fff` are the preferred syntax.

Movie attributes

The attributes of a movie include access restrictions and information about sharing. Movie attributes are presented to the application programmer as a mask of bits, with each bit position representing a specific attribute. The bit positions can be tested by doing a binary of the attribute and seeing whether the result is nonzero.

The program attributes which can be set control the ability of programs to modify the movie. A movie that is ReadWrite (meaning ReadOnly is not set) allows the greatest degree of modification as the media files can be rerecorded. When a movie is made ReadOnly, the media files cannot be rerecorded or written. However, the movie can be edited so that the in/out points of a media file can be adjusted, and media segments can be added or deleted from the tracks of the movie. Finally, when a movie is locked, it cannot be modified; the in/out points cannot be changed.

The attribute of ReadOnly (RO) has a passive effect on other movies that share the recorded media of a movie that has been set to RO. Those other movies using the media are not allowed to rerecord the media. For this reason, those movies have their access mode set to RO to identify at the highest level that the media file(s) included in the movie are RO. The PdrControlRO bit of the attributes can detect whether the movie is RO because it was directly set to RO, or because some media it contains was set to RO. The PdrControlRO is set when a movie is explicitly set to RO.

If a movie is to be extensively edited and new media segments are to be added to its definition, the movie should be opened exclusively. This allows the application to use the movie without impacting other applications that might try to use the same movie.

A movie with a single media segment on each of its tracks is a simple movie—this is recognized as a *clip*. Most recording is made into clips. Clips are used as the foundation pieces for complex movies that have multiple media segments on a track. Because the clip can sometimes be treated differently, a movie that has the simple characteristic will have an attribute reflecting that. Attribute meanings are described in *Table 2* below.

Only PdrReadOnly, PdrLocked, and PdrOpenExclusive attributes are directly controlled by the programmer. The PdrControlRO attribute is set because the PdrReadOnly attribute was explicitly set on the movie, and the PdrOpen and PdrOpenMultiple attributes reflect the dynamic changes of access to the movie.

Table 2. Movie attribute descriptions

Attribute	Description
PdrAudio16Bit	This attribute indicates the predominate sample size for audio, 16-bit in this case.
PdrAudio24Bit	This attribute indicates the predominate sample size for audio, 20-bit in a 24-bit container for this case.
PdrCodecConstruction	This attribute indicates that the movie is under construction and being recorded.
PdrControlRO	This attribute means that the movie was set ReadOnly, and all of its media files have been set ReadOnly. A ReadOnly movie cannot be directly set to ReadWrite unless it has this attribute. Without this attribute, the ReadOnly attribute is a passive indicator that a shared media file has been set ReadOnly.
PdrCopyConstruction	This attribute indicates that the movie is a target of a copy over Fibre Channel.
PdrError	This attribute is returned when there is an error in trying to get the attributes.
PdrLocked	This attribute indicates that the movie has been Locked so that no changes can be made to the movie.
PdrOpen	This attribute indicates that at least one “open” exists on the movie.
PdrOpenExclusive	This attribute indicates that the movie has been opened exclusively. This will keep anyone else from opening the movie, but does allow the space for the movie data (media tokens and tracks) to grow without concern.
PdrOpenMultiple	This attribute indicates that more than one “open” exists on the movie.
PdrReadOnly	This attribute means that some of the media files of the movie have been set to ReadOnly and cannot be modified; the files cannot be written, and media cannot be recorded into them.
PdrRestoreConstruction	This attribute indicates that the movie is the target of a restore operation from a tape cartridge library.
PdrSampleRate50	This attribute indicates the movie’s predominant sample rate: 50Hz for PAL.
PdrSampleRate60	This attribute indicates the movie’s predominant sample rate: 60Hz for NTSC.
PdrSimpleClip	This attribute indicates that the movie is composed of exactly one media segment on each track.
PdrTcDropFrame	This attribute indicates that the moving is using drop-frame timecode.
PdrTcNonDropFrame	This attribute indicates that the movie is not using drop-frame timecode.
PdrUnderConstruction	This attribute indicates the movie meets any of the three construction criteria: recording, Fibre Channel copy, or library restore.
PdrVideoFormatJPEG	This attribute indicates that you are using the JPEG video format.
PdrVideoFormatMPEG	This attribute indicates that you are using the MPEG video format.

User data

Each object in a movie can contain associated user data. The user data can be associated with the movie, a particular track of the movie, or a particular media segment of the movie, but it has no significance to the library. The user data functions are responsible for storing data from the application program, retrieving data for the application program, moving the data with the movie object if it is copied, and deleting the data if the associated object is deleted.

The data is identified with a four-part key consisting of the MovieToken, TrackToken, MediaToken and Tag. The tag enables each application to have its own data associated with the movie object. The tag consists of two parts: a software vendor value and an item value. The high order bits of the tag are assigned to a software vendor so that each vendor can define its own use of the low order bits of the tag field. See the header file *pdrtags.h* for more details on assigning tag values (see the **PdrSetUserData** and **PdrGetUserData** functions).

Because the data that is stored and returned is not used by the library, the application is responsible for casting the data to the correct type. User data is handled as an array of unsigned bytes by the library. The array size is specified in the *Set* function. If a previously set user data item is to be deleted, the item is set with an empty value (for example, the length of the data is 0.)

Change notification

The TekPdr library is responsible for maintaining the inventory of movies in order that an application can be notified when the inventory has changed, or when a movie in the inventory has changed. The library contains functions that allow the application to receive change notifications asynchronously.

The key to the mechanism is that the application gets the handle for a notification event from the library. The application can then create a thread that uses the Win32 functions to WaitForSingleObject or WaitForMultipleObject. The library will satisfy the wait condition whenever a change of the desired type is made in the library. Once notified of a change, the application can use the PdrGetMovieChanges to identify the actual changes that are being reported.

An application that does not want to wait for the change event and has its own timing mechanism can use the PdrGetMovieChanges function in a polling manner to determine the changes to the stored movie inventory.

Saving a movie off the timeline

Some applications edit a movie directly on the dynamic timeline, and save the movie in the Common Movie Format (CMF) once editing is complete. These applications can add resources to the timeline, add media to existing resources, and set in and out points, thus creating an entire edited movie.

Once the movie is in its final form on the timeline, it can be saved in the inventory of movies with the **PdrSaveMovie** function. **PdrSaveMovie** can be invoked with one of three different PdrCopyTypes--PdrExactMedia, PdrRenderedMedia, and PdrSharedMedia--described in *Table 3*.

Table 3. PdrCopyType descriptions

PdrCopyType	Description
PdrExactMedia	This method uses the most bandwidth (and disk space). All media files are copied in their entirety, so that the stored movie owns one copy of each media file, while the original copy remains on the timeline available for additional editing and is referenced from the sources that were combined in the edit. This operation can take a substantial amount of time depending on the amount of media being copied and any other plays or records in progress.
PdrRenderedMedia	This method copies every field *currently* on the timeline into a *single* media file, honoring any in/out points or black on the timeline. Like the PdrExactMedia mode, it can be resource- and time-intensive since media is duplicated. Note, though, that the amount of media duplicated may be much less than with PdrExactMedia depending on the position of in/out points relative to the extent of the media files on the timeline. For example, given two half-hour media files on the timeline with in/out points denoting a 10-second highlight, PdrRenderedMedia will be *much* less resource-intensive than PdrExactMedia.
PdrSharedMedia	This is the normal, and simplest, method of saving a movie. The movie is saved in the inventory; the media segments are shared with other movies. This method is most efficient since no additional space or bandwidth is required from the embedded media file system.

Registry entries

The TekPdr library uses a Windows NT registry section to communicate a few parameters, namely `HKEY_LOCAL_MACHINE`. These parameters are stored under the key `\Software\Tektronix\Profile\PdrMovie`. The three most important keys are as follows:

1. **Max Movies** is the limit of all movies in the system. Storage for the maximum number of movies is allocated as the system starts.
2. **Max Media Definitions** is the limit of total number of tracks and media segments in any single movie. This number can be expanded for a particular movie if the movie is opened exclusively while the additional media segments and tracks are being added to the movie. The default value is large enough to allow the creation of a clip with nine tracks without requiring any special operation. When any specific movie is opened exclusively, the maximum number of media definitions can be increased without taking any special action: the library will increase the space allowed for that movie as necessary.
3. **Max Media References** is the limit of total number of references to media files by all movies in the system. Each media segment of type PdrMediaFile in each movie causes a media reference to be allocated.

Max Movies and **Max Media References** values can be increased without any impact on the existing inventory of movies. (These values may only be increased as decreased values may cause data corruption.) The system should be shutdown and restarted for the new values to take effect.

***ATTENTION: Direct editing of the NT registry is potentially risky. Never edit the registry if you are unsure what the consequences may be. The TekPdr library provides a separate interface to the registry settings above via the functions* PdrGetRegistry *and* PdrSetRegistry.**

The TekPls library

The Profile Library System (PLS) uses the client/server model of computing. There is one instance of the library server for each library. This server has a catalog that describes the contents of every tape cartridge loaded in a library. The catalog is a cache for tape cartridge directories. It can also retain residual knowledge of cartridges that have been removed from the library and stored elsewhere.

The library server works with files as a basic unit of information. A file can be a simple stream of bytes or a multiplexed stream of video, audio, and timecode. The library system copies files from a Profile system to tape cartridges and back, but does not delete files from a Profile unit--file management is handled by the TekVfs library.

The library server keeps a catalog of all files in the attached library. The purpose of this database is to allow a fast search for a given piece of material and to support requests for lists of the available material. When cartridges are removed from a library, the operator or application can have all references to material on the cartridge removed from the catalog. This is useful when cartridges are not going to be used in the near future or are being sent to other facilities. The catalog entries for removed cartridges can be retained. Having entries in a catalog makes locating the material faster. The catalog knows the cartridge is not in a library, and it has a note about where the cartridge is stored.

Tape cartridges are identified with unique barcode labels. Barcode labels are used so machine and human readable cartridge identification is available.

Some vendors' cartridges can be subdivided into partitions. A partition can be treated as if it were a separate tape for material replacement purposes. The first partition on a cartridge (at the load point) stores a master cartridge directory. Several types of tape cartridges may exist in a library. One type is clip, media and data file archive cartridges. Another is tape transport cleaning cartridges. Each cartridge must have unique barcode label. The library system reads and writes basic units of data called files. These can be one of several types: data files, clip files, and so forth.

The TekRem library

The TekRem library is responsible for connections between a remote Win32 system and a Profile. It is also responsible for communicating the calls made to the other API libraries residing on the Profile server.

The TekRem library can access the Profile system, either locally or remotely. The application makes a call to the TekRem library to get a ConnectHandle that will indicate which system to use. In the case of a local system, the ConnectHandle is the defined value of LOCAL_CONNECTION in the file *remtypes.h*.

The TekRem library makes remote Ethernet access possible between Profile systems or between a Profile system and a personal computer. For example, with a remote connection, you can control video operations over LAN from a PC in your office to a remote Profile running in another part of the building.

The first step in establishing a remote connection handle which identifies a Profile unit for the application that wants to communicate with it. If an application is running directly on the Profile unit, you can eliminate the remote connection by using the LOCAL_CONNECTION value for the *connHandle* parameter of the **RemOpenConnection** command.

The **RemOpenConnection** function opens a remote connection from the local host to the target remote system by specifying a host name or Internet Protocol (IP) address. You can also establish a local connection where the handle is named LOCAL_CONNECTION. The call returns a handle for the connection. **RemCloseConnection** close a remote host connection.

The TekVdr library

TekVdr is responsible for the Profile server's transport control (record, stop, and play actions), resource management, and connections. Resources are implicitly tied to a port and represent streams of media. A port functions in only one mode at a time and has a single clock to control all of its resources. Profile resources include physical resources, video resources, audio resources, and timecode resources.

Physical resources

Physical resources are the inputs and outputs of the video server and the JPEG, MPEG, DVCPRO 25, and DVCPRO 50 encoder/decoders that transfer the media streams to the media files on the disks. Some resources can be shared among ports and applications while others must be temporarily owned by the port and the application using them.

In general, input resources may be shared, that is, several ports can use the same video input without conflict. Instead of allocating the input resource, the application gets a connection handle that can be used in the same manner as an allocated handle. The codecs and output resources need allocation so that the port and the application have ownership and no other application can cause a conflict in their use.

JPEG, MPEG, DVCPRO 25, and DVCPRO 50 codec resources are the interface between the media streams and the media files. Drivers read and record the media files. In order to communicate which files, what segment of a file, and in what order to use the files, the application *attaches* the media files to the resources, setting in and out points for the file.

JPEG video resources

The JPEG video codec has three dimensions of control:

- field size goal;
- luminance quantization level; and
- chrominance quantization level.

These parameters interact. After the codec compresses a field, it compares the field size with the target field size. If the field size is smaller than the target, the quantization level is decreased. If the field size is larger, the quantization level is increased. A complex scene following a fade to black would cause a problem if the quantization level for the black fields is allowed to decrease too far. In that case, the complex scene would cause a jump in the field size that might swamp the system bandwidth. For this reason, there is an absolute minimum quantization level. The greater the range of quantization levels allowed, the closer the codec can keep the field size to the goal.

Video goal size

The video goal size is the target field size the JPEG codec tries to achieve when compressing the video. The actual field sizes after compression will vary on both sides of the target. The default value for the goal size is 75,000 bytes per field. The major limitation in increasing the bytes per field is the total bandwidth of the system. At 75,000 bytes per field, a single codec is using about 36Mbps (75,000 x 60 fields-per-second x 8 bits-per-byte). The maximum bandwidth for all four JPEG video codecs is 192Mbps. This can be partitioned among the four video codecs.

The field size varies as a function of the complexity of the field and the quantization level. The field size target is changed with the function **VdrSetVideoGoalSize**. The actual field size of the video going through the codec is obtained with the status function **VdrGetCurrentFieldSize**.

Luminance quantization level

The luminance quantization level is more important than the chrominance level. As the luminance quantization level decreases, the field size of complex scenes increases. The codec sets the actual level and an algorithm adjusts it for each field, trying to keep a constant field size. To avoid a large jump in the field size that might cause system problems, the quantization levels are bound by absolute minimums and maximums. You can narrow the range of quantization levels by having the codec use a reduced range within the absolute range of minimum and maximum levels.

You obtain the value of the absolute minimum and maximum quantization levels for luminance with the functions **VdrGetAbsMinLumQ** and **VdrGetAbsMaxLumQ**, and set new minimum and maximum values using the functions **VdrSetMinLumQ** and **VdrSetMaxLumQ**. To obtain the actual luminance quantization level of the video going through the codec, call the status function **VdrGetCurrentLumQFactor**.

Chrominance quantization level

The chrominance quantization level is less important than the luminance level. Generally, the change you will make to the chrominance quantization level is to reduce the maximum allowed quantization on complex scenes. As with the luminance quantization level, you can obtain the absolute range of the chrominance quantization level with functions **VdrGetAbsMinChrQ** and **VdrGetAbsMaxChrQ**, and set new minimum and maximum values that reduce the range using the functions **VdrSetMinChrQ** and **VdrSetMaxChrQ**; however, there is no status function to get the chrominance quantification level of video currently going through the codec.

MPEG video resources

The MPEG video codec has four dimensions of control:

- chrominance sampling;
- GOP structure;
- bitrate; and
- first and last line encoding.

Chrominance sampling

Use **VdrSetMpegChromaFormat** to set the chrominance sampling to 4:2:0 or 4:2:2, and **VdrGetMpegChromaFormat** (which returns an MpegChromaFmt enumerator such as MpegChroma420 or MpegChroma422) to query for the current format with.

GOP structure

Another early step is to set the GOP structure--the number of P- and B-frames you want to use (up to 16 total, minus one for the single I-frame that is “heart” of the GOP). Use **VdrSetMpegGopStructure** to set the structure and **VdrGetMpegGopStructure** to query the system for the current GOP. These functions use an enumerator of type MpegGopEnd that determines how an MPEG video stream begins, either open (GopOpenEnd) or closed (GopClosedEnd). In an open GOP, the initial B-frames have a preceding I-frame that is part of the previous GOP. A closed GOP, on the other hand, has initial B-frames that have a preceding I-frame that is part of the same GOP.

Bitrate

The bitrate, expressed in megabits per second, essentially sets the video quality for MPEG. The higher the bitrate, the higher the video quality. However, higher bitrates require more disk space to store the data, limiting the number of hours of material you can store on disk. Use **VdrSetBitRate** to set the bitrate in the range of 4 to 45Mbps and **VdrGetBitRate** to query the system for the current bitrate.

First and last line encoding

Finally, you can select which of the incoming lines of video are encoded as MPEG. **VdrSetEncodingRange** allows you to set the first and last encoded line of video. For 525-line systems, the starting and ending lines must be in the range 21 through 260, with an acceptable total of 512 or fewer lines per frame. For 625-line systems, the range is 7 through 310, with an acceptable total of 608 lines per frame. **VdrGetEncodingRange** returns the first and last lines as they are currently set.

Audio

This section discusses the differences between audio and video, including the architecture and how it effects the ResourceTypes and EventTypes. Also included in this section are the functions which reflect the difference in how things are heard versus how they are seen.

Analog audio architecture

In the case of video, each stream of video is placed onto the video bus. Therefore any video source to the bus can be connected to any destination on the bus. On the other hand, there is no audio bus and the audio codecs are not independent from the audio inputs, so the choices for resources are fewer than with video.

Each analog audio circuit board, standard with the PDR100, has four inputs, four audio codecs, and four outputs. Each audio input is permanently connected to its audio codec. Each audio output contains a mixer with four inputs. These mixer inputs are connected to the four audio codecs. The signal at the audio output depends on the percentage of each input signal used and whether the codec is in record/idle or play mode.

In record/idle mode, the codec looks at the audio input and that signal is placed on the input line of the audio output mixer. In play mode, the codec looks at the data coming from the disk and that data is placed on the input line of the audio output mixer. The only difference between record and idle is if the audio codec driver is saving the digitized audio on disk in record mode.

The Audio Signal Processing Board (ASPB) comes with the PDR200, PDR300, and PDR400. A similar board is used in the Profile XP Media Platform. The ASPB architecture accepts and simultaneously processes up to 32 audio inputs and outputs (with two boards) at up to four simultaneous clock rates. Inputs may be individually clocked in groups of four, and any clock group may be referenced to the system reference (house black) or any one of four video inputs. Output clocking is synchronous to system reference.

You can configure the ASPB to operate with analog, AES/EBU digital, or embedded (SMPTE 272M Level A) audio, depending on which options are installed in your system. All three audio formats are supported without external conversion equipment. You can choose an audio format for each video channel. For example, you could enable analog audio with one video channel, embedded audio on another, and AES/EBU on the rest.

Audio resources

The TekVdr library contains the enumerators ResourceAudioCodec, ResourceAudioInput, and ResourceAudioOutput. The application can provide orthogonal images of the media types. This is accomplished by having both types of ResourceHandles for the audio codec resources. Use the shared connection handle as the audio input handle and use the allocated handle for the audio codec handle. Since both handles represent the same resource, the connection geometry is transparent and appears similar to the video connection geometry.

Also, with the audio resources, there may be some limitations on the audio connections, based on the physical limitations of the audio hardware. For example, audio inputs cannot be connected to outputs that are not on the same board. In the PDR100, you may select which output is participating in the connection, then offer only the four inputs/codecs that can participate with that output. This group of four is identified by being resource numbers 0, 1, 2, and 3 modulo 4. That is, the sequence of numbers will be consistent with four resources per circuit board in order. For the PDR200 and higher and the PVS1000, 16 or 32 inputs/codecs are allowed, and so resource numbers are 0 and 1 modulo 32.

Audio minimum play length

To recognize sound, the length of audio being played needs to be more than a single field. Otherwise, the repetitive replay of the sound breaks down into a 60 or 50 cycle drone. Using an audio window for looping the audio helps eliminate that problem. Instead of replaying just a single field as the video replays the field it is positioned over, the audio plays the media of the field it is over, plus several additional fields making the sound more easily understood. The number of fields to play when looping on a single position on the timeline is called the *audio window.* Call **VdrSetAudioWindow** to set the size of the window.

Timecode

In the Profile system, timecode is treated like any other media. Input timecode may be recorded in a file. A file of timecode may be output with the audio and video media streams. In addition, a timecode generator can be used instead of the timecode input or timecode recorder/codec to create timecode values. (Timecode, taken in its larger sense, includes user bits).

At the first level, timecode is like video in that it has all three components in the stream: input resources, codecs, and output resources. Also, the resources are connected together using events of the EventConnectResources type. A media file must be attached to the timecode codec (recorder) in order to either record or playback timecode values. The function **VdrGetCurrentTimeCode** gets the current timecode and the function **VdrGetCurrentUserBits** gets the current user bits. A timecode media file can be inspected on a field-by-field basis to determine the timecode that corresponds to a specific field position in the file.

The port

In order to work with physical resources, an application must first obtain a port object. The port is the center of the application's relationship to Profile resources. The port has two functions:

1. To link together a set of physical resources into a multimedia recorder.
2. To synchronize the resources while recording or playing.

A recorder operates on one or more streams of media (either video, audio, or timecode) and records those streams to media files, or plays back existing media files into those media streams. A recorder/port's resources are linked together and all active resources must be doing the same function: recording media files, playing existing media files, or idle. Different active resources attached to a single port cannot do different things. All inactive resources are considered to be idle.

The second function of the port is to keep all active physical resources of the multimedia recorder synchronized. This means the port tells the different resource drivers which field to play, and when, so that all of the tracks in the recorder work together. One model of the port is a timeline with parallel tracks containing the media files. The port clock can take any position on the timeline, and the media related to that position on the timeline are used by the resource drivers. On record, the timeline position corresponds to the fields in the media files that receive the current field in the media streams. On playback, the timeline position corresponds to the fields in the media files that are the source of the current field on each media stream (*Figure 2*).

An application that must record and play simultaneously needs at least two ports, one for recording and one for playback. Likewise, if the application must play multimedia at two different rates simultaneously, it needs multiple ports. A single port is a single clock that ties all of the port's active resources together.

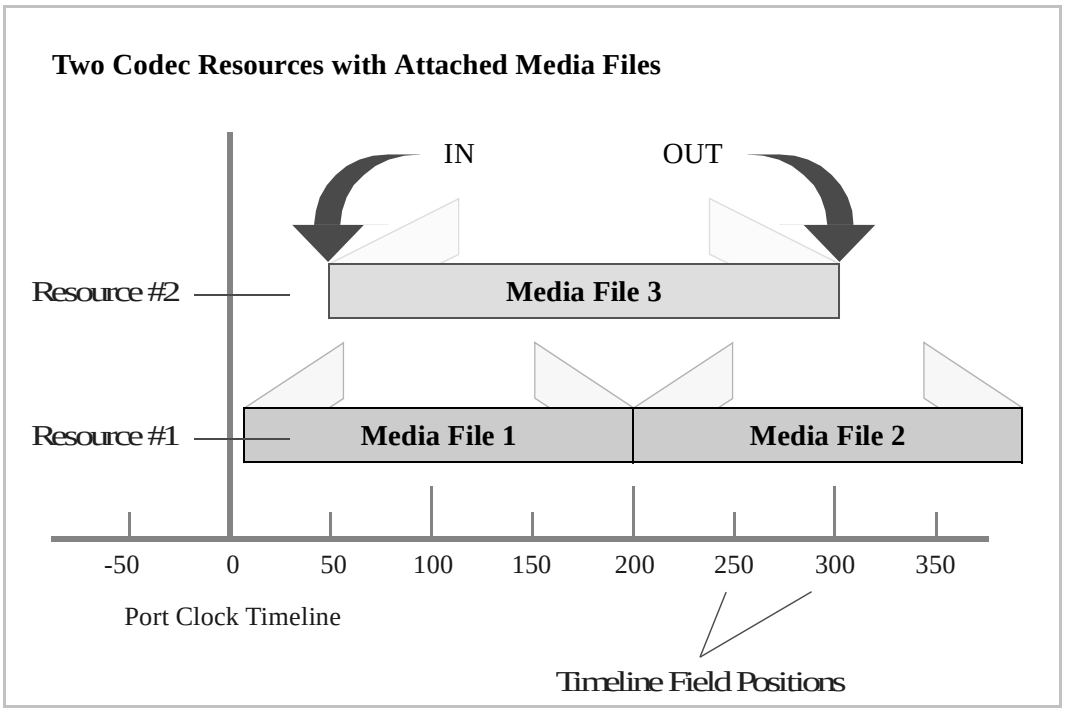


Figure 2. Timeline position in media files

Port clock modes

To this point, the port clock mode has been the default MediaPlayMode of PlayNormal. In this mode, the clock position is implicitly limited by the limits of the attached media. The maximum clock value is determined by the track whose last media out point has the greatest timeline position value. The minimum clock value is determined by the track whose first

media in point has the smallest timeline position value. As the clock moves from one extreme to the other, any track whose media does not exist at all possible clock positions plays black. Any recording where there is no media file is not saved. In general, when the shuttle command is given, the clock changes its position at the rate and direction given in the shuttle command. This means a negative rate makes the clock move backward. The clock does not always start at the same point: it starts with the last point at which it was located. The clock stops changing its position when the last field within the implicit limits of the port is played. The resource drivers continue to use that last field position until told otherwise.

Within the port, there is complete symmetry between record and play operations. The port can be set to jog while recording; the resource drivers continually use the field of the file that corresponds to the clock position. Although the resource drivers are given the same field position many times before the clock position advances, only the first field from the media stream is written into that position of the file. Other fields are discarded. Likewise, a record operation with a rate that is greater than unity will cause empty field positions in the file. This consistency between record and play modes also exists with the clock modes described in the following sections.

Other clock modes

There are three additional clock modes (MediaPlayModes): PlayLimited, PlayLoop, and PlayBounce. In addition, there is a second explicit set of clock limits that are used with these play modes. Set the explicit clock limits with the **VdrSetMinPosition** and **VdrSetMaxPosition** functions. Without setting these explicit limits, the additional clock modes will not work. Change the clock mode with the **VdrSetPlayMode**.

The PlayLimited mode is similar to the PlayNormal mode, except that it plays within the explicit limits on the clock position. The clock advances in the desired direction at the desired rate until the last position in the range is reached. Then the clock stops advancing and the last field position is used repeatedly. In play mode, the last field is shown as if the port was in jog mode.

In PlayLoop mode, once the clock reaches the last field within the explicit clock limits, it jumps to the last field at the other end of the media and continues. If the clock is advancing (forward), the last field of the media delimited by the explicit clock limits is used, then the clock jumps to the first field of the delimited range.

In PlayBounce mode, once the clock reaches the last field within the explicit limits of the clock, the sign of the rate changes, and the clock continues in the other direction.

Still modes

In addition to rate-based clock modes, there is a field-based mode that controls which fields are used while in jog mode. In jog mode, the clock advances only as requested by the application. Each time the clock advances, the application must call **VdrJog.** The function's parameter indicates the number of fields to change the clock and the sign is the direction. In jog mode, the clock remains at the same position for many field times and the resource drivers continually reuse the same field time in their files.

The application can control the fields used at this level with the StillMode setting. The StillMode can be either PlayByField or PlayByFrame.

NOTE: MPEG does not support a StillMode of PlayByField but does support PlayByFrame.

In PlayByField, the clock uses only the field where it is positioned. On playback, the video resource driver uses line doubling to provide a full image. In PlayByFrame mode, the clock alternates its position between the two fields that comprise a frame. On playback, this enhances the video vertical resolution.

Port clock limits

The timeline of a port ranges between field numbers of approximately $-2x10^9$ to $+2x10^9$. The port clock is limited to a subrange of these field numbers. In NormalClock mode, the clock limits are applied implicitly, and you do not need to be concerned with the limits. This is the default case. In NormalClock mode, the clock limits are set to the limits of the attached media files. The clock limits just enclose the earliest and latest times that have associated files on the timeline. The port clock will not take values outside these minimum and maximum limits.

Different codec timelines might start and end at different points after various files have been attached, deleted, and had in/out points changed. This means that the port clock can take positions in which there is no media file for a codec to access. In these cases, the codec driver plays black, or throws the recorded field away. (Black is defined as silence for the audio codecs, 00:00:00:00 for timecode.)

Synchronizing ports

One port timeline can be controlled synchronously with a second port timeline; every position change in the master port will cause the same relative position change in the slave port while the ports are in play or record mode.

The application should first establish the relative positions of the ports on their timelines. Then, the application must place each port into the proper cue mode for the action desired on the port; either CuePlay or CueRecord. The ports can be doing different functions.

The ports will now stay locked to their relative positions; when the master port position is changed, a similar change will be seen by the slave port. It is possible to slave multiple ports to a single master port. The slave mode is established using the **VdrSetPlayMode** function.

Events

With resources allocated to a timeline, the next issue is managing these resources. The dynamic subsystem allows for scheduling of several types of resource-related events via the Events API. Specifically, you can programmatically control video and timecode crosspoints, audio mixes, timecode values and timecode user bits. The following table summarizes the available events. (See the *SDK Reference Manual* for full details on function parameters.)

Table 4. Summary of available events

To manage...	...use this event type
Video or timecode crosspoints	EventConnectResources
Audio mix levels	EventMixAudio
Timecode values	EventSetGTcTime
Timecode user bits	EventSetGTcBits

These events are coordinated with the API functions **VdrDefaultEvent, VdrScheduleEvent** and **VdrStateEvent**. To understand how these functions control the timing of events, be aware of the various states a port may assume. Figure 3 presents a simplified state diagram of a port, including the API calls which can move a port between states.

Default events are not associated with a particular position on the timeline, but take effect any time the port is in an idle or record state. On the other hand, scheduled events take effect at a given video field while a port is in the play state. It is possible to schedule multiple events for a given resource. The API also provides functionality to reschedule and cancel previously scheduled events. Refer to the *SDK Reference Manual* for more details.

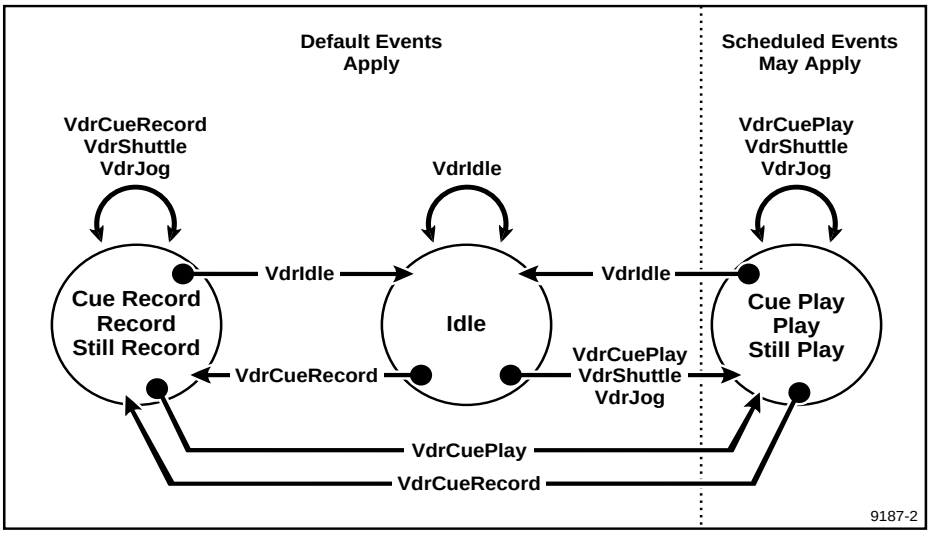


Figure 3. Profile state

State events

VdrStateEvent adds flexibility to the event mechanism. **VdrStateEvent** can replace the use of **VdrDefaultEvent** and **VdrScheduleEvent** in most cases. You can view default and schedule events as subsets of state events. State events can replace most usage of the scheduled events because they are much more efficient and provide greater flexibility.

There are three states of the Profile port relevant to this discussion:

- Record/Idle state is active whenever the port is in Record or Idle mode.
- ActivePlay state is active whenever the port is in PlayShuttle or PlayJog mode.
- ReadyToPlay state is active whenever the port is in CuePlay state, or the port would be in ActivePlay state except that the position cannot be advanced because it is stopped by a position limit.

The **VdrStateEvent** function describes an event that occurs when the Profile port switches into the specified state. The simplest explanation will use the existing Default- and ScheduledEvents to show what happens. With a DefaultEvent declared, the event will occur each time the Profile port switches from any Play state to either a Record state, or an Idle state.

Consider the EventConnectResources type of Event, which determines how the resources are connected. Normally, the application will want the input routed to the output when in Idle or Record modes. This is accomplished by declaring a DefaultEvent with that connection. Each time the application switches the port from Play to Idle, or Record, the input is connected to the output.

Likewise, a ScheduledEvent is normally used by the application to switch the connection such that the codec is connected to the output when the Profile port goes into Play mode. While the Scheduled Event can be used to control more specifically when the switch is made, most applications set the time of occurrence to be as early as possible, and do not attempt to control the switching at a specific vertical interval. When the time is set to infinity, the event becomes a StateEvent--it occurs on the change of state.

By having three states, a wider variety of applications are possible. The use of the Profile video server as a combination pass-through switcher and spot inserter underscores the need for the separation of the Play state into two substates. *Figure 4* shows a Profile unit and a video switcher under automation control.

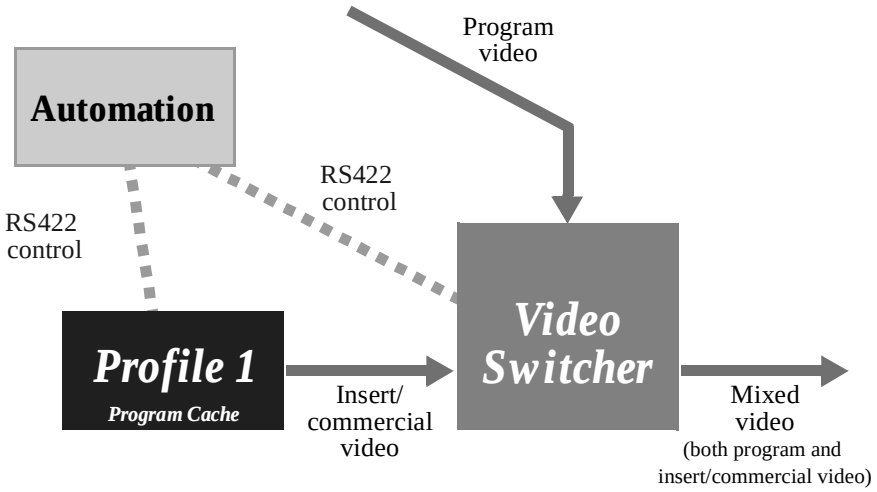


Figure 4. Profile unit and video switcher under automation

Normally, the Profile port would be placed into CuePlay as the spot is prepared for insertion. As the media files are buffered, the initial image of the spot is displayed on the output in a freeze frame mode. This is a confidence check that the material is cued, and the video server is ready.

In the case where a Profile system is acting as a pass-through switcher, it is not desirable to show the cued first frame of the spot. The output needs to continue to show the input until it is actually time to show the spot. Then, as the spot finishes, it is necessary to switch the output back to the input to continue to show program material that is passing through the video server. *Figure 5* shows the combination of two Profiles acting in series.

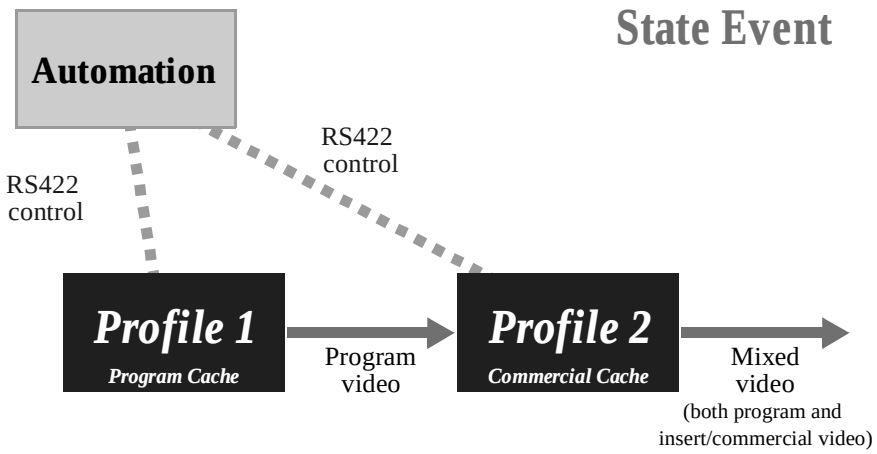


Figure 5. Pass-through Profiles in series

By having three states, we can specify that the output of Profile 2 is connected to the input during the Record/Idle state, and during the ReadyToPlay state, but the output is connected to the codec during ActivePlay.

Current applications, such as the example shown in *Figure 6, Attaching a new media file to a timeline* on page 43, that are showing the first frame of the spot in a freeze frame manner are using the three states such that the output is connected to the Input during the Record/Idle state, but during the ReadyToPlay state and the ActivePlay states the output is connected to the codec.

In the StateEvent, a subtle side effect of the current DefaultEvent and Scheduled Events is eliminated. Currently, if there is no ScheduledEvent, the DefaultEvent will continue to be active. In the StateEvent, the event for each state must be specified. If the same set of connections are to be used in all states, this can be specified in the StateEvent. It is explicitly specified, and not an implicit condition because another event was not specified.

This functionality is still available, but it is specified differently.

```
VdrDefaultEvent(hPort, NULL, EventConnectResources, hInput, hOutput);
```

...becomes:

```
VdrStateEvent(hPort, EventStateAll, EventConnectResources, hInput, hOutput);
```

To override the Play states, the previous ScheduledEvent:

```
VdrScheduleEvent(hPort, MOST_NEG_FIELD, EventConnectResources, hCodec, hOutput);
```

...becomes:

```
VdrStateEvent(hPort, EventStateAllPlay, EventConnectResources, hCodec, hOutput);
```

In the above, the following defines are in effect from *VdrTypes.h*:

```
#define EventStateIdleRecord (1 << 0)
#define EventStatePlayActive (1 << 1)
#define EventStatePlayReady  (1 << 2)
#define EventStateSwitcher  (5)
#define EventStateAllPlay   (6)
#define EventStateAll     (7)
```

The function prototype of the **VdrStateEvent** has the same form as **VdrDefaultEvent** with the one change that the reservedHandle field which was NULL in **VdrDefaultEvent** is now the StateMask field. The StateMask indicates in which states the event should be active. For compatibility, **VdrDefaultEvent** is equivalent to **VdrStateEvent** with the StateMask set to EventStateAll.

Example 9, stateevt.c on page 100 demonstrates how to use **VdrStateEvent** to provide more control over port connections during the ReadyToPlay state and how to use **VdrStateEvent** to replace **VdrDefaultEvent** and **ScheduleEvent**.

Audio events

The audio event is of the type EventMixAudio. This event has two more parameters than the corresponding EventConnectResources. These parameters are the target level (as a percentage) of the input signal in the output and the duration (as a number of fields) over which the change in level should take place. This is the maximum duration of the change (the target level might already be in use). Thus, events to play equal parts of audioCodec1 and audioCodec2 into audioOut1 starting at port clock position 300 and getting to the target level in 1/3 second would look like:

```
VdrScheduleEvent(port, 300, EventMixAudio, audioCodec1, audioOut1, (float)50., 20);
VdrScheduleEvent(port, 300, EventMixAudio, audioCodec2, audioOut1, (float)50., 20);
```

Timecode generator events

Instead of using a file of timecode during playback, or an external source of timecode during record, the application can use a *timecode generator.* A timecode generator is another resource type. Once allocated to the port, it is connected to the timecode codec or output in the same manner that any source would be connected to a destination. The generator is controlled both by functions for control setup and by events for data setup.

The control functions of the timecode generator provide for setting the timecode format and the generator mode. Timecode formats allowed are TcFormatDropFrame or TcFormatNonDropFrame. These are set with the function **VdrSetGenTcFormat**.

Generator modes are set with the function **VdrSetGenTcMode**. The possible modes are: TcModeFreeze, TcModeFreeRun, and TcModeFieldLocked. In Freeze mode, the identical value is generated every field time. The TcModeFieldLocked mode is correlated to the port clock, such that any non-linearity in the positions of the port clock will result in the same non-linearity in the generator output.

The generator's data values are initiated by events. The event types are EventSetGtcTime and EventSetGtcBits. By using events, you can force the time value and the user bit value to a known value at a specific clock position. This could be used to mark the cut point between different clips in a playback list.

The encoding of the function parameters for these events would be as follows:

```
#define JAM_TIME 500
VdrHandle port;
EventHandle setTime, setBits, clrBits;
ResourceHandle tcGenHandle;
UINT time = SET_TIMECODE (1, 30, 15, 00);
// 1 hour, 30 min, 15 sec, 00 frame.
UINT bitsOn = 0x00008000; // set bit on
UINT bitsOff = 0x00000000; //set bit off after 60 fields.
...
setTime = VdrScheduleEvent(port, JAM_TIME, EventSetGtcTime, tcGenHandle, time);
setBits = VdrScheduleEvent(port, JAM_TIME, EventSetGtcBits, tcGenHandle, bitsOn);
clrBits = VdrScheduleEvent(port, JAM_TIME+60, EventSetGtcBits, tcGenHandle, bitsOff);
```

Timecode time is encoded in the system as packed decimal values, while the user bits are a word of hexadecimal values.

Media files

The file system is shared by all applications, all ports, and all the media (audio, video, and timecode). The files have an addressable resolution of one video field (this is more than a single sample of audio.) The media file addresses (field numbers) are all positive, starting at 0. However, all fields of a media file do not need to be recorded.

A media file must be attached to a codec resource in order to be used with a media stream. The file must be delimited with in/out points. The in point is the field position in the file used first by the codec, while the out point is the first field position beyond the last field used by the codec. The in/out points are relative to the media file, and not related to the codec timeline where the file is attached.

The in/out points also protect the rest of the media file. The codec will only use segments of the media file that are bracketed with in/out points.

For a new media file, the default out point is always 0, and the file is empty. Until the in/out points for the attached new media file are set, the file is not used: it is merely a place-holder on the timeline with a length of 0. This means the out point of a new file must be set in order to record into the file. The file appears on the timeline with the length specified by the out point. However, the file system itself will only remember the highest field number that was ever recorded (and not deleted). The file system automatically gives the file a length that includes the highest numbered field actually recorded in the file.

The media file attached to a codec resource is said to have a *position* on the timeline. The position is the relationship between the field numbers on the timeline and those in the file. The position of the file is the timeline field number of the file in point.

Multiple files

A single resource may have multiple media files attached. In that case, the files are placed head to tail along the timeline. There are no gaps between files, and the last field position of the first file is immediately before the first field position of the second file. The out position of the first file is coincident with the in position of the second file. The resource uses the two files without any break between them.

There are several cases where the timeline must be adjusted by shifting. One case is when a file is added to the timeline. In this case, the additional file will be attached *before* another file, and the position of some of the files on the timeline are shifted to make room for the added file

fields. Other cases are if a file is deleted from the timeline, or if the in or out point of a file is changed. In these examples, the number of fields in the abutted material changes, resulting in changing of some files' positions.

In order to simplify things, you can specify how the shifting is accomplished. The shift can be either before or after the point on the timeline where the change is being made. If the shift is after the point, then any file that occupies field positions on the timeline falling at or after the point of action is effected. In the case of the shift being before, then any file occupying field positions on the timeline that fall at or before the point of action is affected.

The effect of shifting is that the abutted files on the timeline may no longer begin at the timeline position of 0. Any **ShiftBefore** actions will move the beginning of the first file away from the position 0, and the new position can be on either side of the position 0 depending upon the action; adding fields to the timeline will move the beginning of the first file to the left, while deleting fields from the timeline will move the beginning of the first file to the right (see *Figure 6* and *Figure 7*).

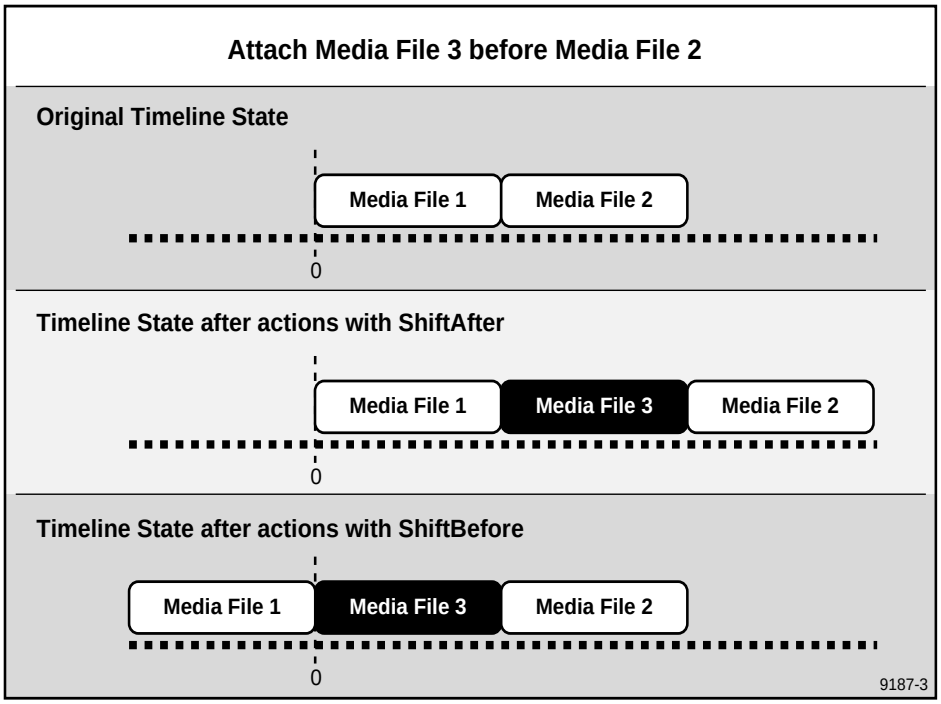


Figure 6. Attaching a new media file to a timeline

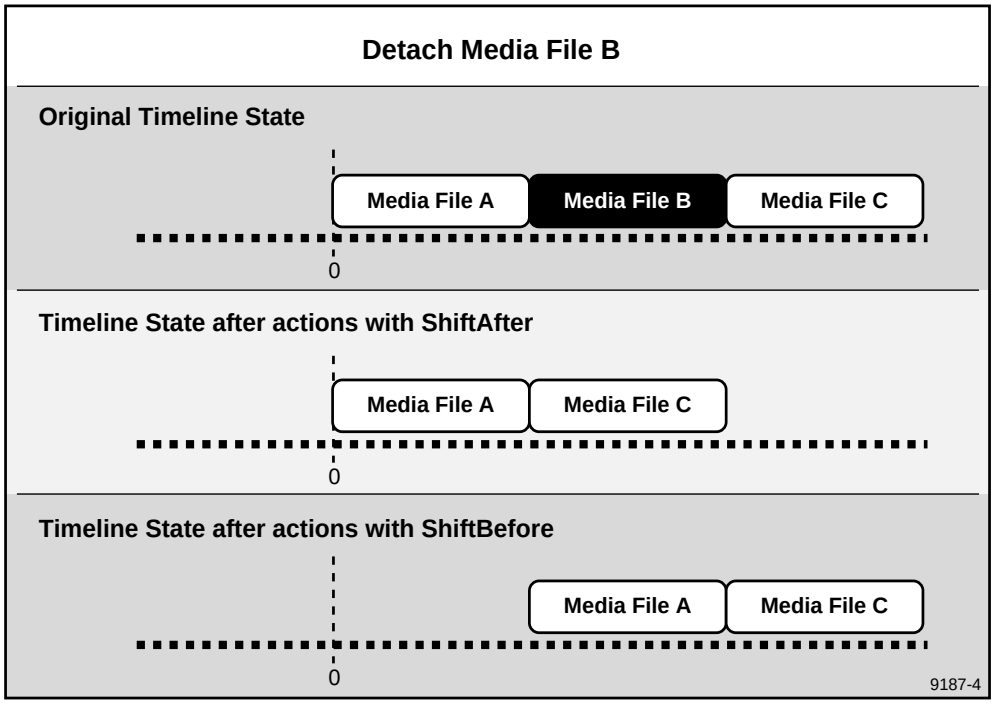


Figure 7. Deleting media from timeline with effect on other media

You might choose to do a **ShiftBefore** action if the point of action is before the current clock position on the timeline, and a **ShiftAfter** action if the point of action is after the current clock position. This method minimizes the impact of the shift. Consider an example in which the current clock position is 30 when a change is made at the action point of 15. If a **ShiftAfter** action is used, the relationship of the files to the timeline at the current clock position of 30 changes, and what should have been a smooth playback could have sudden cuts into other material. Since the application can only change the timeline of one resource at a time, the results are even worse since some of the resources will continue with the expected material and file field numbers.

The TekVfs library

The TekVfs library provides an interface to the media file systems used by the real-time embedded system. Most of the functions of the host file system are available to the application by using the TekVfs library.

While the functions in the TekVfs library may be slightly different in the parameter list from the host file system functions, the logic is similar. The application programmer opens a file by using the **CreateFile** function with the flags set to OPEN_EXISTING. The same is true with the media file system, only the function is **VfsCreateFile**. (There are minor differences in the actual parameterization of functions based on the need to remote the functions for the media file system, and the lack of NT security in the media file system).

The files of a directory are enumerated using the Find functions: **FindFirstFile**, **FindNextFile**, **FindClose**. Similarly, you enumerate the files of the media file system using VfsFind functions: **VfsFindFirstFile**, **VfsFindNextFile**, and **VfsFindClose**. The find data is even returned in a WIN32_FIND_DATA structure.

The big difference between the media file system and the host file system is the amount of storage needed for media files. The media file system has a limited address space, so media files are stored in a special manner: the blocks of media are outside the address space of the file system, and only a pointer to the block is stored in the file.

The media file contains a small header area that identifies a media file, and describes its parameters. An empty media file has a length of 128 bytes. If the file system does not recognize its header on a file being imported into the system, the file will be considered a non-media file. This will cause the file to be stored within the address space of the file system, and this, in turn, may use all the available space for the file system.

The media file is addressable at the field level. Individual fields of the file can be imported and exported to memory buffers on the host system.

The TekVme library

The Profile video mix effects board enables you to create various styles of video transitions. This section includes definitions of the transitions that will be supported and the methods of allocating the mixer segments in the initial implementation of the video mix effects API.

The video mix effects board mixers can be allocated as two independent single stage mixers. Each single stage mixer will have two video inputs, one video key input, one wipe generator, and one video output. While this software implementation only supports single stage mixers, the calls retain a stage specifier so that future implementations can support multiple stage mixers without significant changes to the interface.

The following types of transitions are supported by any mixer stage:

- Dissolve between foreground and background.
- Keyed dissolve between foreground and background.
- Wipe between foreground and background, with or without border:
  - Possible wipe styles are horizontal, vertical, horizontal split, vertical split, corner or diagonal, box, diamond, and circle. These can also have multiplier values, be positioned with *x* and *y* coordinates, have rotation, and have modulation edge.
  - Borders have selectable matte color, width and softness, and can also be modulated by sawtooth, square or sine wave of selectable frequency with changing or fixed phase.
- Keyed wipe between foreground and background without borders.
  - Keys have selectable clip, gain, and invert state.
- Push on/off between foreground and background, with or without border.
- Push over between foreground and background, with or without border or shadow.
- Dissolve between foreground or any of the above transition outputs and matte (fade to black). This can happen at any time relative to any transition in progress.

Each transition is created in a mix context which is referenced through a VmeHandle. The mix context is obtained, the desired transitions created for each stage within the context, then the context is applied to the hardware. When no longer needed, the context can be freed.

The TekXfr library

Fibre Channel enables you to copy media between Profile systems faster than in real time. To transfer media across Fibre Channel connections, Profile systems must have Ethernet local area network (LAN) and Fibre Channel boards installed and the systems need to be attached to both networks in order to communicate. The Ethernet LAN carries commands between Profile units while the Fibre Channel connections carry the actual video, audio, and timecode data. Both networks use TCP/IP (Transmission Control Protocol/Internet Protocol).

Streaming over Fibre Channel makes it possible to transfer parallel tracks of media, such as concurrent audio and video tracks, in small packets, and then reassemble them on the destination Profile unit. This allows the network to transfer a clip while it is still being recorded, or to playback a clip before the transfer is complete. For example, soon after it begins receiving a clip, a destination Profile unit can begin playing it. The streaming transfer continues while playback occurs at the destination, delivering new packets at a fast enough rate to allow playback to proceed uninterrupted.

Chapter 3

Recording and Playing Movies

This chapter explores the basic work of recording and playing media with a Profile video disk recorder/server, such as:

- Recording a simple movie.
- Playing a simple movie.
- Playing a movie that has in and out marks.
- Playing multiple (complex) movies.
- Each explanation is followed by a sample program written in C.
- Recording a movie

Recording and playing back a movie are two very basic operations of the Profile video server. In this section, we discuss the basic steps necessary to accomplish these tasks, then present the same steps in the form of small sample applications that record a movie clip to disk and play back the saved movie clip.

Example 1, record.c illustrates the record function. (This example uses JPEG compression; see *Example 16, mpegdemo.c* on page 183 in for a similar example. It may be useful to review the JPEG play sample before reviewing the MPEG example, since this example builds on the basic play operations.)

The program processes the command line to obtain the movie name and duration for recording. Then it calls the **SetupResources** routine in order to allocate and connect the appropriate resources. A connection to the local machine is established with **RemOpenConnection**, and a port is opened with **VdrOpenPortConnection**. The standard video format for the machine is determined with **CfgGetStandard**, and the port is set to that format using the **VdrSetVideoFormat** function. The number of JPEG codecs is determined with **CfgGetNumCodecs** and a free JPEG codec resource is identified and allocated with **VdrAllocateResource**. The codec allows video encoding and decoding to occur on the specified port.

Similarly, two audio codecs are allocated for the port, one for each stereo channel. Audio inputs are considered permanently attached to the audio codecs and, therefore, available with the audio codecs.

The video input and the video and audio output resources are also allocated for the port, in order to receive a video signal and have somewhere to send it for testing.

The video input resource is allocated using **VdrGetResourceConnectionHandle**, which provides a handle value for the resource to which a connection can be made but doesn't take control of the resource the way **VdrAllocateResource** does. The video input resource is described as "shared".

Lastly, audio output resources are allocated, completing the allocation of resources. Audio input resources do not need to be specified since the audio codes are assumed to be connected to the identically numbered audio input resources.

After this, default (**VdrDefaultEvent**) and scheduled events (**VdrScheduleEvent**) are used to make the connections that occur when the port is in various states. The connection made with the scheduled event is only active during playback. It connects the video codec to the video output resource. In all other states, the connections made with the default events are active, connecting the video input to the video output--which allows viewing the video input at all times that the port is not in play mode--and to the video codec. Since the audio connections are even more complex, the normal pattern of connections has been set to the default. (See *Example 9, stateevt.c* on page 100 in for more information about these events and how to gain more control over their behavior.)

Once the resources have been obtained, the main procedure performs the record operation by calling **StartRecord**. The program tries to open the movie with **PdrOpenMovie**, to see if it already exists. If the program detects that the movie does not exist, the function **VdrAttachMovieWithMarks** creates the movie and attaches it to the timeline with empty media files. With the movie attached, the **VdrCueRecord** function places the port in record mode.

At this point, the port is ready to record, and **VdrShuttle** starts recording.

After the specified duration is recorded, the current position (**VdrGetPosition)** stops changing, so that as soon as two samples of the position separated by 100ms show the same position, the port is returned to idle mode (**VdrIdle**) and the move is detached from the port timeline (**VdrDetachMovie)**. The entire movie is played back because mark-in and mark-out points are not specified.

Finally, the resources acquired during execution must be released. The main procedure calls cleanup to release all of the handles acquired during the setup phase (**VdrReleaseResource**) and close the port (**VdrClosePort**). This completes the process and stops the program.

If an error occurs at any step of the way, an appropriate error message is output to the display for troubleshooting.

Using a command-line program, the accompanying code sample illustrates theses steps by recording and playing back a clip on a local Profile system.

Example 1. record.c

```
//
// File: record.c
// This sample program records a JPEG clip of a specified time (measured
// in seconds) and then plays it back.
//
// Copyright (c) Grass Valley Group Inc. 1996-1998
// All rights reserved.
//
// Usage: record movie_name -s seconds
//
#include <stdio.h>
#include <windows.h>
#include <limits.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>

#define SHUTTLE_RATE 1.0
#define NUM_INPUT   0
#define NUM_OUTPUT  0

// For demo application, we will have several resources. Enumerate
// them for use as indices into an array for VdrAttachMovie calls.
// First three are Codecs.

enum CodecResEnum { VCOD, ACOD1, ACOD2, MAX_CODEC };
enum ResEnum { VIN = MAX_CODEC, VOUT, AOUT1, AOUT2, MAX_RSRC };

// Module static variables.
static ConnectHandle  sConn;
static VdrHandle    sPort;
static ResourceHandle  sResHdls[MAX_RSRC];
static char*      spMovieName;
static int       sSeconds;
static int       sRate;

//
// Print out usage line.
//
void Usage(const char* progName)
{
  printf("Usage: %s movie_name -s seconds\n", progName);

} // Usage

//
// Initialize the Profile. Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources(void)
{
  BOOL  rtn;
  BOOL  event1, event2;
  VideoFormat videoFormat;
  int    i, vlimit;
  EventHandle evtHand;

  printf("Starting setup...\n");
```

```
// Open the connection and the port.
rtn = RemOpenConnection(ConnectLocal, 0, 0, &sConn);
if (!rtn) {
  printf("Error opening connection.\n");
  return FALSE;
}
sPort = VdrOpenPortConnection(sConn);
if (!sPort) {
  printf("Error getting port \n");
  return FALSE;
}

// Is this NTSC or PAL?
switch (CfgGetStandard(sConn)) {
  case PCI_PAL_625_MODE:
    videoFormat = Format625_50Hz_2To1;
    sRate = 50;
    break;
  case PCI_NTSC_525_MODE:
    videoFormat = Format525_60Hz_2To1;
    sRate = 60;
    break;
  case PCI_INVALID_MODE:
  default:
    printf("Invalid or unknown video mode.\n");
    return FALSE;
}
VdrSetVideoFormat(sPort, videoFormat);

//
// Now, get the necessary resources for the demo.
//

// Find first available codec.
vlimit = CfgGetNumCodecs(sConn, JpegCodec
  // same as VideoCodec);
for (i=0; i<vlimit && !sResHdls[VCOD]; i++) {
  sResHdls[VCOD] = VdrAllocateResource(sPort, ResourceJpegCodec,
  // same as ResourceVideoCodec (unsigned int)i);
}

if (!sResHdls[VCOD]) {
  printf("Cannot allocate jpeg video codec.\n");
  return FALSE;
}

// Get two audio codecs.
sResHdls[ACOD1] = VdrAllocateResource(sPort, ResourceAudioCodec,
  NUM_INPUT);
sResHdls[ACOD2] = VdrAllocateResource(sPort, ResourceAudioCodec,
  NUM_INPUT+1);
if (!sResHdls[ACOD1] || !sResHdls[ACOD2]) {
  printf("Cannot allocate audio codec.\n");
  return FALSE;
}

// Get video in and out resources.
sResHdls[VOUT] = VdrAllocateResource(sPort, ResourceVideoOutput,
  NUM_OUTPUT);
if (!sResHdls[VOUT]) {
  printf("Cannot allocate video output.\n");
  return FALSE;
}
```

```
  sResHdls[VIN] = VdrGetResourceConnectionHandle(sPort,
    ResourceVideoInput, NUM_INPUT);
  if (!sResHdls[VIN]) {
    printf("Cannot get video input.\n");
    return FALSE;
  }

  // Get audio resources.
  sResHdls[AOUT1] = VdrAllocateResource(sPort, ResourceAudioOutput,
    NUM_OUTPUT);
  sResHdls[AOUT2] = VdrAllocateResource(sPort, ResourceAudioOutput,
    NUM_OUTPUT+1);
  if (!sResHdls[AOUT1] || !sResHdls[AOUT2]) {
    printf("Cannot allocate audio resources.\n");
    return FALSE;
  }

  // Set the default event.
  event1 = VdrDefaultEvent(sPort, NULL, EventConnectResources,
    sResHdls[VIN], sResHdls[VOUT]);
  event2 = VdrDefaultEvent(sPort, NULL, EventConnectResources,
    sResHdls[VIN], sResHdls[VCOD]);
  if (event1 == FALSE || event2 == FALSE) {
    printf("Cannot schedule default events. \n");
    return FALSE;
  }

  // Schedule the event.
  evtHand = VdrScheduleEvent(sPort, INT_MIN, EventConnectResources,
    sResHdls[VCOD], sResHdls[VOUT]);
  if (!evtHand) {
    printf("Cannot schedule event.\n");
    return FALSE;
  }

  return TRUE;

} // SetupResources

//
// Clean up by releasing resources and closing the control port.
//
void Cleanup(void)
{
  int i;

  printf("Starting cleanup...\n");

  for (i=0; i<MAX_RSRC; ++i) {
    if (sResHdls[i]) {
       VdrReleaseResource(sResHdls[i]);
       sResHdls[i] = 0;
    }
  }

  if (!VdrClosePort(sPort)) {
    printf("Cannot close port. \n");
    return;
  }
  sPort = 0;

} // Cleanup
```

```
//
// Play the movie clip.
//
void StartPlay(void)
{
  INT oldpos, newpos;
  MovieHandle movieHdl;

  // Attach the movie that we just recorded.
  if (!PdrMovieExists(sConn, spMovieName)) {
    printf("Movie %s does not exist \n", spMovieName);
    return;
  }

  movieHdl = VdrAttachMovie(spMovieName, MAX_CODEC, sResHdls, NULL,
    ShiftAfter, MarkLongest);
  if (!movieHdl) {
    printf("Error getting movie handle \n");
    return;
  }

  // Cue up playback of media attached with VdrAttachMovie.
  if (!VdrCuePlay(sPort, SHUTTLE_RATE)) {
    printf("Cannot cue play \n");
    return;
  }

  // Begin motion playback.
  if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
    printf("Cannot begin playback \n");
    return;
  }

  printf("Starting playback...\n");

  // Wait while movie plays.
  // When newpos and oldpos are the same, we're done playing out.
  newpos = 0;
  do {
    oldpos = newpos;
    Sleep(100);            // wait 1/10th second
    newpos = VdrGetPosition(sPort);
  } while (newpos > oldpos);

  // Cease play back.
  if (!VdrIdle(sPort)) {
    printf("Cannot move to idle.\n");
    return;
  }

  // Detach the movie handle from the channel.
  if (!VdrDetachMovie(movieHdl, ShiftAfter)) {
    printf("Cannot detach movie.\n");
    return;
  }
} // StartPlay

//
// Record a clip in JPEG format.
//
BOOL StartRecord(void)
{
```

```
  INT  oldpos, newpos;
  int    markout;
  MovieHandle movieHdl;

  // Check to see if the movie already exists.
  if (PdrMovieExists(sConn, spMovieName)) {
    printf("Movie name already exists.\n");
    return FALSE;
  }

  markout = sSeconds * sRate;
  movieHdl = VdrAttachMovieWithMarks(spMovieName, MAX_CODEC, sResHdls,
    NULL, ShiftAfter, MarkLongest, 0, markout);

  if (!movieHdl) {
    printf("Error getting movie handle \n");
    return FALSE;
  }

  if (!VdrCueRecord(sPort)) {
    printf("Cannot cue record \n");
    return FALSE;
  }

  if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
    printf("Cannot cue shuttle \n");
    return FALSE;
  }

  printf("Starting record...\n");

  // Wait until record is done.
  newpos = 0;
  do {
    oldpos = newpos;
    Sleep(100);            // wait 1/10th second
    newpos = VdrGetPosition(sPort);
  } while (newpos > oldpos);

  // Stop the recording.
  if (!VdrIdle(sPort)) {
    printf("Cannot idle port\n");
    return FALSE;
  }

  // Detach the movie from the timeline.
  if (!VdrDetachMovie(movieHdl, ShiftAfter)) {
    printf("Cannot detach movie.\n");
    return FALSE;
  }

  return TRUE;

} // StartRecord

//
// The main entry point.
//
void main(int argc, char *argv[])
{

  BOOL rtn, rtn1 = FALSE;
  int i;
```

```
  // Read in the new movie name.
  i = 1;
  if (argv[i]) {
    spMovieName = argv[i];
  }
  else {
    Usage(argv[0]);
    exit(1);
  }
  i++;

  // Process seconds argument.
  while (i < argc) {
    if (argv[i][0] == '-')
      switch (argv[i][1]) {
      case 's':
        i++;
        sSeconds = atoi(argv[i++]);
        break;
      default:
        Usage(argv[0]);
        exit(1);
      }
    else {
      Usage(argv[0]);
      exit(1);
    }
  }

  rtn = SetupResources();

  if (rtn)
    rtn1 = StartRecord();

  if (rtn1)
    StartPlay();

  Cleanup();

} // main
```

Playing a movie

Example 2, play.c shows you how to play a movie in the JPEG compression format, with optional mark-in and mark-out points. (This example uses JPEG compression; see *Example 16, mpegdemo.c* on page 183 in for a similar example.)

First, the appropriate resources must be allocated for later use. A connection to the Profile is established with **RemOpenConnection**, and a port is opened using a call to the function **VdrOpenPortConnection**. The standard video format for the machine is determined with **CfgGetStandard**, and the port is set to that format with **VdrSetVideoFormat**. Next, a JPEG codec resource is allocated and attached to the port with **VdrAllocateResource**. The codec allows video decoding to occur on the specified port.

Similarly, two audio codecs are allocated and attached to the port, again with **VdrAllocateResource,** one for each stereo channel. The video input and output resources are then allocated and attached to the port with similar calls.

Finally, default and scheduled events are setup, using **VdrDefaultEvent** and **VdrScheduleEvent** respectively. These calls describe the connections that occur when the port is in various states. (See *Example 9, stateevt.c* on page 100 in for more information about these events and how to gain more control over their behavior.)

Once the resources are obtained, you proceed to the playback. The movie is opened via **PdrOpenMovie** and is attached with the function **VdrAttachOpenMovie**. The optional mark-in and mark-out points are set with **VdrSetMovieMarkIn** and **VdrSetMovieMarkOut**. After that, the movie is cued for play (**VdrCuePlay**). It begins playing when the video server is instructed to shuttle using **VdrShuttle**, and after the specified duration has passed—determined by **VdrGetPosition**—idle mode is set with **VdrIdle** and the movie is detached using **VdrAttachMovie**. The movie is closed with **PdrCloseMovie**.

Finally, the resources acquired during execution must be released. With a call to **VdrReleaseResource**, the Profile unit is told to release all of the handles acquired during the setup phase, and the port is closed with **VdrClosePort**. This completes the process and the program is stopped.

If an error occurs at any step of the way, an appropriate error message is output to the display, so that troubleshooting may take place.

Example 2 illustrates these steps with a command-line program that plays back a clip.

Example 2. play.c

```
//
// File: play.c
// This sample program plays a specified JPEG clip.
//
// Copyright (c) Grass Valley Group Inc. This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// Usage: play movie_name [-i markin] [-o markout]
//
//

#include <stdio.h>
#include <windows.h>
#include <limits.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>

#define SHUTTLE_RATE 1.0
#define NUM_INPUT   0
#define NUM_OUTPUT  0

// For demo application, we will have several resources. Enumerate
// them for use as indeces into an array for VdrAttachMovie calls.
// First three are Codecs.
enum CodecResEnum { VCOD, ACOD1, ACOD2, MAX_CODEC };
enum ResEnum { VIN = MAX_CODEC, VOUT, AOUT1, AOUT2, MAX_RSRC };

// Module static variables.
static ConnectHandle  sConn;
static VdrHandle    sPort;
static ResourceHandle  sResHdls[MAX_RSRC];
static char*      spMovieName;
static int       sMarkIn;
static int       sMarkOut;

//
// Print out usage line.
//
void Usage(const char* progName)
{
  printf("Usage: %s movie_name [-i markin] [-o markout]\n", progName);

} // Usage

//
// Initialize the Profile. Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources(void)
{
  VideoFormat videoFormat;
  int    i, vlimit;
  EventHandle evtHand;

  printf("Starting setup...\n");

  // Open the connection and the port.
```

```
if (!RemOpenConnection(ConnectLocal, 0, 0, &sConn)) {
  printf("Error opening connection.\n");
  return FALSE;
}
sPort = VdrOpenPortConnection(sConn);
if (!sPort) {
  printf("Error getting port \n");
  return FALSE;
}

// Is this NTSC or PAL?
switch (CfgGetStandard(sConn)) {
  case PCI_PAL_625_MODE:
    videoFormat = Format625_50Hz_2To1;
    break;
  case PCI_NTSC_525_MODE:
    videoFormat = Format525_60Hz_2To1;
    break;
  case PCI_INVALID_MODE:
  default:
    printf("Invalid or unknown video mode.\n");
    return FALSE;
}
VdrSetVideoFormat(sPort, videoFormat);

//
// Now, get the necessary resources for the demo.
//

// Find first available codec.
vlimit = CfgGetNumCodecs(sConn,
             JpegCodec // same as VideoCodec
             );
for (i=0; i<vlimit && !sResHdls[VCOD]; i++) {
  sResHdls[VCOD] = VdrAllocateResource(sPort,
  ResourceJpegCodec, // same as ResourceVideoCodec
  (unsigned int)i);
}

if (!sResHdls[VCOD]) {
  printf("Cannot allocate jpeg video codec.\n");
  return FALSE;
}

// Get two audio codecs.
sResHdls[ACOD1] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT);
sResHdls[ACOD2] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT+1);
if (!sResHdls[ACOD1] || !sResHdls[ACOD2]) {
  printf("Cannot allocate audio codec.\n");
  return FALSE;
}

// Get video in and out resources.
sResHdls[VOUT] = VdrAllocateResource(sPort, ResourceVideoOutput, NUM_OUTPUT);
if (!sResHdls[VOUT]) {
  printf("Cannot allocate video output.\n");
  return FALSE;
}

sResHdls[VIN] = VdrGetResourceConnectionHandle(sPort,
ResourceVideoInput, NUM_INPUT);
if (!sResHdls[VIN]) {
  printf("Cannot get video input.\n");
```

```
    return FALSE;
  }

  // Get audio resources.
  sResHdls[AOUT1] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT);
  sResHdls[AOUT2] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT+1);
  if (!sResHdls[AOUT1] || !sResHdls[AOUT2]) {
    printf("Cannot allocate audio resources.\n");
    return FALSE;
  }

  // Set the default event.
  if (!VdrDefaultEvent(sPort, NULL, EventConnectResources,
 sResHdls[VIN], sResHdls[VOUT])) {
    printf("Cannot schedule default event. \n");
    return FALSE;
  }

  // Schedule the event.
  evtHand = VdrScheduleEvent(sPort, INT_MIN, EventConnectResources, sResHdls[VCOD],
    sResHdls[VOUT]);
  if (!evtHand) {
    printf("Cannot schedule event.\n");
    return FALSE;
  }

  return TRUE;

} // SetupResources


//
// Cleanup by releasing resources and closing the control port.
//
void Cleanup(void)
{
  int i;

  printf("Start cleanup...\n");

  for (i=0; i<MAX_RSRC; ++i) {
    if (sResHdls[i]) {
       VdrReleaseResource(sResHdls[i]);
       sResHdls[i] = 0;
    }
  }

  if (!VdrClosePort(sPort)) {
    printf("Cannot close port. \n");
    return;
  }
  sPort = 0;

} // Cleanup


//
// Play the movie clip.
//
void StartPlay(void)
{
  INT oldpos, newpos;
  MovieToken movieTok;
```

```
  MovieHandle movieHdl;

  // Open and attach the movie.
  movieTok = PdrOpenMovie(sConn, spMovieName, 0);
  if (!movieTok) {
    printf("Movie %s does not exist \n", spMovieName);
    return;
  }

  movieHdl = VdrAttachOpenMovie(movieTok, MAX_CODEC, sResHdls, NULL,
                 ShiftAfter, MarkLongest);
  if (!movieHdl) {
    printf("Error getting movie handle \n");
    return;
  }
  if (sMarkIn) {
    VdrSetMovieMarkIn(movieHdl, sMarkIn, ShiftAfter);
  }
  if (sMarkOut) {
    VdrSetMovieMarkOut(movieHdl, sMarkOut, ShiftAfter);
  }

  // Cue up playback of media attached with VdrAttachMovie.
  if (!VdrCuePlay(sPort, SHUTTLE_RATE)) {
    printf("Cannot cue play \n");
    return;
  }

  // Begin motion playback.
  if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
    printf("Cannot begin playback \n");
    return;
  }

  printf("Playing movie \"%s\"...\n", spMovieName);

  // Wait while movie plays.
  // When newpos and oldpos are the same, we're done playing out.
  newpos = 0;
  do {
    oldpos = newpos;
    Sleep(100);            // wait 1/10th second
    newpos = VdrGetPosition(sPort);
  } while (newpos > oldpos);

  // Cease play back.
  if (!VdrIdle(sPort)) {
    printf("Cannot move to idle.\n");
    return;
  }

  // Detach the movie handle from the channel.
  if (!VdrDetachMovie(movieHdl, ShiftAfter)) {
    printf("Cannot detach movie.\n");
    return;
  }

  if (!PdrCloseMovie(movieTok)) {
    printf("Cannot close movie.\n");
    return;
  }

} // StartPlay
```

```
//
// The main entry point.
//
void main(int argc, char *argv[])
{
 int i=1;
// Read in the new movie name.
  if (argv[i]) {
    spMovieName = argv[i];
  }
  else {
    Usage(argv[0]);
    exit(1);
  }
  i++;
// Process optional markin and markout points.
  while (i < argc) {
    if (argv[i][0] == '-')
      switch (argv[i][1]) {
      case 'i':
        ++i;
        sMarkIn = atoi(argv[i++]);
        break;
      case 'o':
        ++i;
        sMarkOut = atoi(argv[i++]);
        break;
      default:
        Usage(argv[0]);
        exit(1);
      }
    else {
      Usage(argv[0]);
      exit(1);
    }
  }
if (SetupResources()) {
    StartPlay();
  }
Cleanup();

} // main
```

Playing a movie with in and out marks

Example 3, play_fitted.c shows you how to play a movie in the JPEG compression format with optional mark-in and mark-out points for a given duration.

First, the appropriate video resources must be allocated for later use (this particular example does not acquire audio resources). A connection handle to the local machine is established with **RemOpenConnection**, and a port is opened using **VdrOpenPortConnection**. Then the standard video format for the machine is determined with **CfgGetStandard**, and the port is set to that format via **VdrSetVideoFormat**.

Next, a JPEG codec resource is allocated and attached to the port with **VdrAllocateResource**. The codec allows video decoding to occur on the specified port. The video input and output resources are then allocated and attached to the port, also using **VdrAllocateResource**. Finally, default (**VdrDefaultEvent**) and scheduled events (**VdrScheduleEvent**) are setup to describe the connections that occur when the port is in various states. (See *Example 9, stateevt.c* in for more information about events and how to gain more control over their behavior.)

Once the resources are obtained, you are ready to perform the playback. The movie is first opened with **PdrOpenMovie**. After the first track and media tokens are obtained with the functions **PdrGetNextTrack** and **PdrGetNextMediaToken**, the media's path is determined with **PdrGetMediaPath**.

Then the movie is attached and fitted to the specified duration with optional mark-in and mark-out points being set (see **VdrAttachFittedMedia** or **VdrAttachFittedMediaWithMarks**). After that, the movie is cued for play using **VdrCuePlay**. It begins playing when the Profile unit is instructed to shuttle with **VdrShuttle**, and after the specified duration has passed, idle mode is set (**VdrIdle**) and the movie is detached with **PdrDetachMedia**. The movie is then closed with **PdrCloseMovie**.

Finally, the resources acquired during execution must be released. The disk recorder/server is told to release all of the handles acquired during the setup phase using **VdrReleaseResource**, and the port is closed with **VdrClosePort**. This completes the process and the program is stopped.

If an error occurs at any step of the way, an appropriate error message is output to the display, so that troubleshooting may take place.

Example 3. play_fitted.c

```
//
// File: play_fitted.c
// This sample program fits a recorded clip into a given time frame.
// It deals with video media only.
//
// Copyright (c) Grass Valley Group Inc. This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// Usage: play_fitted movie_name duration [-i markin] [-o markout]
//
//

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <limits.h>
#include <string.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>

#define SHUTTLE_RATE  1.0
#define NUM_INPUT    0
#define NUM_OUTPUT   0

// Enumerate resource ID's for use as indeces into an array.
enum CodecResEnum { VCOD, VIN, VOUT, MAX_RSRC };

// Module static variables.
static ConnectHandle  sConn;
static VdrHandle    sPort;
static ResourceHandle  sResHdls[MAX_RSRC];
static char*      spMovieName;
static int       sMarkIn;
static int       sMarkOut;
static int       sRate;
static int       sSeconds;

//
// Print out usage line.
//
void Usage(const char* progName)
{
printf("Usage: %s movie_name duration [-i markin] [-o markout]\n", progName);

} // Usage

//
// Initialize the Profile. Report any anomalies.
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources()
{
  VideoFormat videoFormat;
  int    i, vlimit;
  EventHandle evtHand;

  printf("Starting setup...\n");
```

```
// Open the connection and the port.
if (!RemOpenConnection(ConnectLocal, 0, 0, &sConn)) {
  printf("Error opening connection.\n");
  return FALSE;
}
sPort = VdrOpenPortConnection(sConn);
if (!sPort) {
  printf("Error getting port.\n");
  return FALSE;
}

// Is this NTSC or PAL?
switch (CfgGetStandard(sConn)) {
  case PCI_PAL_625_MODE:
    videoFormat = Format625_50Hz_2To1;
    sRate = 50;
    break;
  case PCI_NTSC_525_MODE:
    videoFormat = Format525_60Hz_2To1;
    sRate = 60;
    break;
  case PCI_INVALID_MODE:
  default:
    printf("Invalid or unknown video mode.\n");
    return FALSE;
}
VdrSetVideoFormat(sPort, videoFormat);

// Get the necessary resources--1 codec, 1 video output, 1 video input.

// Find first available codec.
vlimit = CfgGetNumCodecs(sConn, JpegCodec);
for (i=0; i<vlimit && !sResHdls[VCOD]; i++) {
  sResHdls[VCOD] = VdrAllocateResource(sPort, ResourceJpegCodec, i);
}
if (!sResHdls[VCOD]) {
  printf("Cannot allocate jpeg codec.\n");
  return FALSE;
}

// Get video in and out resources.
sResHdls[VOUT] = VdrAllocateResource(sPort, ResourceVideoOutput, NUM_OUTPUT);
if (!sResHdls[VOUT]) {
  printf("Cannot allocate video output\n");
  return FALSE;
}
sResHdls[VIN] = VdrGetResourceConnectionHandle(sPort,
ResourceVideoInput, NUM_INPUT);
if (!sResHdls[VIN]) {
  printf("Cannot get video input.\n");
  return FALSE;
}

// Set the default event.
if (!VdrDefaultEvent(sPort, NULL, EventConnectResources, sResHdls[VIN],
sResHdls[VOUT])) {
   printf("Cannot schedule default event.\n");
  return FALSE;
}

// Schedule the event.
evtHand = VdrScheduleEvent(sPort, INT_MIN, EventConnectResources, sResHdls[VCOD],
  sResHdls[VOUT]);
```

```
  if (!evtHand) {
    printf("Cannot schedule event.\n");
    return FALSE;
  }

  return TRUE;

} // SetupResources


//
// Cleanup by releasing resources and closing the control port.
//
void Cleanup()
{
  int i;

  printf("Starting cleanup...\n");

  for (i=0; i<MAX_RSRC; ++i) {
    if (sResHdls[i]) {
      VdrReleaseResource(sResHdls[i]);
      sResHdls[i] = 0;
    }
  }

  if (!VdrClosePort(sPort)) {
    printf("Cannot close port\n");
    return;
  }
  sPort = 0;

} // Cleanup

//
// Play the movie clip.
//
void StartPlay()
{
  int oldpos, newpos;
  MovieToken movieTok;
  TrackToken trackTok;
  MediaToken mediaTok;
  MediaHandle mediaHdl;
  char mediaFilePath[PDR_MAX_COMPLEX_MEDIA_NAME_LEN+1];
  int duration;

  // Open the movie.
  movieTok = PdrOpenMovie(sConn, spMovieName, 0);
  if (!movieTok) {
    printf("Movie %s does not exist \n", spMovieName);
    return;
  }

  // Get the first track token for this movie.
  trackTok = PdrGetNextTrack(movieTok, 0);
  if (!trackTok) {
    printf("Can not get a track token.\n");
    return;
  }

  // Get the first media token on this track.
  mediaTok = PdrGetNextMediaToken(movieTok, trackTok, PdrNullMediaToken());
```

```
 if (PdrMediaTokenIsNull(mediaTok)) {
   printf("First track media token is null.\n");
   return;
 }

 // Get the pathname for the media we have selected.
 if (PdrGetMediaPath(mediaTok, mediaFilePath,
PDR_MAX_COMPLEX_MEDIA_NAME_LEN) == -1) {
   printf("Error getting media path.\n");
   return;
 }

 duration = sSeconds * sRate;

 if (sMarkIn || sMarkOut) {
   mediaHdl = VdrAttachFittedMediaWithMarks(mediaFilePath, sResHdls[VCOD],
duration, NULL, ShiftAfter, sMarkIn, sMarkOut);
 }
 else {
   mediaHdl = VdrAttachFittedMedia(mediaFilePath, sResHdls[VCOD], duration,
NULL, ShiftAfter);
 }
 if (!mediaHdl) {
   printf("Cannot attach the media.\n");
   return;
 }

 // Cue up playback of media that has been attached with VdrAttachMedia.
 if (!VdrCuePlay(sPort, SHUTTLE_RATE)) {
   printf("Cannot cue play.\n");
   return;
 }

 // Begin motion playback.
 if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
   printf("Cannot begin playback.\n");
   return;
 }

 printf("Starting playback...\n");

 //
 // Wait while movie plays.
 // When newpos and oldpos are the same, we're done playing out.
 //
 newpos = 0;
 do {
   oldpos = newpos;
   Sleep(100);            // wait 1/10th second
   newpos = VdrGetPosition(sPort);
 } while (newpos > oldpos);

 // Cease play back.
 if (!VdrIdle(sPort)) {
   printf("Cannot move to idle.\n");
   return;
 }

 // Detach the moviehandle from the channel.
 if (!VdrDetachMedia(mediaHdl, ShiftAfter)) {
   printf("Cannot detach media.\n");
   return;
 }
```

```
  if (!PdrCloseMovie(movieTok)) {
    printf("Cannot close movie.\n");
    return;
  }
} // StartPlay

//
// The main entry point.
//
void main(int argc, char *argv[])
{

  int i = 1;

   // Read in the required movie name.
  if (argv[i]) {
    spMovieName = argv[i];
  }
  else {
    Usage(argv[0]);
    exit(1);
  }
  i++;

  sSeconds = atoi(argv[i++]);

  // Process optional markin and markout points.
  while (i < argc) {
    if (argv[i][0] == '-')
      switch (argv[i][1]) {
      case 'i':
        i++;
        sMarkIn = atoi(argv[i++]);
        break;
      case 'o':
        i++;
        sMarkOut = atoi(argv[i++]);
        break;
      default:
        Usage(argv[0]);
        exit(1);
      }
    else {
      Usage(argv[0]);
      exit(1);
    }
  }

  if (SetupResources()) {
    StartPlay();
  }

  Cleanup();

} // main
```

Playing a list of movies

Example 4, play_multi.c shows you how to play a list of movies in the JPEG compression format. First, the command line is parsed for a list of movies. Up to five movies are concatenated together and stored in an array. Each movie will be played in its entirety, immediately followed by the next movie.

As always, the appropriate resources must be allocated for later use. Therefore, the program establishes a connection to the local machine with **RemOpenConnection**, and opens a port using **VdrOpenPortConnection**. Next, the program determines the standard video format for the machine with **CfgGetStandard**, and with **VdrSetVideoFormat**, sets the format.

Then the code allocates a JPEG codec resource with **VdrAllocateResource**, and attached to the port with a call to **VdrGetResourceConnectionHandle**. The codec allows video decoding to occur on the specified port. Similarly, the program allocates two audio codecs, again with **VdrAllocateResource**, one for each stereo channel. The video input and output resources are then allocated and attached to the port using similar calls. Finally, default events (**VdrDefaultEvent**) and scheduled events (**VdrScheduleEvent**) are setup that describe the connections that occur when the port is in various states. (See *Example 9, stateevt.c* on page 100 in for more information about these events and how to gain more control over their behavior.)

Once the resources are obtained, the code performs the playback. The program opens each movie with **PdrOpenMovie** and a call to the function **VdrAttachOpenMovie** attaches each movie to the timeline. In this example, each movie in succession is attached to the end of the timeline. Other applications could use different insertion methods to place movies in a different order, shifting before or after other media files. After attachment, the port is cued for play with **VdrCuePlay**. It begins playing the movies in succession when the Profile is instructed to shuttle (**VdrShuttle**). After all the movies have finished playing, idle mode is set with **VdrIdle** and the movies are detached with **VdrDetachMovie** and closed with **PdrCloseMovie**.

Finally, the resources acquired during execution must be released. The Profile is told to release all of the handles acquired during the setup phase (**VdrReleaseResource**), and the port is closed (**VdrClosePort**).

Example 4. play_multi.c

```
//
// File: play_multi.c
// This sample program plays up to 5 specified JPEG clips.
//
// Copyright (c) Grass Valley Group Inc. This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// Usage: play_multi movie_name1 [... movie_name5]
//
//
#include <stdio.h>
#include <windows.h>
#include <limits.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>

#define SHUTTLE_RATE 1.0
#define NUM_INPUT   0
#define NUM_OUTPUT  0
#define MAX_CLIPS   5

// For demo application, we will have several resources. Enumerate
// them for use as indeces into an array for VdrAttachMovie calls.
// First three are Codecs.
enum CodecResEnum { VCOD, ACOD1, ACOD2, MAX_CODEC };
enum ResEnum { VIN = MAX_CODEC, VOUT, AOUT1, AOUT2, MAX_RSRC };

// Module static variables.
static ConnectHandle  sConn;
static VdrHandle    sPort;
static char*      spMovieNames[MAX_CLIPS];
static ResourceHandle  sResHdls[MAX_RSRC];

//
// Print out usage line.
//
void Usage(const char* progName)
{
  printf("Usage: %s movie_name1 [... movie_name%d]\n", progName, MAX_CLIPS);

} // Usage


//
// Initialize the Profile. Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources(void)
{
  BOOL  rtn;
  VideoFormat videoFormat;
  int    i, vlimit;
  EventHandle evtHand;

  printf("Starting setup...\n");

  // Open the connection and the port.
```

```
rtn = RemOpenConnection(ConnectLocal, 0, 0, &sConn);
if (!rtn) {
  printf("Error opening connection.\n");
  return FALSE;
}
sPort = VdrOpenPortConnection(sConn);
if (!sPort) {
  printf("Error getting port \n");
  return FALSE;
}

// Is this NTSC or PAL?
switch (CfgGetStandard(sConn)) {
  case PCI_PAL_625_MODE:
    videoFormat = Format625_50Hz_2To1;
    break;
  case PCI_NTSC_525_MODE:
    videoFormat = Format525_60Hz_2To1;
    break;
  case PCI_INVALID_MODE:
  default:
    printf("Invalid or unknown video mode.\n");
    return FALSE;
}
VdrSetVideoFormat(sPort, videoFormat);

//
// Now, get the necessary resources for the demo.
//

// Find first available codec.
vlimit = CfgGetNumCodecs(sConn, JpegCodec // same as VideoCodec);
for (i=0; i<vlimit && !sResHdls[VCOD]; i++) {
  sResHdls[VCOD] = VdrAllocateResource(sPort,
  ResourceJpegCodec, // same as ResourceVideoCodec
  (unsigned int)i);
}

if (!sResHdls[VCOD]) {
  printf("Cannot allocate jpeg video codec.\n");
  return FALSE;
}

// Get two audio codecs.
sResHdls[ACOD1] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT);
 sResHdls[ACOD2] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT+1);
if (!sResHdls[ACOD1] || !sResHdls[ACOD2]) {
  printf("Cannot allocate audio codec.\n");
  return FALSE;
}

// Get video in and out resources.
sResHdls[VOUT] = VdrAllocateResource(sPort, ResourceVideoOutput, NUM_OUTPUT);
if (!sResHdls[VOUT]) {
  printf("Cannot allocate video output.\n");
  return FALSE;
}

sResHdls[VIN] = VdrGetResourceConnectionHandle(sPort, ResourceVideoInput,
NUM_INPUT);
if (!sResHdls[VIN]) {
  printf("Cannot get video input.\n");
  return FALSE;
```

```
  }

  // Get audio resources.
   sResHdls[AOUT1] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT);
   sResHdls[AOUT2] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT+1);
  if (!sResHdls[AOUT1] || !sResHdls[AOUT2]) {
    printf("Cannot allocate audio resources.\n");
    return FALSE;
  }

  // Set the default event.
  if (!VdrDefaultEvent(sPort, NULL, EventConnectResources,
sResHdls[VIN], sResHdls[VOUT])) {
    printf("Cannot schedule default event.\n");
    return FALSE;
  }

  // Schedule the event.
  evtHand = VdrScheduleEvent(sPort, INT_MIN, EventConnectResources, sResHdls[VCOD],
sResHdls[VOUT]);
  if (!evtHand) {
    printf("Cannot schedule event.\n");
    return FALSE;
  }

  return TRUE;

} // SetupResources


//
// Cleanup by releasing resources and closing the control port.
//
void Cleanup(void)
{
  int i;

  printf("Starting cleanup...\n");

  for (i=0; i<MAX_RSRC; ++i) {
    if (sResHdls[i]) {
       VdrReleaseResource(sResHdls[i]);
       sResHdls[i] = 0;
    }
  }

  if (!VdrClosePort(sPort)) {
    printf("Cannot close port. \n");
    return;
  }
  sPort = 0;

} // Cleanup

//
// Play the movie clip.
//
void StartPlay(void)
{
  int oldpos, newpos;
  int i;
  MovieToken movieToks[MAX_CLIPS];
  MovieHandle movieHdls[MAX_CLIPS];
```

```
  memset(movieToks, 0, sizeof(movieToks));
  memset(movieHdls, 0, sizeof(movieHdls));

  i = 0;
  while (spMovieNames[i] && i < MAX_CLIPS) {
    // Open and attach the movies.
    movieToks[i] = PdrOpenMovie(sConn, spMovieNames[i], 0);
    if (!movieToks[i]) {
      printf("Movie %s does not exist \n", spMovieNames[i]);
      return;
    }

// NULL in this next command attaches the clips one after another. It could
// be replaced with the moviehandle of the movie "before" which the new one
// would be inserted.
    movieHdls[i] = VdrAttachOpenMovie(movieToks[i], MAX_CODEC, sResHdls, NULL,
                    ShiftAfter, MarkLongest);
    if (!movieHdls[i]) {
      printf("Error getting movie handle for %s.\n", spMovieNames[i]);
      return;
    }
    i++;
  }

  // Cue up playback of media attached with VdrAttachMovie.
  if (!VdrCuePlay(sPort, SHUTTLE_RATE)) {
    printf("Cannot cue play \n");
    return;
  }

  // Begin motion playback.
  if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
    printf("Cannot begin playback \n");
    return;
  }

  printf("Starting playback...\n");

  // Wait while movie plays.
  // When newpos and oldpos are the same, we're done playing out.
  newpos = 0;
  do {
    oldpos = newpos;
    Sleep(100);            // wait 1/10th second
    newpos = VdrGetPosition(sPort);
  } while (newpos > oldpos);

  // Cease play back.
  if (!VdrIdle(sPort)) {
    printf("Cannot move to idle.\n");
    return;
  }

  i = 0;
  while (spMovieNames[i] && i < MAX_CLIPS) {

    // Detach the movie handle from the channel.
    if (!VdrDetachMovie(movieHdls[i], ShiftAfter)) {
      printf("Cannot detach movie %s.\n", spMovieNames[i]);
      return;
    }
```

```
    if (!PdrCloseMovie(movieToks[i])) {
      printf("Cannot close movie %s.\n", spMovieNames[i]);
      return;
    }
    i++;
  }

} // StartPlay


//
// The main entry point.
//
void main(int argc, char *argv[])
{
 BOOL rtn;
 int i = 1;

  // Read in the movie names.

  if (argc > 6) {
    printf("Too many clips.\n");
    Usage(argv[0]);
    exit(1);
  }
  if (!argv[i]) {
    printf("No clip names.\n");
    Usage(argv[0]);
    exit(1);
  }
// Process seconds argument.
  while (i <= 5 && argv[i]) {
    spMovieNames[i-1] = argv[i];
    ++i;
  }

  rtn = SetupResources();

  if (rtn)
    StartPlay();

  Cleanup();

} // main
```

Playing a movie using Central Resource Management

Profile System Software 4.0 includes a new Configuration Manager that introduces new resource management concepts. Configuration Manager now allows the user to create channels, pre-defined groups of resources that can be used by ports for record and play operations. Software applications no longer need to allocate individual video, audio, and timecode resources to a port. They can allocate a channel that was created in Configuration Manager, thus avoiding the complexity of allocating individual resources, and benefiting from the Resource Manager's resource conflict reporting.

Example 5, playcrm.cpp shows you how to allocate a channel to a port, how to play a movie on that port, then close the port and free up the resources used by the channel. Note that this process is independent of the compression format. The channel definition specifies the codec type, and the attached movie type must match the codec type.

First, the appropriate resources must be allocated for later use. **VdrGetNumChannelDefs** is used to obtain the number of entries in the channel definition list. Then the definitions are read using **VdrGetChannelInfoList**. **VdrAllocateChannel** assigns the resources defined by a channel to a port and opens that port. In this example, the first channel is arbitrarily assigned to a port, although any named channel definition from the list could be used. Note that the application name is provided to assist in subsequent resource conflict reporting. If the port is not successfully opened, the channel status if tested for problems or resource conflicts using the structure defined in *crmtypes.h* and returned by **VdrAllocateChannel** and **VdrGetChannelInfoList**.

Once the channel is assigned, you proceed to the playback. The movie is opened via **PdrOpenMovie** and is attached with the function **VdrAttachOpenMovie**. After that, the movie is cued for play (**VdrCuePlay**). It begins playing when the video server is instructed to shuttle using **VdrShuttle**, and after the clip has finished playing—determined by **VdrGetPosition**—idle mode is set with **VdrIdle** and the movie is detached using **VdrDetachMovie**. The movie is closed with **PdrCloseMovie**.

Finally, the resources acquired during execution must be released. With a call to **VdrClosePort**, the Profile unit is told to release all of the resource handles acquired during the setup phase, and the port is closed. This completes the process and the program is stopped.

If an error occurs at any step of the way, an appropriate error message is output to the display, so that troubleshooting may take place.

Example 5. playcrm.cpp

```
/* Copyright (c) Grass Valley Group Inc. This program, or portions thereof,
 * is protected as an unpublished work under the copyright laws of
 * the United States.
 *
 * This source code is only intended as a sample code to demonstrate the
 * usage of the Profile XP API.
 */

#include <stdio.h>
#include <windows.h>
#include <tekrem.h>
#include <crmtypes.h>
#include <tekpdr.h>
#include <tekvdr.h>
#include <vdrerror.h>


/* 1) Open connection to ProfileXP
 * 2) Allocate Resources
 * 3) Play MPEG clip
 * 4) Shutdown
 */

/* Module static variables. */
static ConnectHandle  hConn = LOCAL_CONNECTION;
static VdrHandle    hVdrPort = 0;
static ChannelInfochanInfo;

/*****************************************************************************
 * SetupResources
 *
 * Purpose: Set up the appropriate resources for playback
 *****************************************************************************/
BOOL SetupResources()
{
  printf("Setup resources.\n");

  /* USE IF REMOTING: Open the connection and the port.
  if (!RemOpenConnection(ConnectLocal, 0, 0, &hConn))
 {
    printf("Error opening connection.\n");
    return FALSE;
  }
 */

 /* Get channel definitions */
 int iChanDefCount = VdrGetNumChannelDefs(hConn);

 if ( iChanDefCount <= 0 )
 {
     printf("No channel definitions exist on this Profile.");
     return FALSE;
 }

 int iBufferSize = sizeof(ChannelInfo) * iChanDefCount;
 ChannelInfo* pChanInfoList = (ChannelInfo*) malloc(iBufferSize);
 int iFetched;
 if ( (!VdrGetChannelInfoList(hConn, pChanInfoList, iBufferSize, iChanDefCount, &iFetched)) ||
     (iFetched != iChanDefCount) )
 {
     // Failed to get list, or the number fetched does not equal the number requested.
```

```
        printf("Failed to get ChannelInfo list.");
        free(pChanInfoList);
        return FALSE;
    }

    /*  Allocate the first channel in the list.
        Note that this function opens a port, allocates all resources specified
        in the channel definition, and connects the resources on each track.
        The channel name is "playcrm"*/
    hVdrPort = VdrAllocateChannel(hConn, pChanInfoList[0].name, 0x00, "playcrm", &chanInfo);

    if ( hVdrPort )
    {
       printf("Successfully allocated channel '%s' (port handle %d).\n", pChanInfoList[0].name,
    hVdrPort);
    }
     else
    {
        if ( chanInfo.status & CHAN_STATUS_IN_USE )
        {
            printf("Channel '%s' is in use by '%s' on '%s'",
            pChanInfoList[0].name, chanInfo.applicationName, chanInfo.hostName);
        }
        if ( chanInfo.status & CHAN_STATUS_CONFLICTS )
        {
            printf("Some resources are in use by another application.");
        }
        else
        {
            switch ( GetLastError() )
            {
            case VDR_ERROR_CREATE_DICTIONARY:
            case VDR_ERROR_GET_CHANNEL_DEFINITION:
            printf("Could not find configuration for '%s'", pChanInfoList[0].name);
            break;
            case VDR_ERROR_ADD_RESOURCE_OWNER:
            case VDR_ERROR_RESMON_CREATE_FAILED:
            printf("Unable to register resource use.");
            break;
            case VDR_ERROR_PORT_OPEN_FAILED:
            printf("The system is out of ports.");
            break;
            default:
            printf("Unable to allocate channel.");
            }

        }

        free(pChanInfoList);
        return FALSE;
     }

    free(pChanInfoList);
     return TRUE;
    }
```

```
/*****************************************************************************
 * Cleanup
 *
 * Purpose: Release all the resources
 *****************************************************************************/
void Cleanup(void)
{
  printf("Start cleanup...\n");

  if (!VdrClosePort(hVdrPort)) {
    printf("Error closing port. \n");
    return;
  }
  hVdrPort = 0;
}
/*****************************************************************************
 * StartPlay
 *
 * Purpose: Play the clip
 *****************************************************************************/
void StartPlay(char *movieName)
{
  INT     oldpos, newpos = 0;
  MovieTokenmovieToken = 0;
  MovieHandlemovieHandle = 0;

  // Open and attach the movie.
  movieToken = PdrOpenMovie(hConn, movieName, 0);
  if (!movieToken) {
    printf("Movie %s does not exist \n", movieName);
    return;
  }

  movieHandle = VdrAttachOpenMovie(movieToken, chanInfo.resourceCount,
         chanInfo.resourceHandles, NULL, ShiftAfter, MarkLongest);
  if (!movieHandle) {
    printf("Error getting movie handle \n");
    return;
  }

  // Cue up playback of media attached with VdrAttachMovie.
  if (!VdrCuePlay(hVdrPort, 0.0)) {
    printf("Cannot cue play \n");
    return;
  }

  // Begin motion playback.
  if (!VdrShuttle(hVdrPort, 1.0)) {
    printf("Cannot begin playback \n");
    return;
  }

  printf("Playing movie \"%s\"...\n", movieName);

  // Wait while movie plays.
  // When newpos and oldpos are the same, we're done playing out.
  newpos = 0;
  do {
    oldpos = newpos;
    Sleep(100);            // wait 1/10th second
    newpos = VdrGetPosition(hVdrPort);
  } while (newpos > oldpos);
```

```
  // Cease play back.
  if (!VdrIdle(hVdrPort)) {
    printf("Cannot move to idle.\n");
    return;
  }

 printf("Detaching movie \"%s\"...\n", movieName);
  // Detach the movie handle from the channel.
  if (!VdrDetachMovie(movieHandle, ShiftAfter)) {
    printf("Cannot detach movie.\n");
    return;
  }

  if (!PdrCloseMovie(movieToken)) {
    printf("Cannot close movie.\n");
    return;
  }

}
/*****************************************************************************
 * main
 *
 * Purpose: Main entry point
 *****************************************************************************/
void main()
{

 char movieName[PDR_MAX_MOVIE_NAME_LEN+1];
 memset(movieName, 0, sizeof(movieName));

 /* Get the movie and channel name*/
 printf("Usage: playcrm [mpeg movie name]\n");
 printf( "Enter a valid movie name (i.e. INT:/default/#1):\n");
  scanf( "%s",movieName);

  if (SetupResources() != FALSE) {
    StartPlay(movieName);
  }

 Sleep(5000);

  Cleanup();
}
```

Chapter 4

Using the Profile Media File System

The movies that you record on a Profile video file server or disk recorder are stored on a set of high-capacity ultra SCSI disks. This set of disks is called a *dataset*, not unlike a volume on a personal computer. A volume on a PC usually resides on a single disk, but a dataset always consists of multiple disks.

Datasets contain collections of movies called *groups*. This is analogous to a directory on a PC file system. Movies are stored in one or more media files which contain digitized JPEG video, MPEG video, audio, or timecode.

This chapter discusses browsing the inventory of movies on a Profile disk recorder, viewing information and characteristics of movies in the inventory, and checking free disk file space.

Each explanation of coding methods is followed by a sample program written in C.

Browsing the media file system

Example 6, browse.c demonstrates the use of API calls that print a local or remote Profile's entire inventory, similar to the Unix `ls` and MS-DOS `dir` commands. DVCPRO 25, DVCPRO 50, MPEG or JPEG format is also noted, using the attributes element of the PdrMovieState structure.

An optional `-v` flag on the command line specifies verbose mode, which provides more detail about each movie in the inventory. (For more specific information about a single movie file, see *Example 7, viewCMF.c* on page 88.)

Once the connection is established with **RemOpenConnection**, the sample code function IdentifyInventory performs the majority of the work. **PdrFindFirstDataset** and **PdrFindNextDataset** walk through all of the available datasets. **PdrFindFirstGroup** and **PdrFindNextGroup** are used to walk through all of the available groups within each dataset. Similarly, **PdrFindFirstMovie** and **PdrFindNextMovie** are used to walk through each movie within a group. **PdrCloseFind** ends each walk-through.

The sample code function PrintMovieState uses **PdrGetMovieStateInfo** to retrieve a PdrMovieState structure, which it then uses to format some basic information about a movie.

Example 6. browse.c

```
//
// File: browse.c
// This sample program enumerates all clips in the inventory.
// It is able to work either locally or remotely.
//
// Copyright (c) Grass Valley Group Inc., 1996-1998.
//     All rights reserved.
//
// Usage: browse [hostname]
//
#include <windows.h>
#include <stdio.h>
#include "tekpdr.h"
#include "pdrerror.h"
#include "tekrem.h"
#include "tekvfs.h"

static const char *sMonth[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

static BOOL sVerbose = FALSE;

//
// This function is used if the input is either
// a ? or the word help. It shows the usage of the program.
//
void Usage(const char *progName)
{
    printf("Usage:\n");
    printf("%s [-v] [hostname]\n", progName);

    printf("   -v        Verbose\n");
    printf("   hostname  Machine name to act upon remotely\n");
    printf("\n");
    printf("%s ? :    Causes this message to be printed\n", progName);
    printf("%s help : Causes this message to be printed\n", progName);
    printf("\n");
    printf("This tool enumerates movies in the inventory of a Profile.\n");

} // Usage


//
// Convert a FILETIME to a string, using the array of month strings above.
//
void MakeTimeString(FILETIME *ft, char *string)
{
    SYSTEMTIME st;
    WORD thisYear, thisMonth;

    GetLocalTime(&st);
    thisYear = st.wYear;
    thisMonth= st.wMonth;
    FileTimeToSystemTime(ft, &st);
```

```
    if ((thisYear == st.wYear) ||
        ((thisYear == (st.wYear+1)) && (thisMonth < st.wMonth))) {
        sprintf(string, "%3s %2d %02d:%02d",
            sMonth[st.wMonth-1], st.wDay, st.wHour, st.wMinute);
    }
    else {
        sprintf(string, "%3s %2d, %4d",
            sMonth[st.wMonth-1], st.wDay, st.wYear);
    }

} // MakeTimeString


//
// This function gets and prints the movie video format(s) for an individual
// movie. When run in verbose mode, the movie state information is printed.
//
void PrintMovieState(ConnectHandle lch, const char* movieName)
{
    char  create[16];
    char  change[16];
    PdrMovieState state;

    BOOL ret = PdrGetMovieState(lch, movieName, &state);

    if (ret) {
        if (state.attributes & PdrVideoFormatJPEG) printf("JPEG");
        if (state.attributes & PdrVideoFormatMPEG) printf("MPEG");
        printf("\n");
    }
    if (ret && sVerbose) {
        printf("         > Length- min=%d, max=%d \n",
           state.minLength, state.maxLength);
        MakeTimeString(&state.createTime, create);
        MakeTimeString(&state.lastChangedTime, change);
        printf("         > Time- created: %s, changed: %s \n",
           create, change);
        printf("         > #Tracks- V=%d, A=%d, TC=%d \n",
           state.numV, state.numA, state.numT);
        printf("         > Attr.- Open-%s, OpenMultiple-%s, OpenExclusive-%s,\n",
           (state.attributes & PdrOpen)? "Yes":"No",
           (state.attributes & PdrOpenMultiple)? "Yes":"No",
           (state.attributes & PdrOpenExclusive)? "Yes":"No");
        printf("         >      - ReadOnly-%s, CntlRO-%s, Locked-%s,\n",
           (state.attributes & PdrReadOnly)? "Yes" : "No",
           (state.attributes & PdrControlRO)? "Yes" : "No",
           (state.attributes & PdrLocked)? "Yes" : "No");
        if (state.attributes & PdrOpenExclusive) {
            printf("         > Exclusive open by process 0x%x\n",
               state.exclusivePID);
        }
        printf("         \n");
    }
    if (!ret) {
        printf("\n");
        printf("Failed to get movie state for \"%s\"\n", movieName);
        printf("Error code is 0x%x\n", GetLastError());
    }

} // PrintMovieState
```

```
//
// Churn through datasets, groups and movies.
//
void IdentifyInventory(ConnectHandle lch)
{
    EnumToken dset, grp, movie;
    char  dataset[PDR_MAX_DSET_NAME_LEN+1];
    char  group[PDR_MAX_GROUP_NAME_LEN+1];
    char  name[PDR_MAX_MOVIE_NAME_LEN+1];
    char  startName[PDR_MAX_COMPLEX_MOVIE_NAME_LEN+1];

    dset = PdrFindFirstDataset(lch, dataset, sizeof(dataset));
    if (dset <= 0) {
        printf("Failed to find the first dataset\n");
        printf("Error code is 0x%x\n", GetLastError());
        return;
    }
    do {
        // Process dataset information.
        printf("%s/\n", dataset);

        // Now look for groups within this dataset.
        strcpy(startName, dataset);
        grp = PdrFindFirstGroup(lch, startName, group, sizeof(group));
        if (grp <= 0) {
            printf("Failed to find the first group in dataset %s\n", dataset);
            printf("Error code is 0x%x\n", GetLastError());
            continue;
        }
        do {
            // Process group information.
            printf("  --/%s/\n", group);

            // Now look for movies within this group.
            sprintf(startName, "%s/%s/", dataset, group);
            movie = PdrFindFirstMovie(lch, startName, name, sizeof(name));
            if (movie <= 0) {
                printf("Failed to find first movie in %s\n", startName);
                printf("Error code is 0x%x\n", GetLastError());
                continue;
            }
            do {
                // Process movie information.
                printf("     ---/%s - ", name);

                sprintf(startName, "%s/%s/%s",
                    dataset, group, name);

                PrintMovieState(lch, startName);

            } while (PdrFindNextMovie(movie, name, sizeof( name)));

            if (GetLastError() != PDR_FIND_END) {
                printf("Failed to find next movie\n");
                printf("Error code is 0x%x\n", GetLastError());
            }

            PdrCloseFind(movie);

        } while (PdrFindNextGroup(grp, group, sizeof( group)));
```

```
        if (GetLastError() != PDR_FIND_END) {
            printf("Failed to find next group\n");
            printf("Error code is 0x%x\n", GetLastError());
        }
        PdrCloseFind(grp);

    } while (PdrFindNextDataset(dset, dataset, sizeof(dataset)));

    if (GetLastError() != PDR_FIND_END) {
        printf("Failed to find next dataset\n");
        printf("Error code is 0x%x\n", GetLastError());
    }
    PdrCloseFind(dset);

} // IdentifyInventory

//
// In the main function, we determine if the Profile is remote, and
// open the connection if it is; otherwise, the connection stays local.
// Then, the Inventory function is invoked.
// We also allow ? or help on the command line in order to get the
// usage printout.
//
main(int argc, char *argv[])
{
    ConnectHandle ch = LOCAL_CONNECTION;
    const char *chDesc = "LOCAL_CONNECTION";
    BOOL success;
    int hostArg=1;

    if (argc > 1) {
        // If user wants help, give it right away.
        if ((strcmp(argv[1], "help") == 0)
         || (strcmp(argv[1], "?") == 0)) {
            Usage(argv[0]);
            return 0;
        }

        // Process optional verbose flag.
        if (!strcmp(argv[1], "-v")) {
            sVerbose = TRUE;
            ++hostArg;
        }

        // Process optional specific host name (otherwise it's local).
        if (argc > hostArg) {
            success = RemOpenConnection(ConnectEthernet, 0, argv[hostArg], &ch);
            if (success) {
                 chDesc = argv[hostArg];
            }
            else {
                printf("Failed to make a connection to %s\n", argv[hostArg]);
                printf("Error code is 0x%x\n", GetLastError());
                Usage(argv[0]);
                return 0;
            }
        }
    }
    printf("Inventory List of %s\n", chDesc);

    printf("Interfacing to tekpdr version %d.%d\n",
        PdrGetMajorVersion(ch), PdrGetMinorVersion(ch));
```

```
    IdentifyInventory(ch);
    success = RemCloseConnection(ch);
    if (!success) {
        printf("Failed to close the connection to %s\n", chDesc);
        printf("Error code is 0x%x\n", GetLastError());
    }
    return 0;

} // main
```

Viewing CMF information

The Profile API contains a number of functions to help determine specific information about a CMF (Common Movie Format) file. *Example 7, viewCMF.c* uses various functions by determining and displaying information about a single movie stored on a local or remote Profile. To see how to list all available movies on a Profile system, see *Browsing the media file system* on page 81.

First, the connection is established with **RemOpenConnection**. The program then determines if the movie exists with **PdrMovieExists** and can be opened; if not, appropriate error messages are generated. Then **PdrGetMovieAttributes** determines if the movie is simple or complex. **PdrGetMovieStateInfo** fetches a PdrMovieState structure, which is then formatted for output.

Next, the program walks through each track in the movie, and then walks through each media token for each track. Information about each track is presented after querying the functions **PdrGetNumMediaOnTrack**, **PdrGetTrackLength**, **PdrGetTrackTokenNum**, and **PdrGetTrackTokenType**. For each media token, **PdrGetMediaState** fetches a PdrMediaState structure, which is then formatted to output.

For each media file, the TekVfs library determines miscellaneous information about the file. The size, type, format, modification time, default marks and other attributes are determined using **VfsFindFirstFile**, **VfsGetFileType**, **VfsGetFileVideoFormat**, **VfsGetFileModificationTime**, **VfsGetFileDefaultMarks**, and **VfsGetFileAttributes**.

From examining the source code and running the program on a few target movies, you can see that there is a fair amount of information that can be obtained about a single movie file. The PdrMovieState and PdrMediaState structures contain such information as JPEG compression quantization factors, MPEG group of picture (GOP) values, audio gain settings, video sample rates, and timecode formats.

Example 7. viewCMF.c

```
//
// File: ViewCMF.c
// Sample code to view all aspects of a movie in the inventory.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//

#include <windows.h>
#include <stdio.h>
#include "tekpdr.h"
#include "tekrem.h"
#include "tekvfs.h"

static const char *month[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

//
// Print out usage line.
//
void Usage(const char* progName)
{
    printf("Usage:\n");
    printf("%s complexMovieName : \n", progName);
    printf("shows characteristics of the named movie.\n");
    printf("%s -h hostname complexMovieName : \n", progName);
    printf("shows characteristics of the named movie on the machine hostname\n");
    printf("%s ?     : causes this message to be printed\n", progName);
    printf("%s help  : causes this message to be printed\n", progName);
    printf("%s usage : causes this message to be printed\n", progName);

} // Usage

//
// Closes the connection (if remote) and exits.
//
void KillConnectionAndExit(ConnectHandle ch, char* hostname)
{
    if (ch != LOCAL_CONNECTION) {
        if (!RemCloseConnection(ch)) {
            printf("Failed to close the connection to %s\n", hostname);
            printf("Error code is 0x%x\n", GetLastError());
        }
    }
    exit(0);

} // KillConnectionAndDie

//
```

```
// Converts a FILETIME to a string. Uses the array of month strings above.
//
void MakeTimeString(FILETIME *ft, char *string)
{
    SYSTEMTIME st;
    WORD y, m;

    GetLocalTime(&st);
    y  = st.wYear;
    m = st.wMonth;
    FileTimeToSystemTime(ft, &st);
    if (y == st.wYear || (y == st.wYear+1 && m < st.wMonth)) {
        sprintf(string, "%3s %2d %02d:%02d",
            month[st.wMonth-1], st.wDay, st.wHour, st.wMinute);
    }
    else {
        sprintf(string, "%3s %2d, %4d", month[st.wMonth-1], st.wDay, st.wYear);
    }

} // MakeTimeString


//
// Convert a UINT timecode to a string.  It uses the definitions in vdrtypes.h
//
void MakeTimeCodeString(UINT tc, char *string)
{
    if (IS_TIMECODE_INVALID(tc)) {
        sprintf(string, "INVALID TC ");
    }
    else {
        UINT fields =  GET_TIMECODE_FIELDS(tc);
        UINT frames = fields / 2;
        char spacer = '.';
        if (fields & 1) spacer = ':';
        if (GET_TIMECODE_DROPFRAME(tc)) {
            spacer = ',';
            if (fields & 1) spacer = ';';
        }
        sprintf(string, "%02d:%02d:%02d%c%02d",
            GET_TIMECODE_HOURS(tc), GET_TIMECODE_MINUTES(tc),
            GET_TIMECODE_SECONDS(tc), spacer, frames);
    }

} // MakeTimeCodeString


#define LEN_SIZE  20
//
// Prints the information about a single media file from the i960 perspective.
//
void PrintFileInfo(ConnectHandle ch, char* path)
{
    BOOL            success;
    WIN32_FIND_DATA find;
    unsigned long   high, low;
    HANDLE          file;
    DWORD           attributes;
    FileType        type;
    VideoFormat     format;
    FILETIME        time;
    UINT            markIn, markOut;
    char            len[LEN_SIZE];
```

```
char            modtime[16];
char            testbit;
int             bit, i, limit = 64;

success = VfsFileExists(ch, path);
if (!success) {
    printf("The media file '%s' does NOT exist\n", path);
    return;
}
// Use Vfs library to get the file size.
file = VfsFindFirstFile(ch, path, &find);
VfsFindClose(file);

// Miscellaneous information about the file.
attributes = VfsGetFileAttributes(ch, path);
type = VfsGetFileType(ch, path);
format = VfsGetFileVideoFormat(ch, path);
time = VfsGetFileModificationTime(ch, path);
success = VfsGetFileDefaultMarks(ch,
                  path, &markIn, &markOut);

// Can we continue with this file?
if (file == INVALID_HANDLE_VALUE || !success) {
    printf("Vfs error getting file information\n");
    printf("Error code is 0x%x\n", GetLastError());
    return;
}

MakeTimeString(&time, modtime);

// Initialize for 64 bit conversion.
for (i=0; i<LEN_SIZE; i++) len[i] = 0;
high = find.nFileSizeHigh;
low  = find.nFileSizeLow;

// Test each bit of the 64 bit size.
if (!high) {
    high = low;
    low = 0;
    limit = 32;
}
if (!(high & 0xFFFF0000)) {
    high = (high << 16) + (low >> 16);
    low = (low << 16);
    limit -= 16;
}
if (!(high & 0xFF000000)) {
    high = (high << 8) + (low >> 24);
    low = (low << 8);
    limit -= 8;
}
for (bit=0; bit<limit; bit++) {
    testbit = (char)(high >>31) & 1;
    for (i=LEN_SIZE-2; i>=0; i--) {
        len[i] = (len[i] *2) + testbit;
        testbit = 0;
        if (len[i] > 9) {
            len[i] -= 10;
            testbit = 1;
        }
    }
    testbit = (char)(low >>31) & 1;
    high = (high *2) + (testbit);
```

```
        low = (low *2);
    }
    // Change binary to ascii.
    for (i=0; i<LEN_SIZE; i++) len[i] += '0';

    // Change leading 0s to blanks.
    for (i=0; i<LEN_SIZE-2; i++) {
        if (len[i] != '0') break;
        len[i] = ' ';
    }
    len[LEN_SIZE-1] = '\0';

    // Output info.
    printf("The file is of type %s, and format %s.\n",
          (type == FileTypeNonMedia) ? "FileTypeNonMedia" :
          (type == FileTypeJpegVideo) ? "FileTypeJpegVideo" :
          (type == FileTypeMpegVideo) ? "FileTypeMpegVideo" :
          (type == FileTypeAudio) ? "FileTypeAudio" :
          (type == FileTypeTimecode) ? "FileTypeTimecode" : "UNKNOWN",
          (format == Format525_60Hz_2To1) ? "Format525_60Hz_2To1" :
          (format == Format625_50Hz_2To1) ? "Format625_50Hz_2To1" : "UNKNOWN");
    printf("The file attributes (0x%x) are: -FILE_ATTRIBUTE_DIRECTORY %s\n",
          attributes,
          (attributes & FILE_ATTRIBUTE_DIRECTORY) ? "Yes" : "No");
    printf("      -FILE_ATTRIBUTE_NORMAL %s, FILE_ATTRIBUTE_READONLY %s\n",
          (attributes & FILE_ATTRIBUTE_NORMAL) ? "Yes" : "No",
          (attributes & FILE_ATTRIBUTE_READONLY) ? "Yes" : "No");
    printf("The file length is %s bytes\n", len);
    printf("First/Last recorded fields (Default Marks) are- First %d, Last %d\n",
         markIn, markOut);
    printf("File last modified %s\n", modtime);

} // PrintFileInfo
//
// Print info for an individual media segment from the NT perspective.
//
void PrintMediaInfo(PdrMediaState ms)
{
    printf("The media token for this segment is "%d%c%c%02d"\n",
            ms.thisSegment.a, (ms.thisSegment.b >>8) &0xFF,
            ms.thisSegment.b &0xFF, ms.thisSegment.c);
    printf("This media segment is of type %s\n",
            (ms.type == PdrMediaBlack) ? "PdrMediaBlack" :
            (ms.type == PdrMediaFile)  ? "PdrMediaFile"  :
        (ms.type == PdrMediaTrackRecord) ? "PdrMediaTrackRecord" : "UNKNOWN");
    printf("It is at position %d and extends for %d fields\n",
          ms.position, ms.out - ms.in);
    printf("The Pdr Track type is %s\n",
        (ms.track_type == PdrTrackInvalid) ? "Invalid" :
        (ms.track_type == PdrJpegVideoTrack) ? "JpegVideo" :
        (ms.track_type == PdrAudioTrack) ? "Audio" :
        (ms.track_type == PdrTimeCodeTrack) ? "Timecode" :
        (ms.track_type == PdrMpegVideoTrack) ? "MpegVideo" :
        "TrackType Unknown");
    printf("The file IN/OUT points are set at %d/%d\n",
          ms.in, ms.out);
    printf("The segment attributes are (0x%x)\n", ms.attributes);
    printf("      -PdrControlRO %s, -PdrReadOnly %s\n",
        (ms.attributes & PdrControlRO) ? "Yes" : "No",
        (ms.attributes & PdrReadOnly) ? "Yes" : "No");
    if (ms.type == PdrMediaFile) {
        printf("The media file pathname is %s\n", ms.path);
    }
```

```
    if (ms.fileType == FileTypeAudio) {
        printf("Audio ramp control is set as:\n");
        printf("        in-fields %d to target %f; out-fields %d to target %f\n",
            ms.aud.nFields1, ms.aud.aGain1, ms.aud.nFields2, ms.aud.aGain2);
    }
    if (ms.fileType == FileTypeJpegVideo) {
        printf("JPEG Compression Quantization Factors are set as: \n");
        printf("        max Chroma Q %f, max Lumina Q %f\n",
            ms.jpeg.maxChrQ, ms.jpeg.maxLumQ);
        printf("        min Chroma Q %f, min Lumina Q %f\n",
            ms.jpeg.minChrQ, ms.jpeg.minLumQ);
    }
    if (ms.fileType == FileTypeMpegVideo) {
        printf("MPEG parameterization is set as:\n");
        printf("        Chroma Format %s,  GOP End %s\n",
            (ms.mpeg.chroma == MpegChroma420) ? "MpegChroma420" :
            (ms.mpeg.chroma == MpegChroma422) ? "MpegChroma422" :
            "MpegChromaUnknown",
            (ms.mpeg.gopEnd == GopOpenEnd) ? "GopOpenEnd" :
            (ms.mpeg.gopEnd == GopClosedEnd) ? "GopClosedEnd" :
            "GopEndUnknown");
        printf("        Gop Structure is %d IPix per Gop, \n",
            ms.mpeg.iPixPerGop);
        printf("               %d PPix per IPix, %d BPix per IPPix\n",
            ms.mpeg.pPixPerIPix, ms.mpeg.bPixPerIPPix);
        printf("          Pix Type is: %s\n",
            (ms.mpeg.pixStructure == PixStructureFrame) ? "PixStructureFrame" :
            (ms.mpeg.pixStructure == PixStructureField) ? "PixStructureField" :
            "PixStructureUnknown");
    }
    if (ms.fileType == FileTypeTimecode) {
        char firstTc[20];
        char lastTc[20];
        printf("Timecode parameterization is:\n");
        printf("         format is (0x%x) %s\n", ms.tc.format,
            (ms.tc.format == TcFormatDropFrame) ? "DropFrame" :
            (ms.tc.format == TcFormatNonDropFrame) ? "NonDropFrame" :
            "TcFormatUnknown");
        MakeTimeCodeString(ms.tc.firstTimeCode, firstTc);
        MakeTimeCodeString(ms.tc.lastTimeCode, lastTc);
        printf("Timecode Values: first %s,  (0x%x)\n",
            firstTc, ms.tc.firstTimeCode);
        printf("               : last  %s,  (0x%x)\n",
            lastTc, ms.tc.lastTimeCode);
    }
    if (ms.fileType == FileTypeNonMedia) {
        printf("Cached file Type is unknown;  no parameters can be decoded\n");
    }
    printf("Neighboring media tokens are-\n");
    printf("       Previous "%d%c%c%02d", Next "%d%c%c%02d"\n",
       ms.prev.a, (ms.prev.b >>8) &0xFF, ms.prev.b &0xFF, ms.prev.c,
       ms.next.a, (ms.next.b >>8) &0xFF, ms.next.b &0xFF, ms.next.c);
    printf("     ---- RTS file system status of this media file ----    \n");

} // PrintMediaInfo
```

```
//
// Print info for an individual track.
//
void PrintTrackInfo(MovieToken movie, TrackToken track)
{
    int nMedia = PdrGetNumMediaOnTrack(movie, track);
    int len = PdrGetTrackLength(movie, track);
    int num = PdrGetTrackTokenNum(track);
    PdrTrackType type = PdrGetTrackTokenType(track);

    printf("This track is of type %s, and is number %d\n",
        (type == PdrTrackInvalid) ? "Invalid" :
        (type == PdrJpegVideoTrack) ? "JpegVideo" :
        (type == PdrAudioTrack) ? "Audio" :
        (type == PdrTimeCodeTrack) ? "Timecode" :
        (type == PdrMpegVideoTrack) ? "MpegVideo" :
        "UNKNOWN", num);
    printf("The track is %d fields long, and contains %d media segment%s\n",
        len, nMedia, nMedia == 1 ? "" : "s");

} // PrintTrackInfo

//
// Prints the state information for an individual movie.
//
void PrintMovieState(PdrMovieState state)
{
    char  create[16];
    char  change[16];
    char  firstTc[20];
    char  lastTc[20];

    printf("Length- min=%d, max=%d \n", state.minLength, state.maxLength);
    MakeTimeString(&state.createTime, create);
    MakeTimeString(&state.lastChangedTime, change);
    printf("Time- created: %s, changed: %s \n", create, change);
    printf("Tracks- J=%x, M=%x. A=%x, TC=%x \n",
        state.numJ, state.numM, state.numA, state.numT);
    printf("Attributes- 0x%08x\n", state.attributes);
    printf("          - Open-%s, OpenMultiple-%s, OpenExclusive-%s,\n",
       (state.attributes & PdrOpen)? "Yes":"No",
       (state.attributes & PdrOpenMultiple)? "Yes":"No",
       (state.attributes & PdrOpenExclusive)? "Yes":"No");
    printf("     - ReadOnly-%s, CntlRO-%s, Locked-%s,\n",
       (state.attributes & PdrReadOnly)? "Yes" : "No",
       (state.attributes & PdrControlRO)? "Yes" : "No",
       (state.attributes & PdrLocked)? "Yes" : "No");
    printf("     - Construction: Codec %s, Copy %s, Restore %s\n",
        (state.attributes & PdrCodecConstruction)? "Yes":"No",
        (state.attributes & PdrCopyConstruction)? "Yes":"No",
        (state.attributes & PdrRestoreConstruction)? "Yes":"No");
    printf("     - Major Sample Rate 50Hz-%s, 60Hz-%s\n",
       (state.attributes & PdrSampleRate50)? "Yes" : "No",
       (state.attributes & PdrSampleRate60)? "Yes" : "No");
    printf("     - Primary Video Format JPEG-%s, MPEG-%s\n",
       (state.attributes & PdrVideoFormatJPEG)? "Yes" : "No",
       (state.attributes & PdrVideoFormatMPEG)? "Yes" : "No");
    printf("     - Primary Timecode Format NonDF-%s, DropFrame-%s\n",
       (state.attributes & PdrTcNonDropFrame)? "Yes" : "No",
       (state.attributes & PdrTcDropFrame)? "Yes" : "No");
    printf("     - Primary Audio Sample Size 16bit-%s, 20bit-%s\n",
       (state.attributes & PdrAudio16Bit)? "Yes" : "No",
       (state.attributes & PdrAudio24Bit)? "Yes" : "No");
```

```
    if (state.attributes & PdrOpenExclusive) {
        printf("Exclusive open by process 0x%x\n", state.exclusivePID);
    }
    printf("Marks- IN-%d, OUT-%d\n", state.markIn, state.markOut);

    MakeTimeCodeString(state.firstTimeCode, firstTc);
    MakeTimeCodeString(state.lastTimeCode, lastTc);
    printf("Timecode Values: first %s,  (0x%x)\n", firstTc, state.firstTimeCode);
    printf("                 : last  %s,  (0x%x)\n", lastTc, state.lastTimeCode);
    printf("Timecode format appears to be %s\n",
         (GET_TIMECODE_DROPFRAME(state.lastTimeCode)) ?
         "TcFormatDropFrame" : "TcFormatNonDropFrame");

    printf("MPEG parameterization- Chroma Format- %s\n",
        (state.mpegChroma == MpegChroma420) ? "MpegChroma420" :
        (state.mpegChroma == MpegChroma422) ? "MpegChroma422" :
        "MpegChromaUnknown");
    printf("              -GOP End- %s, Pix Type- %s\n",
        (state.mpegGopEnd == GopOpenEnd) ? "GopOpenEnd" :
        (state.mpegGopEnd == GopClosedEnd) ? "GopClosedEnd" :
        "GopEndUnknown",
        (state.mpegPixStructure == PixStructureFrame) ? "PixStructureFrame" :
        (state.mpegPixStructure == PixStructureField) ? "PixStructureField" :
        "PixStructureUnknown");
    printf("    -IPixPerGop-%d, PPixPerIPix-%d, BPixPerIPPix-%d\n",
        state.mpegIPixPerGop, state.mpegPPixPerIPix,
        state.mpegBPixPerIPPix);

} // PrintMovieState


//
// In the main program, we determine if the Profile is remote, and
// open the connection if it is; otherwise, the connection stays local.
// We also allow ? or help on the command line in order to get the
// usage printout.
//
main(int argc, char *argv[])
{
    ConnectHandle ch = LOCAL_CONNECTION;
    char* thisArg = argv[1];
    char* movieName = NULL;
    char* hostname = NULL;
    char  dset[MAX_PATH];
    char  group[MAX_PATH];
    char  name[MAX_PATH];
    BOOL success;
    PdrMovieState movieState;
    PdrAttributes attributes;
    PdrMediaState mediaState;
    MovieToken movie;
    TrackToken track;
    MediaToken media;
    MediaToken nullMedia = PdrNullMediaToken();

    // Require a movie name.
    if (argc < 2) {
        Usage(argv[0]);
        return 0;
    }
    if (!strcmp(thisArg, "help") || !strcmp(thisArg, "?")
        || !strcmp(thisArg, "usage")) {
```

```
    Usage(argv[0]);
    return 0;
}
if (!strcmp(thisArg, "-h")) {
    thisArg = argv[3];
    hostname = argv[2];
    success = RemOpenConnection(ConnectEthernet, 0, hostname, &ch);
    if (!success) {
        printf("Failed to make a connection to %s\n", hostname);
        printf("Error code is 0x%x\n", GetLastError());
        return 0;
    }
    printf("Interfacing to tekrem version %d.%d\n",
        RemGetMajorVersion(), RemGetMinorVersion());
}
movieName = thisArg;

printf("Interfacing to tekpdr version %d.%d\n",
    PdrGetMajorVersion(ch), PdrGetMinorVersion(ch));
printf("Interfacing to tekvfs version %d.%d\n",
    VfsGetMajorVersion(ch), VfsGetMinorVersion(ch));
printf("Interfacing to rts version %d.%d\n",
    VfsGetEngineMajorVersion(ch), VfsGetEngineMinorVersion(ch));
printf("Compiled for PDR_DB_VERSION 0x%x\n", PDR_DB_VERSION);

printf("\nIn the following description, the special pattern\n");
printf("    of a NullMediaToken prints at '0Z 00'\n");
printf("-----------------------------------------------\n");

// Do the exposure of the movie here.
// First, see if the movie exists; if not, go no farther.
success = PdrMovieExists(ch, movieName);
if (!success) {
    printf("Movie '%s' does not exist.\n", movieName);
    KillConnectionAndExit(ch, hostname);
}
printf("Information about movie %s is as follows:\n", movieName);

// Since the movie exists, get and print the movie state.
success = PdrGetMovieState(ch, movieName, &movieState);
if (!success) {
    printf("Failed to get movie state for '%s'\n", movieName);
    printf("Error code is 0x%x\n", GetLastError());
    KillConnectionAndExit(ch, hostname);
}
PrintMovieState(movieState);

// Now the movie must be opened to go on; if can't open, quit.
movie = PdrOpenMovie(ch, movieName, 0);
if (!movie) {
    printf("Failed to open movie '%s'\n", movieName);
    printf("Error code is 0x%x\n", GetLastError());
    KillConnectionAndExit(ch, hostname);
}

// Get the attributes; indicate if it is a simple clip.
attributes = PdrGetMovieAttributes(movie);
if ((attributes == PdrError)
  || !PdrGetMovieDataset(movie, dset, sizeof(dset))
  || !PdrGetMovieGroup(movie, group, sizeof(group))
  || !PdrGetMovieName(movie, name, sizeof(name))){
    printf("Failed to get information for movie '%s'\n", movieName);
    printf("Error code is 0x%x\n", GetLastError());
```

```
        KillConnectionAndExit(ch, hostname);
    }

    printf("The movie name is fully qualified as '%s/%s/%s'\n", dset, group, name);
    printf("It is %s a simple clip\n", (attributes & PdrSimpleClip)? "" : "not");
    printf("-----------------------------------------------\n");
    // Now, get the first track.
    track = PdrGetNextTrack(movie, 0);
    while (track) {

        // For each track, display the track information, then walk the media.
        PrintTrackInfo(movie, track);
        media = PdrGetNextMediaToken(movie, track, nullMedia);
        while (!PdrMediaTokenIsNull(media)) {
            printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");

            // For each media token, get the media state and display it.
            success = PdrGetMediaState(movie, track, media, PdrThisMedia,
                           &mediaState);
            if (!success) {
                printf("Failed to get information for media %s\n", movieName);
                printf("Error code is 0x%x\n", GetLastError());
            }
            else {
                PrintMediaInfo(mediaState);

                // For each File type media, get the Vfs information.
                if (mediaState.type == PdrMediaFile) {
                    PrintFileInfo(ch, mediaState.path);
                }
            }
            media = PdrGetNextMediaToken(movie, track, media);
        }
        printf("-------------------------------------------------------------\n");
        track = PdrGetNextTrack(movie, track);
    }
    // When finished with all tracks, close the movie, and the connection.
    PdrCloseMovie(movie);

    KillConnectionAndExit(ch, hostname);

    return 0;

} // main
```

Checking free file space

Example 8, freespace.c demonstrates use of the TekVfs library, which provides low-level access to the media file system of a Profile. The program does the following:

1. It opens a connection with **RemOpenConnection**.
2. It calls **CfgGetNumFileSystems** to return the number of file systems into which the media disk space has been partitioned on the local machine. The function **CfgGetFileSystemName** returns the name of the referenced file systems.
3. It uses **VfsQueryFileSystemSpace** to obtain the amount of free space and total space on the disks of a local or remote Profile video server. The two values are displayed in megabytes and as a percentage of free space to the total space.

Example 8. freespace.c

```
//
// File: frespace.c
// A demo program to determine free disk space on a local or remote Profile.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
#include <stdio.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekvfs.h>

// Print the proper usage of this command line program.
void Usage(const char *progName)
{
    printf("Usage:\n%s [-r remote_machine]\n", progName);
    printf("    -r remote_machine (local if not specified)\n");

} // Usage

// Determine the free file space on the given machine.
// A NULL input means the local machine.
void file_space(char *remote_machine)
{
    ConnectHandle connHdl;
    int numFileSystems;
    int i, len;
    char dset[PDR_MAX_DSET_NAME_LEN];
    ULARGE_INTEGER totalSpace, spaceRemaining;
    DWORD totalMB, remainMB;
    double percentFree;

    len = PDR_MAX_DSET_NAME_LEN;
    if (remote_machine) {
        if (!RemOpenConnection(ConnectEthernet, 0, remote_machine, &connHdl)) {
            printf("Cannot connect to %s\n", remote_machine);
            return;
        }
    }
    else
        connHdl = LOCAL_CONNECTION;

    numFileSystems = CfgGetNumFileSystems(connHdl);
```

```
    printf("\nMedia file system space for %s:\n",(remote_machine) ?
                remote_machine :"Local" );

    if (numFileSystems <= 0)
        printf("\nNo File Systems found...");

    for (i = 0; i < numFileSystems; i++) {
        CfgGetFileSystemName(connHdl, i, dset, len);
        if (VfsQueryFileSystemSpace(connHdl, i, &totalSpace, &spaceRemaining)) {
          // A ULARGE_INTEGER tells the number of bytes.
             // Each kilobyte is 2^10, so a shift by 20 on the low will equal MB (KB * KB).
          // We assume that the upper 12 bits of the highPart are unused, since
          // 1 exabyte (GB * GB) is more than the entire memory of all computers
          // connected to the Internet in 1998.
            totalMB = (totalSpace.HighPart << 12) + (totalSpace.LowPart >> 20);
            remainMB = (spaceRemaining.HighPart << 12) + (spaceRemaining.LowPart >> 20);
            printf("\n%s Total = %d MB , Remaining = %d MB", dset, totalMB, remainMB);

            percentFree = 100 * ((double)remainMB / (double)totalMB);
            printf("\n\tPercent Free = %.1f%%\n", percentFree);
        }
        else {
            printf("Failed to get File System Space for filesystem %s.
               GetLastError reports 0x%x\n", dset, GetLastError());

        }
    }
    RemCloseConnection(connHdl);
    return;

} // file_space

void main(int argc, char *argv[])
{
    int i;
    char *remote_machine = NULL;

    for (i = 1; i < argc; i++) {
        if (*argv[i] == '-') {
            switch (tolower(*(argv[i]+1))) {
            case 'r':
                remote_machine = argv[++i];
                break;
            default:
                Usage(argv[0]);
                return;
            }
        }
        else {
            Usage(argv[0]);
            return;
        }
    }

    file_space(remote_machine);
}
```

Chapter 5

Using Events

VdrStateEvent is a recent addition to the Profile SDK that adds flexibility to the events mechanism. StateEvent can replace the use of DefaultEvent (**VdrDefaultEvent**) and ScheduledEvent (**VdrScheduleEvent**) in most cases. For the most part, the default and schedule events can be viewed as a subset of StateEvent. StateEvents should replace most usage of the ScheduledEvents, because they are much more efficient compared to ScheduledEvents and provide greater flexibility. (For more information on the differences between event types, see *Events* on page 38.)

The code in *Example 9, stateevt.c* steps through normal setup, such as opening a connection with **RemOpenConnection** and opening a port with **VdrOpenPortConnection**. The video standard is detected using the function **CfgGetStandard** then set with **VdrSetVideoFormat**. Then video and audio resources are allocated with several calls to the function **VdrAllocateResource**.

Finally, the code calls **VdrStateEvent** using the function as a replacement for **VdrDefaultEvent** and **VdrScheduleEvent**. **VdrStateEvent** actually provides more control over port connections during the ReadyToPlay state than its predecessors. The function prototype for **VdrStateEvent** has the same form as **VdrDefaultEvent** with the one exception that the reservedHandle field (NULL in **VdrDefaultEvent)** is the StateMask field. For the sake of compatibility, **VdrDefaultEvent** is equivalent to **VdrStateEvent** with the StateMask field set to the value EventStateAll.

With this done, a movie token is created by specifying a filename to save to with **PdrOpenMovie**. The movie is attached to the video codec with the function **VdrAttachOpenMovie**. The movie is then cued (**VdrCuePlay**), and actual play begins when the video server is instructed to shuttle (**VdrShuttle**) at the given rate (SHUTTLE_RATE). Once the specified duration has passed as determined by **VdrGetPosition**, the Profile unit is again placed into idle mode (**VdrIdle**), and the movie is detached from the timeline with the function **VdrDetachMovie**.

Next, the movie is detached with the function **VdrDetachMovie**. The entire movie is played back because no special mark-in and mark-out points are specified. With a call to **PdrCloseMovie**, the movie is closed.

Resources acquired during execution must be released. The Profile server is told to release all handles acquired during the setup phase (**VdrReleaseResource**), and the port is closed (**VdrClosePort**). This completes the process and stops the program.

Example 9. stateevt.c

```
//
// File: stateevt.c
//
// This sample program plays a JPEG clip using StateEvents.
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof, is
// protected as an unpublished work under the copyright laws of  the United States.
//
// Usage: stateevt movie_name [1]|[2]
//
#include <stdio.h>
#include <windows.h>
#include <limits.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>

#define SHUTTLE_RATE  1.0
#define NUM_INPUT     0
#define NUM_OUTPUT    0

// For demo application, we will have several resources.  Enumerate them for use
// as indexes into an array for VdrAttachMovie calls.  The first three are Codecs.
//
enum CodecResEnum { VCOD, ACOD1, ACOD2, MAX_CODEC };
enum ResEnum { VIN = MAX_CODEC, VOUT, AOUT1, AOUT2, MAX_RSRC };

// Module static variables.
static ConnectHandle    sConn;
static VdrHandle        sPort;
static ResourceHandle   sResHdls[MAX_RSRC];
static char*            spMovieName;
static BOOL             sUseStateEvents;
static BOOL             sActAsSwitcher;

//
// Print out usage line.
//
void Usage(const char* progName)
{
    printf("Usage: %s movie_name [1]|[2]\n", progName);
    printf("    1 Use StateEvents to emulate Default and Scheduled Events\n");
    printf("    2 Use StateEvents with ReadyToPlay state VIN -> VOUT\n");

} // Usage
```

```
//
// Initialize the Profile.  Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources(void)
{
    BOOL    rtn;
    BOOL    event1, event2, event3;
    VideoFormat videoFormat;
    int        i, vlimit;
    EventHandle evtHand;

    printf("Starting setup...\n");

    // Open the connection and the port.
    rtn = RemOpenConnection(ConnectLocal, 0, 0, &sConn);
    if (!rtn) {
        printf("Error opening connection.\n");
        return FALSE;
    }
    sPort = VdrOpenPortConnection(sConn);

    if (!sPort) {
        printf("Error getting port \n");
        return FALSE;
    }

    // Is this NTSC or PAL?
    switch (CfgGetStandard(sConn)) {
        case PCI_PAL_625_MODE:
            videoFormat = Format625_50Hz_2To1;
            break;
        case PCI_NTSC_525_MODE:
            videoFormat = Format525_60Hz_2To1;
            break;
        case PCI_INVALID_MODE:
        default:
            printf("Invalid or unknown video mode.\n");
            return FALSE;
    }
    VdrSetVideoFormat(sPort, videoFormat);

    //
    // Now, get the necessary resources for the demo.
    //
```

```
// Find first available codec.
    vlimit = CfgGetNumCodecs(sConn, JpegCodec // same as VideoCodec
                             );
    for (i=0; i<vlimit && !sResHdls[VCOD]; i++) {
        sResHdls[VCOD] = VdrAllocateResource(sPort, ResourceJpegCodec,
// same as ResourceVideoCodec
                                             (unsigned int)i);
    }

    if (!sResHdls[VCOD]) {
        printf("Cannot allocate jpeg video codec.\n");
        return FALSE;
    }

    // Get two audio codecs.
    sResHdls[ACOD1] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT);
    sResHdls[ACOD2] = VdrAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT+1);
    if (!sResHdls[ACOD1] || !sResHdls[ACOD2]) {
        printf("Cannot allocate audio codec.\n");
        return FALSE;
    }

    // Get video in and out resources.
    sResHdls[VOUT] = VdrAllocateResource(sPort, ResourceVideoOutput, NUM_OUTPUT);
    if (!sResHdls[VOUT]) {
        printf("Cannot allocate video output.\n");
        return FALSE;
    }

    sResHdls[VIN] = VdrGetResourceConnectionHandle(sPort, ResourceVideoInput,
            NUM_INPUT);
    if (!sResHdls[VIN]) {
        printf("Cannot get video input.\n");
        return FALSE;
    }

   // Get audio resources.
    sResHdls[AOUT1] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT);
    sResHdls[AOUT2] = VdrAllocateResource(sPort, ResourceAudioOutput, NUM_OUTPUT+1);
    if (!sResHdls[AOUT1] || !sResHdls[AOUT2]) {
        printf("Cannot allocate audio resources.\n");
        return FALSE;
    }

    if (sUseStateEvents) {
        event1 = VdrStateEvent(sPort, EventStateAll, EventConnectResources,
            sResHdls[VIN], sResHdls[VOUT]);
        event2 = VdrStateEvent(sPort, EventStateAll, EventConnectResources,
            sResHdls[VIN], sResHdls[VCOD]);
        if (!event1 || !event2) {
            printf("VdrStateEvent() failed.  \n");
            return FALSE;
        }

        if (!sActAsSwitcher) {
            // With command line option 1, this acts like the old ScheduleEvent.
            event3 = VdrStateEvent(sPort, EventStatePlayActive | EventStatePlayReady,
                // Same as EventStateAllPlay
                 EventConnectResources, sResHdls[VCOD], sResHdls[VOUT]);
        }
        else {
// With command line option 2, only connect the codec to out
// during active play.
```

```
            event3 = VdrStateEvent(sPort, EventStatePlayActive,
                EventConnectResources, sResHdls[VCOD], sResHdls[VOUT]);
        }
        if (!event3) {
            printf("VdrStateEvent() failed.\n");
            return FALSE;
        }
    }
    else {
        // Set the default event.
        event1 = VdrDefaultEvent(sPort, NULL, EventConnectResources, sResHdls[VIN],
            sResHdls[VOUT]);
        event2 = VdrDefaultEvent(sPort, NULL, EventConnectResources, sResHdls[VIN],
            sResHdls[VCOD]);
        if (!event1 || !event2) {
            printf("Cannot schedule default events.  \n");
            return FALSE;
        }

        // Schedule the event.
        evtHand = VdrScheduleEvent(sPort, INT_MIN, EventConnectResources,
            sResHdls[VCOD], sResHdls[VOUT]);
        if (!evtHand) {
            printf("Cannot schedule event.\n");
            return FALSE;
        }
    }

    return TRUE;

} // SetupResources
```

```
//
// Cleanup by releasing resources and closing the control port.
//
void Cleanup(void)
{
    int i;

    printf("Starting cleanup...\n");

    for (i=0; i<MAX_RSRC; ++i) {
        if (sResHdls[i]) {
            VdrReleaseResource(sResHdls[i]);
            sResHdls[i] = 0;
        }
    }

    if (!VdrClosePort(sPort)) {
        printf("Cannot close port.  \n");
        return;
    }
    sPort = 0;

} // Cleanup

//
// Play the movie clip.
//
void StartPlay(void)
{
    INT oldpos, newpos;
    MovieToken movieTok;
    MovieHandle movieHdl;

    // Open and attach the movie that we just recorded.
    movieTok = PdrOpenMovie(sConn, spMovieName, 0);
    if (!movieTok) {
        printf("Movie %s does not exist \n", spMovieName);
        return;
    }

    movieHdl = VdrAttachOpenMovie(movieTok, MAX_CODEC, sResHdls, NULL, ShiftAfter,
            MarkLongest);
    if (!movieHdl) {
        printf("Error getting movie handle \n");
        return;
    }

    // Cue up playback of media attached with VdrAttachMovie.
    if (!VdrCuePlay(sPort, SHUTTLE_RATE)) {
        printf("Cannot cue play \n");
        return;
    }

    // Wait 15 seconds, so user can see what happens in cue play state.
    printf("Waiting 15 seconds to play.\n");
    Sleep(15000);

    // Begin motion playback.
    if (!VdrShuttle(sPort, SHUTTLE_RATE)) {
        printf("Cannot begin playback \n");
        return;
    }
```

```
    printf("Playing movie.\n");

    // Wait while movie plays.
    // When newpos and oldpos are the same, we're done playing out.
    newpos = 0;
    do  {
        oldpos = newpos;
        Sleep(100);                         // wait 1/10th second
        newpos = VdrGetPosition(sPort);
    }  while (newpos > oldpos);

    // Pause for 10 seconds, so you can watch the video screen.
    // In regular mode, you'll see the final frame frozen.
    // In "switcher" mode, you'll see video in on the output.
    printf("Waiting 10 seconds in idle mode.\n");
    Sleep(10000);

    // Cease play back.
    if (!VdrIdle(sPort)) {
        printf("Cannot move to idle.\n");
        return;
    }

    // Detach the movie handle from the channel.
    if (!VdrDetachMovie(movieHdl, ShiftAfter)) {
        printf("Cannot detach movie.\n");
        return;
    }

    if (!PdrCloseMovie(movieTok)) {
        printf("Cannot close movie.\n");
        return;
    }

} // StartPlay
```

```
//
// The main entry point.
//
void main(int argc, char *argv[])
{

    BOOL rtn;

    // Read in the new movie name.
    if (argv[1]) {
        spMovieName = argv[1];
    }
    else {
        Usage(argv[0]);
        exit(1);
    }

    // Optional argument shows state events usage.
    if (argv[2]) {
        // Make sure there is no third argument.
        if (argv[3]) {
            Usage(argv[0]);
            exit(1);
        }
        if (!strcmp(argv[2], "1")) {
            sUseStateEvents = TRUE;
        }
        else if (!strcmp(argv[2], "2")) {
            sUseStateEvents = TRUE;
            sActAsSwitcher = TRUE;
        }
        else {
            Usage(argv[0]);
            exit(1);
        }
    }

    rtn = SetupResources();

    if (rtn)
        StartPlay();

    Cleanup();

} // main
```

Chapter 6

Transferring Media with Fibre Channel

Fibre Channel enables you to copy and transfer media between Profile systems faster than in real-time. Not only can you copy media between Profiles, you can also transfer media to and from SGI servers, as long as appropriate software is installed on the SGI server (refer to the *Profile Fibre Channel Server Interface Manual* for complete information). Fibre Channel transfers must be used with some care since they can consume a substantial amount of the bandwidth of the media disks and could thus restrict the bandwidth available for playing and recording video.

Streaming over Fibre Channel makes it possible to transfer parallel tracks of media, such as concurrent audio and video tracks, in small packets, and then reassemble them on the destination Profile unit.

This allows you to transfer a clip while it is still being recorded or playback a clip before the transfer is complete. For example, soon after it begins receiving a clip, a destination Profile unit can begin playing it. The streaming transfer continues while playback occurs at the destination, delivering new packets at a fast enough rate to allow playback to proceed uninterrupted.

Transfer of complex movies over Fibre Channel is now supported. A *complex movie* is a movie that consists of pointers to several other clips. Such a movie is not rendered and is not a physical clip taking up disk space. Fibre Channel support for complex movies requires that the source Profile send the appropriate pieces of the complex movie's constituent clips along with the data required to reassemble the pieces on the destination Profiles.

With Media Manager under Profile system software 2.5 or higher, all clips (simple movies) and masters (complex movies) are streamed with Media Manager. SGI servers also can only stream clips. Remote recording, playback, and editing of media is not available on an SGI server.

> ***NOTE: You must have Profile system software version 2.2 to copy media via Fibre Channel. Version 2.4 allows you to stream simple clips and interoperate with SGI servers. Versions 2.5 and higher allow you to stream complex clips and archive across Fibre Channel by specifying movie names prefixed with a Profile name.***

To transfer media across Fibre Channel, a Profile system must have an Ethernet local area network (LAN) board and a Fibre Channel board installed, each one connected to its respective network. The Ethernet LAN carries commands between Profile units, while the Fibre Channel connection carries the actual video, audio, and timecode data. Both networks use TCP/IP (Transmission Control Protocol/Internet Protocol).

For instructions on setting up these two networks, see the installation manual that came with your Profile system; if you have installed a Fibre Channel board as an f-kit, see the *Profile Family Local Area Network Installation Manual*. You should also consult the chapter on Fibre Channel video networking in the *Profile Family User Manual*.

Configuring Fibre Channel

To transfer media between Profiles over Fibre Channel, you must first configure two discrete networks, Ethernet and Fibre Channel. Network configuration is covered in more detail in the *Profile Family User Manual*, but here is a brief overview of what you need to do to configure your Profiles for Fibre Channel transfers (assuming your Profile system has the appropriate hardware installed).

1. Set up your Ethernet LAN.

   Using a twisted-pair cable, connect your Profile's LAN board to a standard 10Base-T or 100Base-T Ethernet hub or switch. (You cannot connect Profiles directly from one LAN board to another without a special cable.) Set your Profile system's Ethernet IP address in Windows NT with the **Control Panel | Network** applet. (See your system administrator for the correct IP addresses to use on your network.)

2. Set up your Fibre Channel network.

   Each Profile requires a separate and distinct Fibre Channel IP address in addition to its Ethernet IP address. You can set your Fibre Channel IP address either through the Fibre Channel dialog box in the **Configuration Manager** or with the **fcconfig** command line utility. To use **Configuration Manager**, see the chapter describing the configuration application in the *Profile Family User Manual*. To use **fcconfig**, see the chapter on Fibre Channel video networking in the *Profile Family User Manual.*

3. Edit the *hosts* file.

   This file can usually be found in the *c:\winnt\system32\drivers\etc* directory. *Example 10* shows a sample hosts file with bogus IP addresses. (See your system administrator for the correct IP addresses to use on your network.)

Example 10. Sample hosts file

```
Computer Name:          PROFILE1
Ethernet Address:       123.123.99.1
FC Address:             123.123.100.1
Computer Name:          PROFILE2
Ethernet Address:       123.123.99.2
FC Address:             123.123.100.2
```

The *hosts* file lists the Profile unit names (**PROFILE1**, **PROFILE2**, etc.) and their respective Fibre Channel names (**PROFILE1_fc0**, **PROFILE2_fc0**, etc.) and which IP addresses are associated with them. If you make one comprehensive *hosts* file with the IP addresses of all the Profile units on your network, you can copy that file from one Profile to all others and save yourself a lot of time editing each *hosts* file individually. You can also use a DNS server for name lookup, but that discussion is outside the scope of this manual. Another alternative is to use the PDR network configuration service (**fcncs**) to automatically update and maintain the *hosts* file.

4. Set up the *pdrstart.bat* file for version 2.5 or earlier.

   Both the **htssvc** service and **Port Server** program must be running at all times on all Profiles where you are using Fibre Channel. The *pdrstart.bat* file (in the *c:\profile* directory) starts **htssvc** and **Port Server** in the proper sequence. If you want, you can place *pdrstart.bat* in a startup folder and set it to run minimized. Make sure it's running on the other Profiles, too. After configuring all machines, you can verify name resolution with **fcping**.

Multicast programming

Multicasting is the simultaneous transfer of a single streamed file to several destinations. Multicasting can streamline applications in environments where one unit with a large amount of storage can multicast files to a network of several Profiles (up to a maximum of eight) for playout. Multicast transfers, like unicast transfers, occur through the TekXfr library. The only real difference is that, instead of a single host/file destination pair for a unicast call, a multicast call sends a list of destination pairs.

Limitations include the following:

- Profile multicasting (one-to-many transfers) requires the dedicated bandwidth available in a Fibre Channel *switch* environment; multicasting is not available in a shared bandwidth *hub* environment. To enable multicasting, the Fibre Channel network must be configured with a Fibre Channel switch. Multicast is not supported on Profile XP Media Platforms.
- Fibre Channel hubs and Fibre Channel switches cannot coexist on the same network. If you are converting from a hub network to a switched network, you must convert all nodes on the network.
- Converting from a Fibre Channel hub network to a switched network may require a firmware upgrade to each Profile's Fibre Channel board.
- Multicasting requires a software option that must be running on each Profile.
- The current limit is eight destinations in a single transfer. For more than eight destinations, the transfer command must be called multiple times.
- Each destination host in a multicast must be unique.
- The source host in a multicast cannot also be a destination host.

NOTE: When multicasting, the transfer rate of the sending Profile is cut approximately in half. Thus, multicasting makes the most "performance" sense during transfers of a 1:3 (or higher) ratio. (A 1:2 multicast yields the same performance as making two transfers.)

Switched Fibre Channel networks

There are many advantages to using a switched Fibre Channel network, the biggest being the guaranteed bandwidth (as opposed to a hub environment's shared bandwidth) between devices. A switched environment also offers increased system reliability through its complete isolation between Profiles on the network. And with browser-based management tools, it is easy to view complete statistics and information on each node and detect network problems, simplifying Fibre Channel troubleshooting.

The greatest drawback to a switched Fibre Channel network is the price; switches are more expensive than hubs.

If your expenses allow it, you should configure your Fibre Channel network with a switch to provide faster transfers, more robustness, and more advanced configuration/monitoring tools. If, however, multicasting is not a requirement, and you are more interested in reducing costs, you may wish to stick with a hub environment.

If you do choose to go with a switched network, you should consider connecting switches redundantly by tying two ports together between two switches. If you have three switches, connect each switch to the other two. (It is possible to cascade up to 32 switches together.) What this does is allow your switches to automatically default to the redundant path in case of

a failure. Switch ports are hot-swappable too, so that if a port in the switch fails, you can reconnect the Profile to a different port, and the system will resync in a few seconds without powering down the Profile or the switch.

The drawback to using redundant paths is the reduction in the total number of connections allowed. With redundant connections, each switch will lose two ports for interconnects, thus a 16-port switch could have only fourteen Profiles connected and an 8-port switch could have only six Profiles connected.

Multicasting errors

In a multicast, a calling application can discern between three results: complete success, partial success, and complete failure. The transfer will continue until it is successful or an error occurs. In the event that there is not *complete success*, it is up to the calling application to determine which hosts did not get the movie.

- *Complete success* means that all the destination hosts successfully received the transferred file.
- *Partial success* means that at least one destination host *did* successfully receive the transferred file and at least one destination host did *not*. In a partial success situation, a special error code is returned. Those hosts which have received the movie successfully retain the file.
- *Complete failure* means that none the destination hosts successfully received the transferred file.

Possible errors returned are shown in *Table 5, Streaming error codes*. Most of these are the same as unicast errors. Other modules involved in the transfer can return both multicast and unicast error codes.

Table 5. Streaming error codes

Error code	Error name
0x02030001 *	STRM_E_MALFORMED_UML
0x02030002 *	STRM_E_BAD_UML_COUNT
0x02030003	STRM_E_NO_RESOURCE
0x02030004 *	STRM_E_INTERNAL
0x02030005 *	STRM_E_TIMEOUT
0x02030006	STRM_E_INVALID_STREAMID
0x02030007	STRM_IUE_INVALID_SESSION
0x02030008	STRM_E_NOT_SUPPORTED
0x02030009	STRM_E_BAD_ARGUMENTS
0x0203000a	STRM_E_UNKNOWN_CODE
0x0203000b	STRM_E_INCOMPLETE **
0x0203000c	STRM_E_MALFORMED_LINE
0x0203000d *	STRM_E_REMOTE_FAILURE
0x0203000e	STRM_E_NONETWORK
0x0203000f *	STRM_E_NO_CONNECTION
0x02030010 *	STRM_E_OUT_OF_RESOURCES
0x02030011	STRM_E_NO_SPACE
0x02030012	STRM_E_IO
0x02030013	STRM_E_BADAUTH
0x02030014	STRM_E_SERVER
0x02030015	STRM_E_WRITE_PROTECT
0x02030016	STRM_E_VOLUME_OVERFLOW

* *These errors are specific to multicast mode.*

** *This is the error reported for partial success--when at least one but not all of the destinations get the data successfully.*

The PDR network configuration service

The PDR network configuration service (**fcncs**) collects information about other Profiles on the network via multicast, and maintains a local table of the information. It also will update the *hosts* file if and only if the `fcconfig -hta` option is set to *on*.

This service has the following command line options:

`fcncs -install`	Installs the service on the system. The service starts automatically when installed.
`fcncs -remove`	Removes the service from the system.
`fcncs -start`	Starts the service.
`fcncs -stop`	Stops the service.

UML descriptions

A UML (Uniform Media Locator) provides a complete description of the source or destination of a transfer over Fibre Channel. In the Fibre Channel world, UMLs are similar in concept to the URL (Universal Resource Locator) scheme used with HTTP on the Internet.

Here is the generic form for a UML:

<host>/*<type>*/*<typeSpecificInformation>*[?*<options>*]

..or, to use a specific example:

`Profile1/explodedFile/INT1:/default/mymovie?(3600-21600)`

The *<host>* portion consists of the Windows NT computer name of the source or destination computer, in the above example, *Profile1*. The host must be available on the Fibre Channel network with the other associated machines; if the host is not specified, it will default to LOCALHOST.

The *<type>* portion represents the type of transfer to be performed at the source or destination. Presently, *ExplodedFile* is the only type defined for the external API.

The *<typeSpecificInformation>* portion is described by the dataset (*INT1:)*, the group (*default)*, and the name of the clip (*mymovie*).

The bracketed ?*<options>* piece indicates a group of optional arguments that will allow you to specify which fields will be copied from the clip and also to designate streaming priority. If you choose to append any options to the UML, add the question mark and the rest of the syntax described below for each option.

Currently, there are three options defined: *flattened*, *exact*, and *HOT.*

The *flattened* option

This option, the default mode of operation, requires a (*<startField>*-*<endField>*) value range after the question mark character. In the example below...

`Profile1/explodedFile/INT1:/default/mymovie?(3600-21600)`

...the range `(3600-21600)` specifies a five-minute portion of an NTSC clip starting after the first minute[1].

In general, only the media found between the parenthetical movie marks is transferred; however, material outside the marks *may* be transferred for MPEG files, because of the need to tansfer I-frames on which some of the predictive frames are based. For example, material

1. A five-minute clip is defined as 5 x 60 seconds x 60 fields = **18,000** frames;
the starting point (one minute into the clip) is defined as 60 seconds x 60 fields = **3600** frames);
an 18000-frame (five-minute) clip, starting one minute into the clip, yields the range **3600-21600**.

back to and including the previous I-frame will be transferred for closed-GOP streams, and material back to and including the I-frame before the previous I-frame will be transferred for open-GOP streams. Also, if the *<endField>* frame (which is exclusive) comes just after a P-frame, one additional frame will be transferred.

The *exact* option

This option transfers an exact copy of all the different pieces of media in their original format. This includes all material located between the movie's first field and last field, plus all the other data in the media files that is not included in the movie description, such as material before the first field or after the last field. (This allows an archive of an edit session on a complex movie).

Using the exact option...

```
Profile1/explodedFile/INT1:/default/mycomplexmovie?exact
```

...will transfer all media files touched by the complex movie *mycomplexmovie*, even if the complex movie uses only 10% of any one media file. In comparison, using the same UML without the exact option...

```
Profile1/explodedFile/INT1:/default/mycomplexmovie
```

...will transfer only the media that appears between the movie marks (or just the 10%).

> ***NOTE: You cannot specify startField and endField values with the exact option. Any such request will fail.***

The *HOT* option

This option allows you to designate a single stream as a "HOT" transfer that supercedes all other transfers. This option suspends all other Fibre Channel transfers and reserves all available Fibre Channel bandwidth for that one stream alone. After the HOT transfer has finished, all suspended transfers will resume.

> ***NOTE: The available bandwidth in a loop topology might not be enough to guarantee that your HOT stream's performance will not be degraded, but in a switched Fibre Channel environment, the HOT option will allot full Profile-to-Profile bandwidth for your stream, from source to sink.***

The example below shows the syntax for the HOT option:

```
Profile1/explodedFile/INT1:/default/mymovie?HOT
```

Note that the word "HOT" must appear in capital letters after the question mark. Here are some other requirements for HOT stream usage:

- Only one HOT stream is allowed per Profile; a second request for a HOT stream will fail.
- For an **XfrRequest** call, the option HOT must be specified on the source UML and on the destination UML. Failure to do so will yield undefined results.
- For an FTP (STOR or RETR) call, the HOT option is specified on the UML.

It is possible to use the HOT option in conjunction with both the flattened and exact options, but each option must be separated by its own question mark:

```
Profile1/explodedFile/INT1:/default/mymovie?(3600-21600)?HOT
Profile1/explodedFile/INT1:/default/mymovie?exact?HOT
```

Using UMLs

Example 11 demonstrates how to use an FTP file transfer to get all media files from a movie `mymovie` consisting of one video track, two audio tracks, and a timecode track. The files are saved locally as media files, then they are sent back as media files `yourmovie.XX` to the Profile system called *profile_fc0*.

Example 11. UML usage in file mode

```
c: ftp profile_fc0
User : anything (except movie) or RETURN
ftp > bin
ftp > get INT1:/PDR/default/default/mymovie.V01 mymovie.V01
ftp > get INT1:/PDR/default/default/mymovie.A01 mymovie.A01
ftp > get INT1:/PDR/default/default/mymovie.A02 mymovie.Ao2
ftp > get INT1:/PDR/default/default/mymovie.T01 mymovie.T01

ftp > put mymovie.V01 INT1:/pdr/default/default/yourmovie.V01
ftp > put mymovie.A01 INT1:/pdr/default/default/yourmovie.A01
ftp > put mymovie.A02 INT1:/pdr/default/default/yourmovie.A02
ftp > put mymovie.T01 INT1:/pdr/default/default/yourmovie.T01

ftp > quit
```

Out of band control via the TekPdr API must be exercised to open the movie and query the media file names on each track, before issuing the FTP request. When the media files are "put" onto a profile, a movie is not created, only media files bearing the proposed movie name. Again, out of band control via the SDK must be exercised to construct movie `yourmovie` from the media files yourmovie.

Example 12 describes how to get a movie `mymovie` from a Profile system called profile_fc0 via FTP, save it locally as `demo`, then send the stream back as movie `yourmovie`. The local file name `demo` must be explicitly specified, as the `/explodedFile/INT1:/default` directory does not exist on the local system (the FTP client). Be sure to set the transfer type to image (bin).

Example 12. UML usage in streaming

```
c: ftp profile_fc0
User : movie
ftp > bin
ftp > get /explodedFile/INT1:/default/mymovie demo
ftp > put demo /explodedFile/INT1:/default/yourmovie
ftp > quit

REQUIRED:
set the transfer type to image
ftp > bin
```

You can now play the movie `yourmovie` directly, without any other operations.

Copying media via Fibre Channel

Profile offers you several ways to write your own Fibre Channel media transfer applications:

- with TekXfr function calls (the preferred method);
- with TekPdr function calls;
- with TekVfs function calls;
- with the Media Manager application; and
- with the **copymovie** command line utility.

NOTE: A transfer may be initiated from a remote, non-Profile PC running Windows NT and connected via Ethernet, but the actual transfers must take place between units connected via Fibre Channel.

Our recommended transfer method is through the recently added TekXfr library

Copying media with TekPdr functions

Fibre Channel copies are possible through several function calls in the TekPdr library of the Profile SDK:

- The **PdrCopyMovie** function initiates a copy, specifying the media source and destination. This call also initiates a WaitToken which helps to manage the copy queue.
- The **PdrGetWaitOpStatus** ***must*** be called in conjunction with **PdrCopyMovie**. This call polls the copy status until the copy is complete or it reports a failure. The application must poll with this function to ensure that the copy operation proceeds smoothly.
- The **PdrTerminateWaitOperation** concludes the copy operation. You ***must*** call this function to terminate all copy operations, whether successful or not. The WaitToken closes when **PdrTerminateWaitOperation** is called; therefore, subsequent calls to **PdrCopyMovie** will not return a WaitToken until the previous WaitToken is terminated. This function can also cancel a current copy operation.

For parameter and other information for these function calls, see the *Profile SDK Reference Manual*.

A master (complex movie) definition may include many individual media files. To optimize network performance, the **PdrCopyMovie** function queues all media files to be copied, but will only initiate a fixed number of copies at a time. For example, a master may consist of five media files (for example, one media file for video and four audio files), but **PdrCopyMovie** might elect to perform just three concurrent media copies.

As these copies complete, the remaining two media files will not be copied automatically. They will only be copied when **PdrGetWaitOpStatus** is called. Once a copy operation has been started, subsequent media copies are initiated only when **PdrGetWaitOpStatus** is called. Thus, the first three media copies might all complete successfully, but the next two will not be initiated until the application next issues a **PdrGetWaitOpStatus**. If **PdrTerminateWaitOperation** were called before **PdrGetWaitOpStatus**, the two remaining files would never be copied.

A media copy can be initiated as one of the following:

- a push operation (where the source machine initiates the copy);

- a pull operation (where the destination machine initiates the copy); or
- a remote operation (where a third machine initiates copy).

Nonetheless, all copies are translated into pulls. All copies are pulls at both the movie database level and at the video file system level. Although this is all handled transparently by the application, a brief understanding of the underlying mechanism may aid in application design.

Any **PdrCopyMovie** request, regardless of where it is initiated, is immediately handed to the destination machine. The destination machine then queries the source machine's movie database for the movie definition. This includes all the data associated with the movie, such as the list of all media files that are part of the movie. Once the destination machine has compiled this list of media files, it commences queueing media file copies. These individual media file copies are all handled as pulls by the video file system.

As mentioned earlier, there is a restriction of one WaitToken per Profile system when using TekPdr functions. Since all movie copies are movie database pulls, this restriction translates to one WaitToken per destination Profile. This also means that a source Profile (server) mays safely be involved in multiple copies.

Copying media files with TekVfs functions

The movie copying functions in the TekPdr library--namely **PdrCopyMovie**, **PdrGetWaitOpStatus**, and **PdrTerminateWaitOperation**--meet the needs of many application developers. If you need finer control over the copy process, however, you may want to use the lower-level TekVfs library commands. These commands do not copy *movies*; they copy the underlying media files. An application employing TekVfs calls must assume the burden of maintaining integrity with the movie database. TekVfs calls are recommended only for developers familiar with their implications.

Like TekPdr function calls, the TekVfs copies are handled by a set of three functions:

- The **VfsCopyFile** function initiates a media file copy, specifying the media source and destination, and Boolean to wait for completion of the copy.
- The **VfsStatusOfCopy** polls the system for the status of the copy operation.
- The **VfsCancelCopy** function cancels a media file copy.

For parameter and other information for these function calls, see the *Profile SDK Reference* manual. The pattern of use for these three functions is analogous to their equivalents in the TekPdr library. However, note the following differences:

- There is a limit of eight simultaneous TekVfs copies per destination machine.
- Synchronous operation is optional because **VfsCopyFile** has a Boolean WaitForCompletion parameter that monitors the WaitToken. More than one WaitToken is possible, so your application does not have to manage the queue.
- Since TekVfs does not implement queue management, the application is not required to use **VfsStatusOfCopy** if **VfsCopyFile** is given a nonzero *waitForCompletion* flag.
- The source and destination file names take the following form:

  ```
  profile8_fc0.INT:/PDR/default/#1-001.V0
  ```

  This can be done using code like:

  ```
  sprintf(szFinalName, "%s_fc0.%s", node, pathName);
  ```

In review, here is a clarification of the limitations of the current system:

- When copying via TekPdr functions, there can be only one WaitToken per Profile system, which means your application must manage copy task queuing.
- You can make no more than three concurrent copies from the same Profile server without performance penalties.
- Nonmedia files cannot be copied across Fibre Channel except by **VfsCopyFile**.

Copying media with Media Manager

A simple way to test your work so far is to start Media Manager by double-clicking its icon on the Windows NT desktop. Use **File | Add/Remove Machine** in Media Manager to add or remove a Profile machine on the network. For more information on copying media and connecting to other Profile systems, see the chapter on Media Manager in the *Profile Family User Manual.*

Copying media with copymovie

The **copymovie** command line utility also copies media between Profiles disk recorders using the Fibre Channel. In this example, typed at a command line prompt, you copy media from **INT:/default/movie1** on **PROFILE1** to the local Profile where the command was run, where it is named **INT:/default/movie7**.

```
copymovie PROFILE1 INT:/default/movie1 * INT:/default/movie7
```

A companion to the **copymovie** command is the **listnames** command line utility. It provides a way to list media on a remote Profile without using Media Manager. In the example here, you list all movies in **INT:/default** on **PROFILE1**:

```
listnames -r PROFILE1 -l INT:/default/
```

Using FTP for streaming transfers

Another way to transfer media over Fibre Channel is through FTP.

FTP (File Transfer Protocol) is a TCP/IP protocol that has been used for many years to log on to a network, list directories, copy files, and make other transfer operations. The FTP daemon that runs on the Profile real-time processor has been enhanced to allow, not only traditional FTP file transfers, but streaming file transfers as well.

The streaming protocol, when called by FTP, is passed a UML (see *UML descriptions* on page 112) that will start an FTP buffer source (or sink, depending on the direction of transfer). The external world sees an RFC-959-compliant data connection since the internal streaming protocol processing is not visible to the outside world.

FTP commands can run in *file mode* or in *movie mode* (see *Table 6*). File mode is traditional FTP, used for straightforward file transfers; movie mode uses FTP to initiate a movie stream transfer.

Table 6. Supported commands processed by the FTP daemon

FTP command	File mode	Movie mode
USER	*no operation*	movie
PASS	*no operation*	*no operation*
PORT	yes	yes
TYPE	yes	yes
SEND	yes	yes

Table 6. Supported commands processed by the FTP daemon

FTP command	File mode	Movie mode
RECV	yes	yes
QUIT	yes	yes
PASV	yes	yes
ABOR	yes	yes
SITE	yes	yes
CWD	yes	*n/a*
NLST	yes	*n/a*

File mode

When in file mode, FTP is running directly on the native Profile file system (media file system or MFS) in the real-time processor. The MFS stores media files which are actually individual movie tracks such as the video track, the audio tracks and the timecode track. In this mode, media files can be listed, directories can be changed, etc.. Since no concept of a "movie" exists in the media file system, all transfers using this mode are done on a track-by-track (file-by-file) basis.

If an FTP client wishes to transfer a movie rather than media files, the client may dictate that the session be in movie mode. When in movie mode, all `SEND` and `RECV` commands are processed by the streaming protocol on the targeted Profile. If movie mode is not selected, all `SEND` and `RECV` commands are processed by the FTP session directly on the media files specified by the request.

Movie mode

Movie mode is selected by initiating an FTP session with the user designated as MOVIE. Movie mode provides a streamed file (multiplexed audio, video, and timecode) rather than a track-by-track transfer. The notion of a "movie" only exists in the Windows NT PDR database. Since the FTP connection is actually on the real-time side rather than on the Windows NT side, it is not possible to query the Windows NT database, change directories, etc. when operating in movie mode. Queries must be handled out of band over Ethernet through the Profile remote API.

Sample code: Media copies

Example 13, deepcopy.c demonstrates use of the **PdrCopyMovie** function call to copy a movie file. A single movie can be copied between any two Profiles on the Profile video network. You can specify if the copy is to be *exact*, *rendered*, *extract* or *shared*. An exact copy is the default option if none is specified.

Once the command-line is parsed, **PdrCopyMovie** is called with the appropriate parameters and a WaitToken is received. **PdrGetWaitOpStatus** then queries the state of the copy as it happens. Finally, the function **PdrTerminateWaitOperation** closes the copy operation. Some timing and bandwidth information is displayed throughout the process, to allow you to gauge the performance of the system.

Example 14, xferumls.c on page 125 provides a similar example using Fibre Channel to perform streaming operations.

Example 13. deepcopy.c

```
//
// File: deepcopy.c
// Sample code to demonstrate PdrCopyMovie().
//
// Copyright (c) Grass Valley Group, Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
#include <windows.h>
#include <stdio.h>
#include "tekpdr.h"
#include "tekrem.h"

//
// Display usage line.
//
void Usage(const char *progName)
{
  printf("Usage:\n");
  printf("%s srcMachine srcName destMachine destName [copy type]\n",progName);
  printf("      if Machine is *, uses LOCAL_CONNECTION\n");
  printf("      copy type is exact(default), rendered, extract, or shared\n");
  printf("%s ?     : causes this message to be printed\n", progName);
  printf("%s help  : causes this message to be printed\n", progName);
  printf("%s usage : causes this message to be printed\n", progName);

} // Usage

//
// Take care of closing any remote connections.
//
void CloseConnections(ConnectHandle srcConn, ConnectHandle dstConn)
{
    if (srcConn != LOCAL_CONNECTION) {
        RemCloseConnection(srcConn);
    }
    if (dstConn != LOCAL_CONNECTION) {
        RemCloseConnection(dstConn);
    }

} // CloseConnections

//
// Parse the command-line and perform the copy.  Determine if the Profile is remote,
// and open the connection if it is;  Otherwise, the connection stays local.
// Also allow ? or help on the command line in order to get the usage printout.
//
main(int argc, char *argv[])
{
    ConnectHandle srcConn = LOCAL_CONNECTION;
    ConnectHandle dstConn = LOCAL_CONNECTION;
    char* srcHost = NULL;
    char* srcName;
    char* dstHost = NULL;
    char* dstName;

    const char* thisArg;
    DWORD tickStart, tickWait, tickFinish;
    PdrCopyType copyType;

    BOOL            success;
```

```
WaitToken       wait;
PdrWaitOpStatus status;
double          bandwidth, average;

if (argc < 5) {
    Usage(argv[0]);
    return 0;
}
if (argc < 6) {
    // No copy type supplied, setting default.
    copyType = PdrExactMedia;
}
else {
    switch (argv[5][0]) {
    case 's':
    case 'S':
        copyType = PdrShareMedia;
        break;
    case 'r':
    case 'R':
        copyType = PdrRenderedMedia;
        break;
    case 'e':
    case 'E':
        // Is the argument exact or extract?
        switch (argv[5][2]) {
        case 'a':
            copyType = PdrExactMedia;
            break;
        case 't':
            copyType = PdrExtractMedia;
            break;
        default:
            Usage(argv[0]);
            return 0;
        }
        break;
    default:
        Usage(argv[0]);
        return 0;
    }
}

if (!strcmp(argv[1], "help") || !strcmp(argv[1], "?") || !strcmp(argv[1], "usage")) {
    Usage(argv[0]);
    return 0;
}
if (strcmp(argv[1], "*")) {
    srcHost = argv[1];
}
srcName = argv[2];
if (strcmp(argv[3], "*")) {
    dstHost = argv[3];
}
dstName = argv[4];
if (srcHost || dstHost) {
    printf("Interfacing to tekrem version %d.%d\n",
        RemGetMajorVersion(), RemGetMinorVersion());
}
if (srcHost) {
    success = RemOpenConnection(ConnectEthernet, 0, srcHost, &srcConn);
    if (!success) {
        printf("Failed to make a connection to %s\n", srcHost);
```

```
        printf("Error code is 0x%x\n", GetLastError());
        return 0;
    }
    printf("srcHost %s connection is 0x%x\n", srcHost, srcConn);
}
if (dstHost) {
    success = RemOpenConnection(ConnectEthernet, 0, dstHost, &dstConn);
    if (!success) {
        printf("Failed to make a connection to %s\n", dstHost);
        printf("Error code is 0x%x\n", GetLastError());
        CloseConnections(srcConn, dstConn);
        return 0;
    }
    printf("dstHost %s connection is 0x%x\n", dstHost, dstConn);
}

if (!PdrMovieExists(srcConn, srcName)) {
    printf("src movie %s does NOT exist!\n", srcName);
    printf("Request failed.\n");
    CloseConnections(srcConn, dstConn);
    return 0;
}
printf("%06d: Input validated\n", GetTickCount()%1000000);
printf("       srcConn  0x%x, srcMachine  %s, srcName  %s\n",
      srcConn, srcHost, srcName);
printf("       destConn 0x%x, destMachine %s, destName %s\n",
      dstConn, dstHost, dstName);

// Everything is ready, grab the current time then start the transfer.
tickStart = GetTickCount();
wait = PdrCopyMovie(srcConn, srcName, dstConn, dstName, copyType);
tickWait = GetTickCount();
if (!wait) {
    // Failed to get wait token.
    printf("Error in PdrCopyMovie call; WaitToken is 0\n");
    printf("      LastError code is 0x%x\n", GetLastError());
    printf("Request failed.\n");
    CloseConnections(srcConn, dstConn);
    return 0;
}
printf("        WaitToken value is 0x%x\n", wait);
printf("        Ticks for startup of operation are %d\n", tickWait - tickStart);
printf("%06d   Time started\n%06d   Start up completed\n",
      tickStart%1000000, tickWait%1000000);

status.state = PdrWaitOpContinue;
success = TRUE;
while (success && status.state == PdrWaitOpContinue) {
    success = PdrGetWaitOpStatus(wait, &status);
    if (!success) {
        printf("PdrGetWaitOpStatus call failed\n");
        printf("    reason is 0x%x\n", GetLastError());
    }
```

```
        else {
            switch (status.state)
            {
                case PdrWaitOpError:
                    thisArg = "PdrWaitOpError";
                    break;
                case PdrWaitOpRequest:
                    thisArg = "PdrWaitOpRequest";
                    break;
                case PdrWaitOpCanceled:
                    thisArg = "PdrWaitOpCanceled";
                    break;
                case PdrWaitOpContinue:
                    thisArg = "PdrWaitOpContinue";
                    break;
                case PdrWaitOpInvalidHandle:
                    thisArg = "PdrWaitOpInvalidHandle";
                    break;
                case PdrWaitOpCompleted:
                    thisArg = "PdrWaitOpCompleted";
                    break;
                default:
                    thisArg = "??  ??";
                    break;
            }
            bandwidth = (double)status.lastBandWidth / 1000000.;
            average = (double)status.averageBandWidth / 1000000.;
            printf("%06d  pcnt %3d, #W %3d, #A %1d, BW %6.3f, aver %6.3f, %s\n",
                GetTickCount()%1000000,
                status.percentCompleted,
                status.numOfFileWaiting, status.numOfFileCopying,
                bandwidth, average, thisArg);
            Sleep(990);
        }
    }
    tickFinish = GetTickCount();

    PdrTerminateWaitOperation(wait, NULL);
    printf("        Total time: %d\n", tickFinish - tickStart);

    CloseConnections(srcConn, dstConn);

    return 0;

} // main
```

Streaming with Fibre Channel

Fibre Channel streaming is made possible because of the TekXfr library in the Profile SDK. The TekXfr library currently contains four function calls:

- XfrAbort
- XfrGetActiveTokens
- XfrGetStatus
- XfrRequest

For detailed information on these functions, see the *Profile SDK Reference Manual*.

XfrAbort

XfrAbort requests that a transfer in progress be stopped, or releases the XfrToken in a streaming transfer that is already done.

XfrGetActiveTokens

XfrGetActiveTokens returns an array of the transfers currently occurring on the system it is requested on. This allows applications to provide in-transit status. Given this list, the application can then call **XfrGetStatus** for all the transfers currently occurring or in the status cache.

XfrGetStatus

XfrGetStatus returns the current status of a transfer in progress or a transfer that was completed earlier. The final status of any request is cached so that an application can request it at a later time. This helps you keep track of transfers as a request may be queued and finalized later automatically.

XfrRequest

The **XfrRequest** function initiates a transfer between a single data source and a destination. This call returns immediately after initiation (NULL if unsuccessful).

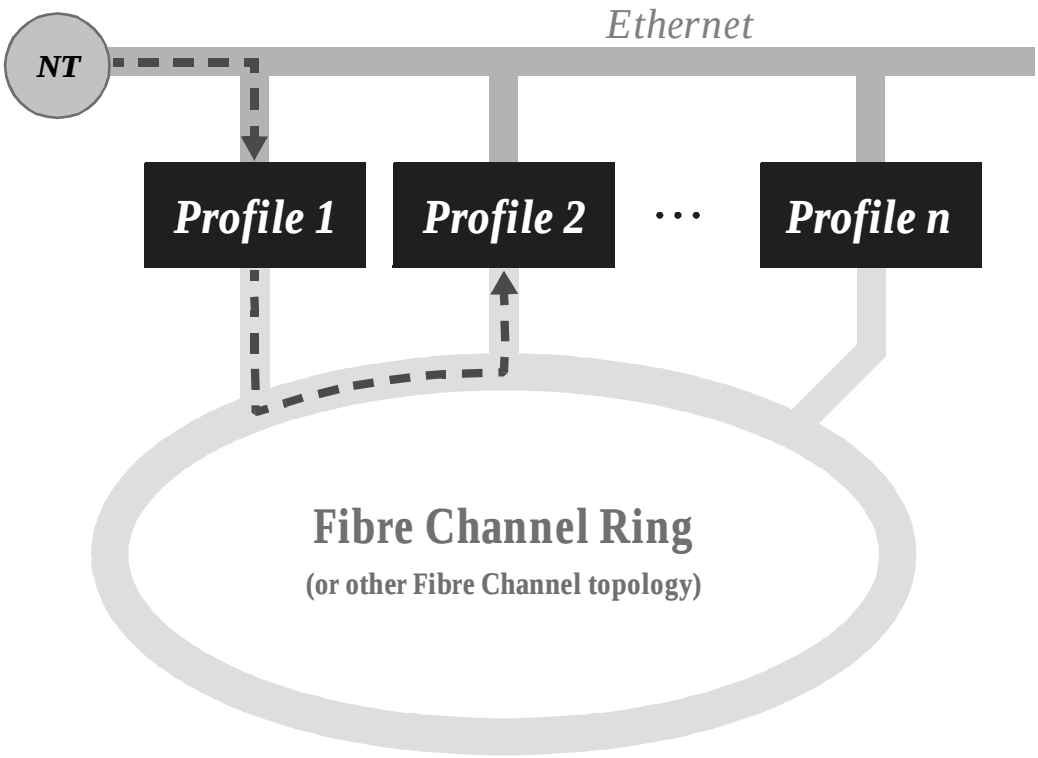
Streaming function calls keep track of transfers by means of the XfrToken, which is specific only to streaming operations. With XfrToken, it is possible to make any number of transfers, though performance may degrade as the number of transfers increases. A transfer request may be queued if bandwidth limitations keep a request from being performed immediately.

The source and destinations of the transfer are specified by a Uniform Media Locator (UML). For a detailed explanation of UMLs, see *UML descriptions* on page 112.

Sample code: Fibre Channel streaming

Figure 8 illustrates what happens in a typical Fibre Channel streaming file transfer. (Although a Fibre Channel *ring or loop* topology is shown, a Fibre Channel *fabric* topology relies on the same principles.)

Figure 8. Fibre Channel file transfer



First, the Fibre Channel transfer request travels over Ethernet from the NT operating system to **Profile1**. Then **Profile1** initiates a Fibre Channel transfer of the media file *mymovie* over to **Profile2** with the name *mycopy.* The example uses fully qualified UML names to avoid any ambiguity.

Example 14, xferumls.c demonstrates the use of Fibre Channel streaming transfers. A source clip, designated by a UML, is copied to a destination using the Fibre Channel transfer API. Bandwidth and total time are reported as status output. Given that *xferuml.exe* iss executed on a remote, non-Profile Windows NT system over Ethernet, the command would be formed like this:

```
xferuml Profile1 Profile1/explodedFile/INT:/default/mymovie Profile2/explodedFile/INT:/
default/mycopy
```

The C code required to accomplish this is relatively straightforward. Some initial setup and parsing of the command line is performed. Then the API functions **XfrRequest**, **XfrGetStatus**, and **XfrAbort** are called to control transfer of data through Fibre Channel.

For a similar example using Fibre Channel to perform simple (nonstreaming) copy operations, see *Example 13, deepcopy.c* on page 119. The Profile SDK abstracts the transfer control to a much higher level.

Example 14. xferumls.c

```
//
// File: xferuml.cpp
// Demonstrates using Fibre Channel transfers.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
#include <windows.h>;
#include <stdio.h>;
#include "tekpdr.h"
#include "tekrem.h"
#include "tekxfr.h"

// This function is used if the input is either a ? or the word help.
// It shows the usage of the program.

void Usage(const char *progName)
{
    printf("Usage:\n");
    printf("%s srcProfile srcUml dstUml\n", progName);
    printf("   srcProfile   Profile to run streamcopy from\n");
    printf("   srcUml       Source UML\n");
    printf("   dstUml       Destination UML\n");

} // Usage

// In the main program, we determine if the Profile is remote, and
// open the connection if it is; otherwise, the connection stays
// local.  We also allow ? or help on the command line in order to
// get the usage printout.

int main(int argc, char *argv[])
{
    ConnectHandle srcConn = LOCAL_CONNECTION;
    char* srcHost = NULL;
    char  lochost[82];
    char*  srcUml;
    char* dstUml = NULL;
    char* dstList = NULL;
    const char* dstArray[1];
    DWORD tickStart, tickFinish;
    XfrToken    waitxfer;
```

```
// Parse command line arguments.
if (argc != 4) {
    Usage(argv[0]);
    return 0;
}
srcHost = argv[1];
srcUml = argv[2];
dstUml = argv[3];

// Validate the source movie.
gethostname(lochost, 82);
if (strcmp (srcHost,lochost) != 0) {
    if (!RemOpenConnection(ConnectEthernet, 0, srcHost, &srcConn)) {
        printf("ERROR: Failed to connect to %s, error code 0x%x\n", srcHost,
            GetLastError());
        return 1;
    }
}
printf("Source Profile %s connection is 0x%x\n", srcHost, srcConn);
printf("The source UML is: %s\n", srcUml);

// Only one destination from the command line.
printf("The destination UML is: %s\n", dstUml);
dstArray[0] = dstUml;

// Initiate the transfer.
tickStart = GetTickCount();

waitxfer = XfrRequest(srcConn, srcUml, 1, (const char **)dstArray);
if (waitxfer == 0) {
    // We failed to get wait token.
    printf("ERROR: XfrRequest token is 0, error code 0x%x\n", GetLastError());
    if (srcConn != LOCAL_CONNECTION)
        RemCloseConnection(srcConn);
    return 1;
}
printf("    WaitToken value is 0x%x\n", waitxfer);

for (;;) {
    const char* thisArg;
    double bandwidth, average;
    XfrStatus status;

    if (!XfrGetStatus(waitxfer, &status)) {
        printf("\nERROR: XfrGetStatus call failed, error code 0x%x\n",
            GetLastError());
        break;
    }
```

```
            switch (status.state) {
            case XFR_ST_ERROR:
                thisArg = "XFR_ST_ERROR";
                printf("\nERROR: Status error code %d\n", status.error);
                break;
            case XFR_ST_QUE:
                thisArg = "XFR_ST_QUE";
                break;
            case XFR_ST_ACTIVE:
                thisArg = "XFR_ST_ACTIVE";
                break;
            case XFR_ST_DONE:
                thisArg = "XFR_ST_DONE";
                break;
            case XFR_ST_LAST:
                thisArg = "XFR_ST_LAST";
                break;
            default:
                thisArg = "UNKNOWN";
                break;
            }
            bandwidth = (double)status.lastBandWidth / 1000000.;
            average = (double)status.averageBandWidth / 1000000.;
             printf(" %06d  %3d%%  BW %6.3f  ave %6.3f  %-15s\r", GetTickCount()%1000000,
                    status.percentCompleted, bandwidth, average, thisArg);
            fflush(stdout);
            if ((status.state != XFR_ST_ACTIVE) && (status.state != XFR_ST_QUE))
                break;

            Sleep(990);
        }

        tickFinish = GetTickCount();

        XfrAbort (waitxfer);
        printf("\nTotal time: %d\n", tickFinish - tickStart);

        if (srcConn != LOCAL_CONNECTION) {
            RemCloseConnection(srcConn);
        }

        return 0;

    } // main
```

Chapter 7

Programming the Profile Library System

The Profile Library System (PLS) uses the client/server model of computing. There is one instance of the library server for each library. This server has a catalog that describes the contents of every tape cartridge loaded in a library. The catalog is a cache for tape cartridge directories. It can also retain residual knowledge of cartridges that have been removed from the library and stored elsewhere.

The library server works with files as a basic unit of information. A file can be a simple stream of bytes or a multiplexed stream of video, audio, and timecode. The library system copies files from a Profile system to tape cartridges and back, but does not delete files from a Profile unit; file management is handled by **Media Manager** or other third-party applications.

The programming model can be approached from two distinct views. First is the ANSI C function calls. The second model supports serial protocols used over RS-422 and Ethernet. We have striven for a consistent model and feature set between these two programming paradigms. In some cases, the limitations of one model have shaped the features of both. The API is available for local applications and as remote procedure calls using the SDK over the Ethernet.

The C programmer's model is based on ANSI C functions. These are implemented as a remote procedure call library that binds function calls to a network transport protocol. Memory management is the responsibility of the application. This model lowers the risk of network-wide memory leaks. The library server uses a set of handles or identifiers to pass linkages to objects the application references or reserves. The use of handles supports communication of object references across serial links. This mechanism is prone to leaks if the application programmer is not careful. Leaks are due to the application not closing handles before exit. A concurrent execution model is supported for long commands.

The byte stream serial protocol is modeled after the existing RS-422 Profile protocol. We have limited the length of transport packets in the command set because of the limits imposed by the current RS-422 packet framing model.

Programming model and serial protocols

The programming model can be approached from two distinct views: the ANSI C function calls and serial protocols used over RS-422 and Ethernet. We have tried to make a consistent model and feature set between these two programming paradigms. In some cases, the limitations of one model have shaped the features of both. The API is available for local applications and as remote procedure calls using the SDK over the Ethernet.

A C programmer's view

The C programmer's model is based on ANSI C functions. These are implemented as a Remote Procedure Call library that binds function calls to a network transport protocol. Memory management is the responsibility of the application. This model lowers the risk of network-wide memory leaks. The library server uses a set of handles or identifiers to pass linkages to objects the application references or reserves. The use of handles supports

communication of object references across serial links. This mechanism is prone to leaks if the application programmer is not careful. A concurrent execution model is supported for slow commands.

Serial protocols

A byte stream serial protocol will be implemented that is modeled after the existing RS-422 protocols. Several decisions in the command set have been made to limit the length of transport packets because of the limits imposed by the current RS-422 packet framing model.

Library concepts overview

Here's a description of many of the basic concepts used to design the library server.

Local library catalog

The library server keeps a catalog of all files in the attached library. The purpose of this database is to allow a fast search for a given piece of material and to support requests for lists of the available material. When cartridges are removed from a library, the application can have all references to material on the cartridge removed from the catalog. This is useful when cartridges are not going to be used in the near future or are being sent to other facilities. The catalog entries for removed cartridges can be retained. Having entries in a catalog makes locating the material faster. The catalog knows the cartridge is not in a library, and it has a field noting where the cartridge is stored or where it can be placed.

Cartridges and partitions

Tape cartridges are identified with unique barcode labels. Barcode labels are used so machine and human readable cartridge identification is available.

You can subdivide cartridges into partitions. A partition can be treated as if it were a separate tape for material replacement purposes. The first partition on a cartridge (at the load point) will be used to store a master cartridge directory. The master cartridge directory need not be the same size as the rest of the partitions. The remaining partitions will all be of equal size and used to store data and media files. From the customer's view, partition numbers start at 1 and ascend as you move away from the cartridge's load point. Partition numbers **PlsAnyPartition** and **PlsNoPartition** (`-1`) are reserved to mean no partition or all partitions, depending on the context.

The code design allows a directory partition to be of different size than data partitions. Data partitions must all be the same size. Some vendor restrictions may require that all partitions be the same size.

Several types of tape cartridges may exist in a library. One type is clip, media and data file archive cartridges. Another is tape transport cleaning cartridges. Each cartridge must have unique barcode label.

Files

Files are the basic units of data the library system reads and writes. They can be one of several types:

- **A data file.** A simple data file, a stream of bytes. It must be local on the Profile server attached to the PLS machine.
- **A clip file.** Audio, video(motion-JPEG encoded), and timecode media files multiplexed

into an MPEG program element stream. *MPEG compression formats are not used in Profile system software version 2.4.*

An archive clip file is a degenerate case of a movie. It has video streams, audio streams, and possibly time codes. An archive clip file is a flat or rendered media stream.

- **A published (finished) movie file.** This is a multiplexed collection of audio, video, timecodes, and control track information. Material is stored in the playout time sequence and unused material is not archived.

Barcode labels

Tape cartridges are identified with barcoded labels. The library server supports barcode label identifier strings of up to sixteen ASCII characters. While the design allows for sixteen characters, the number offered by vendors may differ.

At each installation, barcode identifiers must be unique. If required, an operator can replace a duplicate label on a cartridge from another facility, then load it into a library.

Strings and file names

Strings contain only printing characters and white space, that is, tab and space through tilde. A null pointer to a string is considered to be a null string. Strings will always be null terminated.

File names are strings that identify the file. The path string (dataset name and file path prefix) is omitted from file names in a library. The file name on a cartridge may have a partial Profile file system path as well as the formal Profile file name and extension. File names must be unique in each partition. Files in a library are found by qualifying the names with a cartridge's label.

The name spaces are flat for archived files. The library server will keep files in the order they are stored on cartridges, not in a hierarchical file system tree.

File names are a subset of ASCII strings. The file name character set excludes: `NULL` (0x00) to US (0x1f), `*`, `?`, and `<DEL>` (0x7f). Case differences are not considered when comparing names. The case of provided strings will be preserved by the library server. Names that only differ by white space at the end will be considered matches; trailing white space is removed. A null pointer to a name is considered equal to a null name.

Resource reservation

Some types of system resources, tape transports, and tape cartridges can be reserved by an application or operator. Resource reservation is intended to control tape transport and cartridge access when time critical operations are underway. It is assumed system resources will not be reserved for extended periods of time.

When a single application is driving the library server, the application may implement its own resource reservation model. When multiple applications share a library server, they should reserve resources that will be required in the near future. Applications may also want to reserve resources so other applications using the same library server will not cause unexpected problems.

In/out points

In/out points are used to control what portions of audio/video media files are archived or restored.

When an audio/video media file is archived, a pair of in/out points specifies the starting and ending locations of the material to be saved. The mark-in and mark-out points for a movie are saved with the file.

An optional set of in/out points can be used with a restore command. This in/out pair is used to restore a portion of the material from a file.

Field numbers

Fields in a media file are addressed in time with field numbers. The first field number in a media file or clip is 0.

Multicartridge sets

A file may not fit on a single cartridge. In this case, the library server will split the material between cartridges as needed. Multicartridge files will be stored on separate tape cartridges (cartridges with no other material). All of the cartridges used in a multicartridge set will be taken from a pool of free cartridges. If the location of material on multicartridge sets is not constrained, a facility's cartridges would become interwoven.

The library server will decide if a multicartridge set is required when an archive command is starting. If one of the media files is having material added while an archive command is proceeding, the results will be unpredictable and can be undesirable.

Commands that reference a multicartridge set should use the cartridge label for the first cartridge. An exception is the import cartridge command. It does not know a multicartridge set is being processed so it deals with individual cartridges.

When the material on a multicartridge set is deleted, the set is broken up. The cartridges are all returned to the pool of free cartridges.

Material categories

Cartridges can be assigned to categories. These are used when an application wants the library server to select cartridges for storing files from a known group of cartridges. All of the cartridges in a multicartridge set belong to the category specified for the first cartridge.

The programming model

In several cases the library server uses handles to bind resources between commands. These handles are integers (actually void pointers) and can easily be used in C and C++ function calls or network protocols. Handles are used as a compact representation of a resource or a connection. They also represent the link between an application and an active resource or other system state information.

In some instances an application can offer NULL for a handle to the library server. This means the library server is free to use any resource that is available.

Connection and library handles

An application must establish communications with the library server using the TekRem library. To acquire a connection handle, use **RemOpenConnection**. This function returns a connection handle that is used with three function calls. These return the library server's major or minor version numbers or open a library server session.

When an application wants to work with a library server, it opens a library handle. This is logically a connection handle that can reference a local Profile or one somewhere on the network. Each library handle has its own set of state variables. Some examples are the file system path, a name that identifies the operator or application, and so on. As transport handles, cartridge handles, file handles, and loop handles are created, they all are linked back to the parent library handle. Closing a parent handle closes all associated child handles.

Loop handles are a special case. They are created when an application wants to acquire a potentially long list of entries, like a bin or directory listing. A loop is opened and the handle is returned with a **FindFirst** command. The loop handle serves as a reference point by logically pointing to the state vector for that loop.

Applications should always close handles when they are no longer needed. You must use **PlsOpCodeGetAnyEvent** to close down **TransActionHandle**. Closing a parent handle does not automatically close a child. If these handles are not closed, the resources used by the handle will not be released and eventually the library server could fail for lack of free handles.

Library server API memory model

When a structure is to be returned, the application must provide a pointer to memory. The library server builds a structure in the application's memory space. The application is responsible for allocating and releasing this memory. The library server does not return large structures with any single command. Collections of objects that require significant amounts of space are returned one at a time.

Operations returning multiple data items

Some operations return bin maps and directories that can have a large number of entries. These operations return results using an application-supplied pointer to memory and looping constructs. One command is issued to start the looping operation. This is a loop-type **FindFirst** command. It may have parameters that allow starting an operation at a location other than the beginning. The loop-type **FindFirst** command returns a handle that can be used with the loop-type **FindNext** command to retrieve successive responses. The loop is terminated with a loop-type **Close** command.

An application must close an active loop handle. If it does not, the library server will eventually run out of memory.

If a **FindFirst** command is issued for an empty list, the returned handle has the value of NULL. In this case no active loop handle exists.

Some of the looping commands can restrict the set of items returned. This is done with an action code parameter used in the loop-type **FindFirst** command.

Concurrent command execution

Some commands require long periods of time to execute. One of the parameters to these commands is a pointer to a transaction handle. If this pointer is NULL, synchronous command execution will occur. If the pointer to a concurrent command transaction handle is not NULL, a handle is returned that can be used with a status command to track the concurrent command execution. This handle can also be used to attempt to cancel concurrent commands. Concurrent commands are considered complete when a command completion event is processed.

While concurrent commands are executing, additional commands can be queued. The number of commands that can be queued depends on physical and logical resources limitations. The number of commands that can be queued is not specified. A single application's commands are always executed first in, first out. The order of execution of commands between different applications is not specified.

Error codes

All commands return a Boolean, an integer, or a pointer. By convention, if the returned value is 0, an error has occurred. A nonzero value can be useful information or simply a successful command completion code. An error code can be acquired by calling **GetLastError**.

When a command is executed asynchronously, the initial function call may fail (that is, return a 0) if something is wrong, in which case no concurrent execution will have started. If the return value is not zero, an asynchronous execution will start.

Asynchronous commands send completion events when they terminate. Concurrent command completion events contain an error code as part of the event structure. This error code describes the execution after the successful start. It will be zero if the command has completed successfully.

The inversion of the meaning of 0 from a function call return value (0 is an error) to event error codes (0 means success) must be carefully considered when writing applications.

Configuration, status, and information commands

Configuration commands return data that describes the *physical components* installed in a given system, such as the number of bins in a library. **Status** commands return data that describes the *current state*, such as the number of bins that contain a cartridge. **Information** commands are used when combining configuration and status information is appropriate.

Important notes and assumptions

It is important to consider the following notes when developing applications for library servers:

- Archive and restore operations will use the same resources as disk recorder operations. As a first approximation, one archive data stream requires the same resources as one video stream. Since audio and video I/O takes precedence over all other operations, full-speed tape operations may require shutting down disk recorder channels.
- Each library may be attached to only one Profile system, and each Profile system may have only one library attached to it.
- Media files that only contain audio information will be addressed in video field time units.
- Protection for dangerous operations is the responsibility of the application level software, not the library server. Examples of dangerous operations are reformatting cartridges and deleting files.
- The library server will not move files. Files are copied.
- When current status information cannot be acquired from a device, the last accurate status information is returned. For example, many tape drives will not return status while a format is in progress, so the library server returns status information acquired when the format operation was started.
- The PLS 200 can queue up to 250 commands.

- The PLS 200 can queue up to 100 asynchronous events (that is, events which are not command completion events).

Configuration

A library with several tape transports attached to a single Profile system. Transports without libraries are attached to a Profile system that supports the library server software. This is named a stand-alone transport.

Tape partitioning

Digital computer tapes do not allow replacement of embedded portions of the stored data. New data can be appended to existing data, at the cost of losing access to all data stored beyond the freshly written data.

Since some types of material handled in broadcast facilities have a short life with high turnover, the inability to replace parts of the data on a cartridge limits the applications. One solution is the division of the tape into segments or partitions that can be individually written without altering the contents of other partitions.

To support this facility, tapes must be formatted with special markers and buffer zones between each partition. The buffer zones are used accommodate variances between drives and media. They guarantee that under all circumstances the data in one partition can be replaced without altering the information in the following partition.

When tapes are divided into more and more partitions, the buffer space and other overhead grow as a percentage of the total tape. Eventually it becomes impractical to increase the number of partitions. On some drives, other factors limit the number of available partitions.

Since the division of the tape is limited, the minimum size may be larger than is desired in some applications. All current tape drives either don't support partitioning, or limit the minimum partition size. Unfortunately for some broadcast applications, the minimum partition size is not as small as would be desirable. For PLS 200 tape drives the minimum size is approximately 200MB. This is about 30 seconds of high quality material or 55 seconds of highly compressed material. (See *Table 7, PLS tap‘e partitioning* on page 136 for tape partitioning specifics.) Applications that work with tapes that are approaching their total capacity should handle errors from tape overflows, as well as hard write errors.

Several variables alter the storage requirements, including how old the tape is, how worn the heads are, when the drive was last cleaned, and how clean the video signals are.

Table 7. PLS tap‘e partitioning

Number of partitions	Partition size (MB)	Partition capacity (H:MM:SS) at given video rates (MBps)			
		24	32	40	48
63	199	0:00:56	0:00:44	0:00:36	0:00:30
62	199	0:00:56	0:00:44	0:00:36	0:00:30
61	199	0:00:56	0:00:44	0:00:36	0:00:30
60	199	0:00:56	0:00:44	0:00:36	0:00:30
59	199	0:00:56	0:00:44	0:00:36	0:00:30
58	200	0:00:56	0:00:44	0:00:36	0:00:30
57	207	0:00:58	0:00:45	0:00:37	0:00:32
56	212	0:01:00	0:00:47	0:00:38	0:00:32
55	218	0:01:01	0:00:48	0:00:39	0:00:33
54	225	0:01:03	0:00:49	0:00:40	0:00:34
53	231	0:01:05	0:00:51	0:00:42	0:00:35
52	238	0:01:07	0:00:52	0:00:43	0:00:36
51	245	0:01:09	0:00:54	0:00:44	0:00:37
50	252	0:01:11	0:00:55	0:00:45	0:00:38
49	260	0:01:13	0:00:57	0:00:47	0:00:40
48	268	0:01:15	0:00:59	0:00:48	0:00:41
47	276	0:01:18	0:01:01	0:00:50	0:00:42
46	285	0:01:20	0:01:03	0:00:51	0:00:43
45	294	0:01:23	0:01:05	0:00:53	0:00:45
44	303	0:01:25	0:01:07	0:00:55	0:00:46
43	313	0:01:28	0:01:09	0:00:56	0:00:48
42	324	0:01:31	0:01:11	0:00:58	0:00:49
41	334	0:01:34	0:01:13	0:01:00	0:00:51
40	346	0:01:37	0:01:16	0:01:02	0:00:53
39	358	0:01:41	0:01:19	0:01:04	0:00:55
38	370	0:01:44	0:01:21	0:01:07	0:00:56
37	383	0:01:48	0:01:24	0:01:09	0:00:58
36	397	0:01:52	0:01:27	0:01:11	0:01:01
35	412	0:01:56	0:01:30	0:01:14	0:01:03
34	428	0:02:00	0:01:34	0:01:17	0:01:05
33	445	0:02:05	0:01:38	0:01:20	0:01:08
32	462	0:02:10	0:01:41	0:01:23	0:01:10
31	481	0:02:15	0:01:46	0:01:27	0:01:13
30	501	0:02:21	0:01:50	0:01:30	0:01:16
29	523	0:02:27	0:01:55	0:01:34	0:01:20
28	546	0:02:34	0:02:00	0:01:38	0:01:23
27	571	0:02:41	0:02:05	0:01:43	0:01:27
26	597	0:02:48	0:02:11	0:01:47	0:01:31

Table 7. PLS tap‘e partitioning (Continued)

Number of partitions	Partition size (MB)	Partition capacity (H:MM:SS) at given video rates (MBps)			
		24	32	40	48
25	625	0:02:56	0:02:17	0:01:52	0:01:35
24	657	0:03:05	0:02:24	0:01:58	0:01:40
23	690	0:03:14	0:02:31	0:02:04	0:01:45
22	727	0:03:25	0:02:40	0:02:11	0:01:51
21	768	0:03:36	0:02:49	0:02:18	0:01:57
20	813	0:03:49	0:02:58	0:02:26	0:02:04
19	862	0:04:03	0:03:09	0:02:35	0:02:11
18	917	0:04:18	0:03:21	0:02:45	0:02:20
17	978	0:04:35	0:03:35	0:02:56	0:02:29
16	1046	0:04:54	0:03:50	0:03:08	0:02:39
15	1124	0:05:16	0:04:07	0:03:22	0:02:51
14	1213	0:05:41	0:04:26	0:03:38	0:03:05
13	1316	0:06:10	0:04:49	0:03:57	0:03:21
12	1436	0:06:44	0:05:15	0:04:18	0:03:39
11	1578	0:07:24	0:05:46	0:04:44	0:04:01
10	1749	0:08:12	0:06:24	0:05:15	0:04:27
9	1957	0:09:11	0:07:10	0:05:52	0:04:58
8	2219	0:10:24	0:08:07	0:06:39	0:05:38
7	2552	0:11:58	0:09:20	0:07:39	0:06:29
6	3001	0:14:04	0:10:59	0:09:00	0:07:38
5	3631	0:17:02	0:13:17	0:10:53	0:09:14
4	4575	0:21:27	0:16:44	0:13:43	0:11:38
3	6156	0:28:52	0:22:31	0:18:28	0:15:39
2	9343	0:43:49	0:34:11	0:28:01	0:23:45
1	18999	1:29:06	1:09:31	0:56:59	0:48:17

Library server commands

Table 8 below lists the command set available from the library server. These commands are implemented as ANSI C functions.

Table 8. Frequently used C function parameters

Parameter	Description
action code	An enumeration type that specifies special actions in a command. This may restrict or expand a search or request one of several processing options.
barcode label	See cartridge label.
bin class	A string that can be used to determine the capabilities of a given bin. Some libraries store multiple kinds of cartridges that are physically incompatible.

Table 8. Frequently used C function parameters

Parameter	Description
bin number	An address assigned to a bin in the library. Bin numbers start at `0` and increase to the maximum number, `bins-1`. The bin map is hard-wired so applications can draw graphics using bin numbers to show locations in a GUI.
cartridge class	A string describing the type of cartridge. This can be used to find out capacities and other attributes associated with each type of cartridge.
cartridge description string	A string that can be used to store user data.
cartridge handle	An identifier that points to a specific cartridge and partition. If a NULL is used, the library server is free to use any cartridge. This can be set up by an application to point to a cartridge category, or a specific cartridge.
cartridge label	This is a string that matches the barcode label from a tape cartridge. These values only change if the printed barcode label on a cartridge is replaced.
category	A cartridge can be assigned to a category that is used to organize material for long term storage.
field number	This represents an in or out point expressed in time units of fields. The field's number is established when the original source media file is recorded.
file description string	A string that can be used for user defined-data.
file name	A generic string with a file name.
library handle	A library server identifier that is a library server connection handle.
library name	A name (string) that is the host network name for the machine on which a library server runs.
location info	This is a string that describes where a cartridge is stored if it has been removed from a library with an export command.
loop handle	A library server returned identifier that is used in looping commands.
partition number	Specifies the partition or segment of a cartridge to be used. A partition number of PlsAnyPartition or PlsNoPartition (-1) implies the whole cartridge or no partition number specified.
path	A partial directory path used when the library server references a part of a Profile file system.
return values	All commands return a boolean, an integer or a handle. In some cases these are success/fail codes with the value of 0 used to indicate an error. When a function returns a handle the value of NULL indicates an error. GetLastError must be used to get detailed error information.
session name	A string established when a library connection is opened. This string is returned by some of the status commands so one can discover who is using (reserving) devices or other resources. The session name has no meaning to the library server.
time date	This is a time and date stamp used as the system's wall clock. It is not intended for precise real time operations.
transaction handle	A handle that can be used to get the status or cancel an asynchronous command. The transaction handle is returned with the event that completes a concurrent execution.
transport class	This is a string that can be used to identify what type (vendor and model) of tape transport is installed.
transport handle	A system-returned handle for a tape transport.

Table 8. Frequently used C function parameters

Parameter	Description
transport number	This is a number that identifies a specific tape transport. Transport numbers are device addresses that have assigned values that only change when a library is reconfigured. Transport numbers are always in the range of 0-255.

File selection rules

Files are located in the Profile file system using the following concepts. A file is specified with a stored path name (dataset name and file name prefix) and a file name string from a command. The resulting complete file name is used to find the material on a Profile.

The dataset name is the volume name for the Profile to be used. It must be the Profile to which a transport is attached for archive and restore operations. The path string is not stored by the library server as part of a file name. If a dataset name is not given, the default is the machine to which the library is physically connected (the host Profile).

Cartridge selection rules

A specific cartridge can be requested by acquiring a handle for that cartridge. An application can also use a handle value of NULL to indicate any cartridge can be used to archive new material. In this case, the library server will attempt to find a cartridge with enough free space.

A cartridge can be assigned to a category by providing a category name when the cartridge is formatted. The library server can be requested to limit the search when archiving new material to cartridges in a selected category. This is intended to be used for storing departmental material on a limited set of cartridges.

In many commands, a cartridge handle or cartridge label must be provided so the library server can find the correct cartridge. Cartridges can be reserved when an application wants to control access to the cartridge.

For stand-alone transports, if an application does not specify a cartridge, only cartridges in ready transports will be considered for use.

Tape transport selection rules

In most commands, a transport handle of NULL means the library server can select any free transport. The exceptions are commands where selecting an arbitrary transport yields unpredictable results. For example, requesting the status of an arbitrary transport will not be meaningful.

An application can specify a transport by reserving a transport and using that handle in a command. This is useful when many commands should be executed in sequence or for near real time operations.

An application can get a transport handle without reserving the device. This is important for inquiring about a device's status without waiting in a device reservation queue. In some applications, using an unreserved transport handle can lead to undesirable and unpredictable waits.

For stand-alone transports, the application must specify a transport.

Transport load/unload rules

Cartridges are loaded into transports on demand. The load request can be an explicit PlsLoadTransport command or an implicit load request. A load request may result in an implicit unload request if all transports are loaded and a transport appears to be idle. A transport is idle when the transport is not reserved and no commands are using the loaded cartridge.

An unload request to a transport that is not reserved will always be executed. If a transport is reserved, all commands from users other than the device's owner will be rejected until the device is released.

If an application wants to control when cartridges are loaded and unloaded, it should reserve the transport. Reserved transports will never be implicitly unloaded by the library server.

Application need to do some kind of optimization to keep multiple concurrent commands from "thrashing" cartridges. For stand-alone transports, the application must preload a cartridge. The library server does not have a robot to do an automatic load operation.

Library server API function descriptions

The following sections describe each of the C function calls. This set of function call descriptions is organized into groups of similar functions.

Library functions

The following functions are used to open and close communications with a library. This group also supports operations that act on the entire library or the catalog of known files and cartridges. These functions can be used to get the major and minor version numbers for the current library server. They do not require an open library handle.

The **PlsGetMajorVersion** and **PlsGetMinorVersion** functions retrieves the major and minor version numbers for the current library server. Neither requires an open library handle.

PlsOpenLibrary returns a library connection handle, given a remote connection handle. The remote connection handle can be acquired with RemOpenConnection. The library name is for future use.

> ***The connection handle for the local machine is LOCAL_CONNECTION. The library handles for the local machine will not be NULL.***

The session name is used to identify who owns a resource. It is returned with status commands. The session name does not alter the execution of a command, nor does it control how resource are allocated.

This command also has version information parameters. These represent the version of the library server an application requires for correct operation. If the requested version is current or the library server can perform all functions for the requested version, the connection is accepted.

Power up and reset server initialization runs in the background. This reduces the Profile system start up time. If an attempt is made to open a library connection before the robot initialization process is complete, an error is reported and the library connection is refused. When this occurs, the application should wait for a few seconds and attempt to open the library connection again. The error issued for a server that is still initializing is: PLS_AS_INIT_INCOMPLETE.

PlsCloseLibrary logically closes a user's or application's session with a library server. The communications link established with Prolink/TekRemote is not disrupted. The library handle is no longer valid.

PlsGetLibraryConfig returns a description of a library. This command returns the total bins in the library, number of robots installed, and so forth.

PlsGetLibraryStatus requests a status report on a library. Volatile library data and device-specific information will be returned. The device-specific information will include usage data, error recover information and so on. The report includes information such as whether the library is active, the number of empty bins, and so on.

Transport configuration commands

The following set of commands is used to get a list of all transports under the control of a library server. The library server is bound to a single library (robot).

The starting transport number can be specified as PlsAnyTransport. This will start the list of transports at the first installed transport.

Library bin information commands

These commands return the bin map for a library. For each bin, an indicator is returned describing the bins status and contents. A bin can contain a data cartridge, a cleaning cartridge, and other types of tapes. A bin may also be empty or unavailable. Many libraries have a few special bins. These include bins designated for cleaning cartridges. This results in some bin usage restrictions based on library design.

The normal case is to return information on all bins. A subset of bins can be selected. The selection operations are controlled with the action code parameter.

The application is responsible for converting bin map addresses to locations in a physical library. This will be required for GUIs to draw a graphic display showing where a cartridge is stored in a library.

This function is implemented with three commands. The first establishes a starting point. This command also returns a handle to be used to retrieve the next entry. The second command is used to acquire each successive entry. The third command is used to terminate the loop, possibly before the list is exhausted.

For stand-alone transports, the **PlsFindFirstBinInfo** will yield an end of bins result; no bins are installed.

List all cartridges commands

This set of commands is used to get a copy of the current library's cartridge inventory. The set of cartridge inventory entries can be restricted with a command action parameter. The following actions are supported:

Library and cartridge directory commands

These commands request a directory (file information) report for a file, partition, cartridge, or library. If the library server has a local catalog, it is used. If the server does not have a local catalog, the operator or application must issue an PlsInventoryCartridge before executing a FileInfo command.

The FileInfo commands will select a subset of files based on the action parameter. The action code is used when the catalog for a complete library is being searched.

The commands **PlsFindFirstHandle**, **PlsFindNextHandle**, and **PlsCloseFindHandle** are used to acquire information on all open or active handles. This provides a snapshot of the current system state. Handles can be asynchronously closed or created by other users, which may produce unpredictable results.

Transport functions

The following group of functions is used to manage tape transports.

PlsConnectTransport acquires a transport handle, given the number (address) for a transport. The user may request any free transport by specifying a transport number of **PlsAnyTransport (–1)**. Acquiring a transport handle does not reserve the device for exclusive use by one application.

PlsAllocateTransport reserves a tape transport for exclusive use. If the transport number is not specified (PlsAnyTransport), the library server will reserve any available device. A transport reservation requests a transport by number, not a transport handle. This was done so the acquisition of a transport handle and its reservation are an atomic action. If these were separate commands, error management software would be difficult to write. This capability is being implemented to support applications that are archiving and restoring material in near real time. It is assumed tape transports will be reserved for short periods of time (tens of minutes, not hours). A second use of reserved devices is to force them into an unusable or logically off-line state.

PlsCloseTransport releases a transport handle when an application no longer needs access to that device. Closing a reserved transport handle releases the transport.

PlsGetTransportStatus requests the status of a tape transport. This command always returns a standard set of parameters and some additional device-specific data. The device-specific information will include usage data, error recovery counts, cleaning cycle codes and so on. This command requires a transport handle. If a transport handle is not specified, an error is issued. Failing to specify a transport handle would cause the library server to select a tape transport, which would yield unpredictable results. The returned information includes:

PlsLoadTransport loads a cartridge into a tape transport. This command is intended to preload transports for near real time applications. If the requested cartridge is already in the requested transport, the command was successful. For stand-alone transports this command is a no-operation if the cartridge is already loaded. It fails if the specified cartridge is not loaded.

PlsUnloadTransport unloads the cartridge in the specified transport and returns it to the library. The cartridge's directory is updated and the cartridge rewound if appropriate. This command is intended to be used to unload transports when the cartridge is no longer needed in near real time applications. Unloading an empty transport is not considered an error. For stand-alone transports, this command performs a rewind/unload tape operation. This is an implicit execution of an export operation in normal mode (the cartridge directories are updated and the cached directory information is removed from the local catalog).

PlsCleanTransport loads a cleaning cartridge into the specified transport and executes a cleaning cycle. When the command is complete, the cleaning cartridge is returned to its storage location in the library if it can be used again. If the cleaning cartridge has been used the maximum number of times, it will be exported. For stand-alone transports that have built-in cleaning functions, these functions will be executed. Otherwise this command is a no-operation.

Cartridge functions

PlsConnectCartridge binds a cartridge label to a handle for a cartridge. This binding is always immediate so the command executes synchronously. An application can request a cartridge handle for a cartridge it has reserved. The library server identifies who has reserved the cartridge by comparing library handles. Multiple handles can be open to the same cartridge.

PlsAllocateCartridge reserves a cartridge. Cartridges are reserved by label instead of a cartridge handle so the acquisition of a cartridge and its reservation are an atomic action. It is assumed this command will be used to reserve cartridges for short periods of time in near real time applications.

PlsCloseCartridge releases and recycles a cartridge handle. It should be used when the cartridge associated with the handle is no longer needed.

PlsGetCartridgeConfig returns a structure with information about a specific cartridge. This command will draw an error if the cartridge handle is for a cartridge category.

PlsGetCartridgeStatus requests status information for a given tape cartridge. The information returned is the same as the cartridge information commands (PlsFindFirstCartridgeInfo).

PlsGetPartitionMap finds the free space in one or more partitions. The user may specify the starting partition number and the number of values to be returned. The maximum number of returned values is limited to PlsPartMapSize. If the starting partition number is 0, the value returned is the space allocated to the tape directory partition. If the tape is not partitioned, the directory size is returned as 0. If the starting partition value is greater than the number of partitions on the tape, an error is returned. If the starting partition number plus the number of partitions is greater than the number of partitions on the tape, the number of returned values is limited to the actual partitions on the given cartridge. In the current implementation, the directory partition size and data partition's free space can't be acquired in a single command. The returned partition size information is a vector of not more than PlsPartMapSize values. The size entries are 32 bit integers in units of megabytes. The function will return the number of partitions and map vector during a synchronous function call as parameters, or with the command completion event for a concurrent command execution.

PlsInventoryCartridge updates the inventory (read the directory) for a given cartridge and update the local catalog. Operators can use this command when they think the local or master catalog and the actual contents of a cartridge are not in agreement.

PlsUpdateCartridge forces directory updates on a cartridge. Delete and Rename will make entries in a local catalog and not update a cartridge directory until the cartridge is loaded into a transport. Exactly how and when tape directory updates are accomplished is not specified, with two exceptions. Cartridge directories are always updated by a **PlsUpdateCartridge** command and by an Export Cartridge (if "forget" is not specified.) This command can be used to force cartridge directory updates when a tape may not be used for a long period of time.

PlsFormatCartridge formats a cartridge for later use. The cartridge label must be provided to insure the correct cartridge is formatted. In systems without barcode readers (stand-alone transports) no verification of the label can be performed. The cartridge master directory records the barcode label at the time the cartridge is formatted. The value of the cartridge barcode label stored on the tape is used as a hint only. The machine-readable barcode label is always considered to be the correct cartridge label. The data partition size is in megabytes. The cartridge is divided into as many equal-sized partitions as possible. If the partition size

does not exactly fill the cartridge, the extra space will be divided equally between all partitions. The master directory size is actually a hint. Depending on the actual tape drive, larger than requested master directories may be written.

On some vendors' transports, the operator has no control over how the cartridge master directory space will be allocated. The API has a parameter that allows the user to specify a master directory size. This will be used if the tape transports support multiple partition sizes. On partitioned cartridges the master directory is stored in a partition at the beginning of tape (cartridge load/unload point.) On unpartitioned cartridges, the master directory will be stored after all data files. An action code is provided with format cartridge. This can be used to force a cartridge into a mode that only allows one file per partition. This is intended to be used for automatic allocation of spots.

With **PlsCopyCartridge,** one tape cartridge is copied to another cartridge. The destination cartridge must be empty, and formatted like the source cartridge. Both cartridges must have the same number and size of partitions. Copying a cartridge does not change the order of the files on a cartridge. It may change the partition the files are in. The options are:

Copy the cartridge and compact files within partitions. Files are not moved to lower partitions on the cartridge. This allows the user to duplicate a cartridge without altering any partition reference information that may be in other system-wide databases.

The following information may have been cached in the local catalog and it will be moved to the cartridges directories:

- Rename file information.
- Delete file information. Deleted files are *not* copied.
- Purge time/date information. Files with expired purge dates are *not* removed.
- File description strings.

PlsExportCartridge removes a cartridge from a library using a library's import/export mechanism. A note can be made in the catalog describing the location of the cartridge. The library server may need to load the cartridge into a tape transport before the cartridge leaves the library. This will be done to update cartridge directory information. Cartridges will not be exported unless the on-tape directories are current. An action code can be set to request special processing:

- Export a cartridge, but don't forget about it (Normal). Update the cartridge directories and physically export the cartridge. The local and master catalog entries are marked to show the cartridge is out of the library.
- Update cartridge directory, export it, and remove entries from the catalog (Retire). This is used when cartridges are going to be sent to another site and the catalogs should be purged. The cartridge's directory is updated if it is not current. A cartridge can be reinstalled by importing the cartridge.
- Export cartridge, don't update cartridge directories and delete catalog entries (Forget). This should be used when a cartridge is lost or damaged. This purges the local and global catalog of information about the cartridge and its files. This can also be used when a cartridge is damaged and the operator wants the cartridge removed without the cartridge directories being updated. If a cartridge leaves a library without updating the directories and is later imported, changes made since the last actual directory update on tape will be lost.

Local catalog entries for cartridges that are not in a library can be deleted using the retire or forget options. Exporting a cartridge from a multicartridge set causes all of the cartridges in that set to be removed form the library. A location information string is provided by the application. This string should have notes that can be presented to the operator so he knows where to store the cartridge. The location information string is saved in the local catalog. If the cartridge already has a location information string and a new one is not provided, the old location string is retained. For a stand-alone transport, this command does the normal export processing, followed by a tape rewind/unload operation to physically export the cartridge.

With **PlsImportCartridge**, a cartridge is accepted from the operator using a library's import/export mechanism. Cartridges that do not have a barcode label will not be loaded into the library.

A cartridge label can be provided by the application. This is used in messages to an operator where appropriate. It is also used to select the correct cartridge in libraries with multiple import/export ports. Cartridge labels are verified by reading the printed barcode label when a cartridge is loaded. Importing a cartridge does not change the location information string-stored in the local catalog. An import command without a cartridge label parameter can be used to load cartridges from another facility, unformatted tapes, and cleaning cartridges. In this case, the location information string can be used for operator messages.

An action code describes the cartridge and what processing should be done. In some cases the library server will reject a cartridge. This means it will be put back in an import/export port. Import cartridge and assume the barcode label accurately identifies the cartridge. If the barcode on the cartridge matches one in the catalog, assume this cartridge's contents match the catalog. If the catalog has no record of the cartridge label, store it as an unknown cartridge for now. If the barcode matches a cartridge already in the library, reject the cartridge.

- Import cartridge and make simple checks to verify contents of tape. If the barcode on the cartridge matches a cartridge already in the library, reject the cartridge. If the barcode on the cartridge matches one in the catalog and the master directory on the cartridge has the correct entries (format time, format date, system name match), assume this cartridge contents match the catalog. If the barcode matches a catalog entry and the master cartridge directories don't agree with the information in the catalog, inventory the cartridge. If the catalog has no record of the cartridge barcode label, store it as an unknown cartridge.
- Read cartridge directories. If the barcode is unique, load the cartridge into a transport and use the cartridge directory to construct local and master catalog entries. If the barcode on the cartridge matches one in the library, reject the cartridge.
- Import a cleaning cartridge. If it does not have a unique barcode, reject the cartridge. If it has a unique barcode, store the cartridge in a cleaning cartridge bin.

Import operations are subject to a time-out. If an operator has not loaded a tape within the allotted time-out, the command will fail. The current time-out value is 5 minutes. A second version of the **PlsImportLoadCartridge** import command does the above import processing and loads the cartridge into the specified transport. **PlsImportLoadCartridge** is intended to be used with stand-alone transports. **PlsImportCartridge** is intended to be used with libraries and stackers.

PlsImportLoadCartridge imports cartridges into a specified stand-alone transport. When used in a full library, it physically imports the cartridge and automatically loads it into the specified transport. A transport handle must be specified. With two exceptions, this command is identical to **PlsImportCartridge**.

- The cartridge is loaded into the specified transport.

- The barcode label may not be tested for validity. Stand-alone transports may not have barcode label readers.

PlsSetLocationString changes the location information string. It can be used for cartridges that are in a library or out in a storage area. The purpose of this command is to allow errors in the local catalog location information strings to be corrected. This command will fail if another user has the cartridge reserved.

PlsGetLocationString acquires the current location information string stored in the local catalog. An application may use this information to build operator message strings.

PlsSetCartDescription sets a user-specified string that can describe the cartridge's contents.

One cartridge description string is stored per cartridge. **PlsGetCartDescription** returns the cartridge description string.

Basic archive functions

PlsConnectFile provides a file handle for use in later commands. The application must provide a file name. If the file handle refers to an existing file, the application must also provide a cartridge handle and partition number.

PlsCloseFile disconnects a file handle.

PlsGetClipSize allows non-zero in/out points to find the size of a portion of a clip. Size is an estimate based on uniform recording of media throughout the clip. Sections of black media will greatly confuse the estimation process. This function returns the space required for a specified clip when it is written to tape. If the specified in/out points are 0, the value returned is sufficient to store the complete clip. A transport handle can be provided. If the transport handle is NULL, the space returned is an estimate for an arbitrary tape drives. If a transport handle is used, additional space is added for overhead that is specific to the specific tape drive.

Examples of additional overhead are space for tape marks, minimum tape record requirements and overhead that are associated with logical-to-physical blocking. The user must specify a file size pointer or a transaction handle pointer, but not both. If the transaction handle is given (not NULL), the file size is returned with a command completion event. If the file size result variable pointer is given, the function does not return until processing is complete. Note: The information needed to calculate the clip size is on disk, so retrieving the information and calculating the size will require some time.

PlsArchiveClip copies a file from a Profile unit to a cartridge. The machine that receives and executes this command must be the host system for the library. The clips can be on the library host or any machine that has a Fibre Channel connection to the library host. This command always stores media streams as multiplexed MPEG streams. The compressed video track (limited to JPEG for versions 2.2 and 2.4) is multiplex encoded, following the MPEG standard. If a cartridge is not specified, the library server will select a cartridge with enough space to store the file. With an archive operation, a pair of field numbers (in/out points) can be specified.

MPEG video is archivable with version 2.5 software; DVCPRO 25 and DVCPRO 50 video is archivable with version 3.2.

The field numbers for the clip are not altered by an archive or restore command. This prevents the logical field numbers for a clip from slipping as it is archived and restored. The pair of field numbers specifies the in/out points for the part of the clip to be archived. Only material that is actually recorded will be archived. The specification of in and out points beyond the range of the actual recorded material is not considered to be an error. The user-specified in

and out points are returned with a file information request. The default values for the in/out pair is the actual beginning and end of the clip. The default is specified by using 0, 0 as an in/out pair.

The mark-in and mark-out points for the movie are saved with the file. If an out point field number is less than an in point field number, the command will not be executed and an error issued. If the file name specified already exists in this partition, the archive command is rejected with an error. This command returns the cartridge label and partition number used to store the file. These are important when the library server does cartridge and space allocation. If the FileStored parameter is NULL, no data is returned. This is not considered an error. The FileStored data is returned when the command is completed, or with the completion event for concurrent commands. The PlsFileStored structure contains:

- Partition number file was stored in.
- Cartridge label file was stored in.

PlsArchiveDataFile is similar to the **PlsArchiveClip** function except for the type of file processed. This function archives an unstructured byte stream from a Windows NT file system file. The data stream is assumed to be a play list, edit decision list or some other material that is part of the program material. Unstructured files must be archived and restored as a complete unit. The Windows NT file path is not saved on tape. The user must specify where the file should be placed in the Windows NT file system when a restore is issued. This command returns the cartridge label and partition number used to store the file.

PlsRestore copies the specified files from a cartridge to a Profile file system. The machine that receives and executes this command must be the host system for the library. The restored files can be on the library host or any machine that has a Fibre Channel network connection to the library host. For clips, a pair of in/out point field numbers can be used to control the material restored. If these are not specified (0, 0 given), the complete clip will be restored. If the in or out points specified go beyond the range of the recorded material, only the archived material will be restored. No error will be issued in this case. If the requested in point is later than the requested out point, an error will be issued. This has the effect of providing black, silent fields beyond the actual in/out points when the clip is replayed from a Profile. The movie mark-in and mark-out points will be restored with the clip.

PlsRestoreDataFile restores Windows NT data files. It follows the same basic model as **PlsRestore**. The user must supply the location in the Windows NT file system where the restored file should be placed.

PlsRenameFile renames a file on a cartridge. To improve performance, file rename operations will be saved in the local catalog. These changes will be recorded in the cartridge's directory the next time the cartridge is in a transport. Under some serious error conditions (such as a power failure), it is conceivable a file will lose its new name and revert to the old name.

PlsDeleteFile deletes a file from a cartridge. Most tape transports can't delete the material unless it is the last file on a cartridge. Therefore, deleting a file makes the material unreachable, but the space is not reusable. When possible, the library server will recover free space from deleted files. To improve performance, file delete operations will be saved in the local catalog. These changes will be recorded in the cartridge directory the next time the cartridge is in a transport.

Under some serious error conditions (such as a power failure), it is conceivable that a file will reappear on a cartridge. This is better than the alternative, that is, losing material.

A free format file description string can be stored with each file with **PlsSetFileDescription**. This data is saved for the host application's use. If the string is set after a file handle is acquired and before the file is created, the description string will be saved with the file. If it is set or altered at a later date, it will be kept in the local catalog and saved in the cartridge master directory the next time the cartridge is loaded into a transport. **PlsGetFileDescription** returns the file description string.

Library server management functions

The following functions are system support features. They allow management of asynchronous events, the time of day clock, the file name path and other functions.

PlsSetEventMask sets the library server asynchronous event mask. If a bit is set, it enables the specified events. Asynchronous command completion events are always enabled. The following classes of events can be controlled.

- Cartridge import, export, format and delete all files events.
- File add, delete and rename events.
- Transport on-line/off-line events.
- Extended data for import, export, and format on command completion.

The default mask is all zeros, meaning all events except concurrent command completion are disabled.

PlsGetEventMask returns the current event mask value.

PlsGetAnyEvent gets information on asynchronous events and concurrent command completion events. When called, it returns information on one event. The returned data has sufficient tags and other information so an application can process the event. Two other commands, **PlsGetAsynchEvent** and **PlsGetCommandEvent,** can be used to get events from the asynchronous event and command completion queues respectively and these commands offer a conceptually simpler model to the application developer. **PlsGetAnyEvent** requires fewer serial link transactions in polling loops.

The event types include:

- "No Event Queued" indicator.
- Cartridge import (explicit and implicit), export, delete all and format events.
- Archive file creation, deletion and rename notifications.
- Transport on-line/off-line (transport add and remove) events.
- Concurrent command completion events.

All events, except concurrent command completion events, are sent to all open library handles except the one that initiated the action. Concurrent command completion events are saved for the library handle that started the operation. Concurrent command completion events are queued for each open library handle so they will not be lost. Other events may be lost if they are not processed in a timely manner. Cartridge import and export events include the cartridge label. The files added or removed by an import, export, cartridge format or delete all files are not reported with file change events. This is done to prevent event storms when cartridges are moved in and out of libraries. Using the provided cartridge label, an application can get all of the file information using **PlsGetFileInfo**.

For file creation, **PlsArchiveClip**, **PlsArchiveFile**, and **PlsArchiveFile** and file deletion, the cartridge label, partition number and file name are part of the event message. For rename operations, the cartridge label, partition, current file name string and new file name string are returned. Adding or removing transports on a given library causes on-line or off-line events. These allow an application to track the available physical resources. Concurrent command completion events have more information than other events. The actual command completion code (success or failure and coded error number), the handle used to track a concurrent command, and any data a command normally returns are part of the event structure. Concurrent command completion events are only returned for commands associated with the current library handle. An event mask is part of the open library connection state. This mask can be used to enable or disable each class of event.

The command completion events for **PlsFormat**, **PlsImport**, **PlsImportLoad**, and **PlsExport** can have extended results if desired. If the extended data event mask bit is not set, the return information matches all simple command completion events. In this mode, the current version behaves like older versions of the software. If the extended event mask bit is set, the target cartridge barcode label and other useful information is returned. In this mode, the extended data fields are not compatible with old versions.

PlsGetAsynchEvent allows an application to retrieve events other than command completion. It can be used in applications that have separate polling loops for concurrent command tracking and external event management. With the exception of not retrieving command completion events, this command is identical to **PlsGetAnyEvent**.

PlsGetCommandEvent allows an application to retrieve command completion events. With the exception of retrieving only command completion events, this command is identical to **PlsGetAnyEvent**.

The **PlsSetModes** function can be used to alter the fundamental operating modes for the library server. The data returned from a **PlsGetModes** call is identical to the data used in a **PlsSetModes**. An application can read the current library server mode set, alter a single variable, and use the result as the argument of a **PlsSetModes** command. **PlsGetModes** returns the current mode set for the library server.

PlsSetPath sets the dataset and file name prefix directory string used for finding files in the Profile file system. It does not alter the names of files on cartridges. A path string is stored for each open library handle.

PlsGetPath returns the string currently set as a directory path.

This command does not return a backslash (\) symbol at the end of the path string.

PlsGetTimeDate returns the current time of day.

PlsAddTransport adds transports to the library server. Transports appear in a library configuration as soon as they are added. The transport number is an integer in the range of 0 to 255. It specifies the logical address of the transport. The robot address is an integer that describes the physical location of a transport in a library. A Profile name is also supplied that identifies a Profile the transport is attached to. If a Profile name is not given, the machine the library server is running on is used as a default. The device address is a string that describes the low-level device attachment. For the first implementation, the following syntax applies:

`SCSI:<chan>.<scsi-addr>.<scsi-lun>`

Where `<chan>` selects which SCSI interface the transport is attached to and is a single letter. The SCSI device address (`<scsi-addr>`) is specified as a two digit number in the range of 00 to 15. SCSI logical units (`<scsi-lun>`) are not currently used, but are coded as a 00. An example is: SCSI:A.01.00

PlsRemoveTransport removes a transport from the library server control files. As soon as the command executes, the tape transport is no longer usable. A **PlsRemoveTransport** command will be rejected if the transport is in use or reserved when the command is executed.

PlsInventoryLibrary verifies and corrects the bin map and cartridge inventory of the specified library. The old inventory will be used as a baseline to continue operations while the new inventory is in progress. This command checks for cartridges by cartridge labels. Cartridges are not loaded into tape transports to verify their contents. This command can be used when an operator has opened a library and loaded or removed cartridges. It can also be used when an operator thinks the library server's inventory is not correct.

When **PlsHouseKeeping** is executed, the library server will perform systems management functions. The command has two parameters that currently are not used. The first is an integer and should be coded as 0. The second is a pointer that should be coded as NULL. This function also causes the internal directory cache database to remove any entries that are no longer needed.

Local catalog management functions

This set of commands allow the user to save a mirrored copy of the local catalog on an alternate Windows NT system. The directory path must point to a Windows NT machine directory on a specified machine. For example:

`\\<machine-name>\<disk-path>`

The drive name parameter identifies the drive letter on the local machine (where the Library system server resides) to be used to reference the remote machine.

Setting the directory does *not* cause a backup copy to be made. Executing **PlsBackupCatalog** causes the local catalog to be copied to the alternate machine. **PlsBackupCatalog** also causes the local catalog to purge any unused space and do other system maintenance functions. **PlsBackupCatalog** can be issued without a backup directory specified to force local catalog maintenance operations.

Command management functions

These two commands are used to manage concurrent command execution.

PlsGetStatusCommand requests the status of a concurrently executing command. If the command is still in progress, the return code will specify an incomplete command. The percent done valid code will be set if the command is waiting for busy or reserved resources. When actual execution starts, the percent complete value will be set. If a previously started command has finished, the return code describes a successful execution, or specifies what error stopped the execution.

PlsCancelCommand cancels an asynchronous (concurrent) operation. Data (byte stream) files and archive sets (multicartridge sets) will be deleted if a cancel occurs while they are being archived or restored. Clips and media files will be truncated by a cancel, but the material already copied to tape or disk will be retained. The transaction handle was a returned value from another command. The concurrent command will be considered complete when a command completion event is processed.

Sample code: Managing a library system

Example 15, plsdemo.c demonstrates Profile Library System use. It shows how to archive a clip, restore a clip, format a cartridge, inventory a cartridge, import a cartridge, export a cartridge, and back up a local catalog.

Example 15. plsdemo.c

```
//
//
// File: plsdemo.cpp
// A demo program for use with the PLS software library.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
//

#include <windows.h>
#include <stdio.h>
#include "tekpls.h"
#include "tekrem.h"
#include "tekvdr.h"

// Here are the PLS Demo error codes.
#define DEMO_NO_ERROR            0
#define DEMO_INIT_FAIL          -1
#define DEMO_NO_MEDIA           -2
#define DEMO_MISSING_FILENAME   -3
#define DEMO_MISSING_LABEL      -4
#define DEMO_BACKUP_PATH_ERROR  -5
#define DEMO_BACKUP_DRIVE_ERROR -6
#define DEMO_BAD_COMMAND_FORMAT -7

//
// Show the correct usage of the command-line app.
//
void Usage(const char* progName)
{
    printf("Usage:\n");
    printf("%s profileName command [command options]\n", progName);
    printf("    profileName   the name of a remote profile, or 'local'\n");
    printf("    command:      one of the following commands\n");
    printf("        archive   clipName  [plsPath]\n");
    printf("        restore   clipName  [plsPath]\n");
    printf("        format    label\n");
    printf("        inventory label\n");
    printf("        import    label\n");
    printf("        export    label\n");
    printf("        backup    path plsDrive\n");

} // Usage
```

```
// This function is used to wait for a transaction to succeed or
// fail.  Could also have used PlsGetCommandEvent and waited for success .
void WaitTransaction(PlsLibraryHandle hLibrary,
                     PlsTransactionHandle hTransaction)
{
    PlsCmdStatus cmd;
    while ((PlsGetStatusCommand(hLibrary, &cmd, hTransaction))
           && (cmd.status & PlsCmdStatRunning)) {
        Sleep(100);
    }
    return;

} // WaitTransaction


// This command will archive the specified clip to a cartridge chosen
// by the PLS software.  It will try to identify a free transport,
// failing that will let the PLS software make the selection.
int ArchiveClip(ConnectHandle ch, PlsFileNameStr clipName, PlsPathStr path)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsFileHandle hFile = NULL;
    PlsTransportHandle hTransport = NULL;
    PlsLibraryStatus libStatus;     // used to find free transports
    PlsTransportStatus transStatus; // used to find free transport
    PlsTransportConfig transConfig; // used to find free transport
    PlsTransConfigLoopHandle hLoop; // used to find free transport
    PlsFileStored fileStored;
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;
    PlsModes modes;

    printf("\nPreparing to archive clip...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL) {
        return DEMO_INIT_FAIL;
    }


    // Set modes to transport queuing.
    modes.flags = PlsModeTransQueuing;
    if (!PlsSetModes(hLibrary, &modes)) {
        return GetLastError();
    }

    // Get the library status to determine if there are any media
    // cartridges present and if there are any free transports.
    if (PlsGetLibraryStatus(hLibrary, &libStatus)) {
        if (libStatus.mediaBins == 0) {
            printf("\nNo media cartridges\n");
            return DEMO_NO_MEDIA;
        }
    }
```

```
// See if the are any free transports, if so, select one for the archive.
if (libStatus.freeTransports > 0) {
    // PlsAnyTransport means to search all transports.
    hLoop = PlsFindFirstTransportConfig(hLibrary, PlsAnyTransport,
                                        &transConfig);
    if (hLoop) {
        for (;;) {
            if (transConfig.location != PlsStandAloneTransport) {
                // Don't use a standalone.
                hTransport = PlsConnectTransport(hLibrary,
                                transConfig.transportNumber);
                if (!hTransport)
                    break;
                if (PlsGetTransportStatus(hTransport, &transStatus)) {
                    if (transStatus.status & PlsTransStatBusy)
                        hTransport = NULL;
                    else
                        break;
                }
            }
            if (!PlsFindNextTransportConfig(hLoop, &transConfig)) {
                hTransport = NULL;
                break;
            }
        }
    }
}

// Set the path for the clip location in the video file system.
if (!PlsSetPath(hLibrary, path))
    return GetLastError();

// create a handle to the clip file to be archived
hFile = PlsConnectFile(hLibrary,
                       NULL,            // PLS will choose the cart and
                       PlsAnyPartition, // part in the archive command.
                       clipName);

if (hFile == NULL)
    return GetLastError();

    printf("Archiving clip \"%s\"...\n", clipName);

// archive the clip
result = PlsArchiveClip(hFile,
                        0,    // 0 for in and out point will force
                        0,    // the archiving of the entire clip.
                        hTransport,
                        &fileStored,
                        &hTransaction);
if (!result)
    return GetLastError();

// Wait until the archive is complete.
WaitTransaction(hLibrary, hTransaction);

// Get the results.
if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
    return GetLastError();

    printf("Archive complete.\n");
```

```
    return event.e.cmd.returnCode;

} // ArchiveClip


// This command will restore the specified clip to the video file system.
// (Note: The clip can not exist on the video file system.)  Also, the format of
// the restore command will cause the PLS software to try to find a unique clip
// with the PLS file system.  If the clip is not unique, this command will fail.
int RestoreClip(ConnectHandle ch, PlsFileNameStr clipName, PlsPathStr path)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsFileHandle hFile = NULL;
    PlsTransportHandle hTransport = NULL;
    PlsLibraryStatus libStatus;   // Used to find free transport.
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;
    PlsModes modes;

        printf("\nPreparing to restore clip...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
        return DEMO_INIT_FAIL;

    // Set modes to transport and cartridge queuing.
    modes.flags = (PlsModeTransQueuing | PlsModeCartQueuing);
    if (!PlsSetModes(hLibrary, &modes))
        return GetLastError();

    // Get the library status to determine if there are any media cartridges
    // present and if there are any free transports.
    if (PlsGetLibraryStatus(hLibrary, &libStatus)) {
        if (libStatus.mediaBins == 0) {
            printf("\nNo media cartridges\n");
            return DEMO_NO_MEDIA;
        }
    }

    // Set the path in the video file system to where the clip is to be
    // restored.  If a clip by the same name exists in this path the
    // command will fail.
    if (!PlsSetPath(hLibrary, path))
        return GetLastError();
```

```
    // Create a handle to the clip file to be archived.
    hFile = PlsConnectFile(hLibrary,
                           NULL,  // PLS will try and find a unique clip.
                           PlsAnyPartition, // If clip is not unique,
                                            // this will fail.
                           clipName);
    if (hFile == NULL)
        return GetLastError();

        printf("Restoring clip \"%s\"...\n", clipName);

    // Restore the clip.
    result = PlsRestore(hFile,
                        0,        // 0 for in and out point will force
                        0,        // archiving of the entire clip.
                        hTransport,
                        &hTransaction);

    if (!result)
        return GetLastError();

    // Wait until the restore is complete.
    WaitTransaction(hLibrary, hTransaction);

    // Get the results.
    if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
        return GetLastError();

        printf("Restore complete.\n");

    return event.e.cmd.returnCode;

} // RestoreClip


// This command will format the specified cartridge to a single
// partition format.
int FormatCartridge(ConnectHandle ch, PlsCartridgeLabelStr cartLabel)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;
    PlsModes modes;

                printf("\nPreparing to format cartridge...\n");
```

```
    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
        return DEMO_INIT_FAIL;

    // Set modes to transport and cartridge queuing.
    modes.flags = (PlsModeTransQueuing | PlsModeCartQueuing);
    if (!PlsSetModes(hLibrary, &modes))
        return GetLastError();

    // Format the cartridge
    result = PlsFormatCartridge(hLibrary, NULL, PlsFormatCartNormal,
    // Can also be PlsFormatCartSingle
    // forcing only one file per partition.
    cartLabel, NULL, 0,
    // These parameters cause default 0,
    // single partition format. &hTransaction);
    if (!result)
        return GetLastError();

        printf("Formating cartridge \"%s\"...\n", cartLabel);

    // Wait until the format is complete.
    WaitTransaction(hLibrary, hTransaction);

    // Get the results.
    if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
        return GetLastError();

        printf("Format complete.\n");

    return event.e.cmd.returnCode;

} // FormatCartridge
```

```
// This command will cause the specified cartridge to be inventoried.
// This will update the local catalog with the information contained in the
// tape cartridge directory.  This information supercedes the original
// contains of the local Catalog.
int InventoryCartridge(ConnectHandle ch, PlsCartridgeLabelStr cartLabel)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsCartridgeHandle hCartridge;
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;
    PlsModes modes;

        printf("\nPreparing to inventory cartridge...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error  to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
        return DEMO_INIT_FAIL;

    // Set modes to transport and cartridge queuing.
    modes.flags = (PlsModeTransQueuing | PlsModeCartQueuing);
    if (!PlsSetModes(hLibrary, &modes))
        return GetLastError();

    // Connect to the target cartridge.
    hCartridge = PlsConnectCartridge(hLibrary, cartLabel);
    if (!hCartridge)
        return GetLastError();

    // Inventory the cartridge.
    result = PlsInventoryCartridge(NULL, // Let PLS choose transport.
    hCartridge, &hTransaction);
    if (!result)
        return GetLastError();

        printf("Inventorying cartridge \"%s\"...\n", cartLabel);

    // Wait until the format is complete.
    WaitTransaction(hLibrary, hTransaction);
```

```
    // Get the results.
    if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
        return GetLastError();

        printf("Inventory complete.\n");

    return event.e.cmd.returnCode;

} // InventoryCartridge

// This command will import a cartridge into the library system.  The
// cartridge will be imported using the accept mode.  Other modes
// include forcing an inventory, specifying cleaning, backup or
// unformatted cartridges, and simply verfying the cartridge.
int ImportCartridge(ConnectHandle ch, PlsCartridgeLabelStr cartLabel)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;

       printf("\nPreparing to import cartridge...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
         return DEMO_INIT_FAIL;

    // Import the cartridge.
    result = PlsImportCartridge(hLibrary, PlsImportActionAccept, cartLabel,
                                "In the library", &hTransaction);
    if (!result)
        return GetLastError();

       printf("Importing cartridge \"%s\"...\n", cartLabel);

    // Wait until the format is complete.
    WaitTransaction(hLibrary, hTransaction);
```

```
    // Get the results.
    if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
        return GetLastError();

        printf("Import complete.\n");

    return event.e.cmd.returnCode;

} // ImportCartridge


// This command will export a cartridge out of the library system.  The
// Cartridge will be exported using the forget mode.  Other modes force
// a cartridge update, and will ten either leave or remove knowledge of
// the cartridge in the local catalog.
int ExportCartridge(ConnectHandle ch, PlsCartridgeLabelStr cartLabel)
{
    PlsLibraryHandle hLibrary = NULL;
    PlsCartridgeHandle hCartridge;
    PlsTransactionHandle hTransaction;
    PlsEvent event;
    unsigned long err;
    BOOL result;
    PlsModes modes;

        printf("\nPreparing to export cartridge...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
        return DEMO_INIT_FAIL;

    // Set mode to cartridge queuing.
    modes.flags = PlsModeCartQueuing;
    if (!PlsSetModes(hLibrary, &modes))
        return GetLastError();

    // Connect to the target cartridge.
    hCartridge = PlsConnectCartridge(hLibrary, cartLabel);
    if (!hCartridge)
        return GetLastError();
```

```
    // Export the cartridge.
    result = PlsExportCartridge(hCartridge, PlsExportActionForget,
                                "Out of the library", &hTransaction);
    if (!result)
        return GetLastError();

        printf("Exporting cartridge \"%s\"...\n", cartLabel);

    // Wait until the format is complete.
    WaitTransaction(hLibrary, hTransaction);

    // Get the results.
    if (!PlsGetCommandEvent(hLibrary, hTransaction, &event))
        return GetLastError();

        printf("Export complete.");

    return event.e.cmd.returnCode;

} // ExportCartridge


int BackUpLocalCatalog(ConnectHandle ch, PlsBackupPathStr path,
                       PlsBackupDriveStr drive)
{
    PlsLibraryHandle hLibrary = NULL;
    unsigned long err;

        printf("\nPreparing to back up catalog...\n");

    // Open library - loop is used to wait while library system initializes.
    err = PLS_AS_INIT_INCOMPLETE; // Set error to incomplete init.
    while ((hLibrary == NULL) && (err == PLS_AS_INIT_INCOMPLETE)) {
        hLibrary = PlsOpenLibrary(ch, "demo", "demo_session", 0, 0);
        if (hLibrary == NULL) {
            err = GetLastError();
        }
    }
    if (hLibrary == NULL)
        return DEMO_INIT_FAIL;
```

```
    // Set backup path.
    if (!PlsSetBackupDir(hLibrary, path, drive)) {
        err = GetLastError();
        return err;
    }

        printf("Backing up catalog in path %s, drive %s \n", path, drive);

               if (!PlsBackupCatalog(hLibrary)) {
        err = GetLastError();
        return err;
    }

        printf("Back up complete.\n");

               return DEMO_NO_ERROR;


} // BackUpLocalCatalog

int main(int argc, char *argv[])
{
    int result = 0;

    ConnectHandle cHdl = LOCAL_CONNECTION;
    if (argc > 2) {
        // Check to see if this is remote or local.
        if (strcmp(argv[1], "local")) {
            // if remote, host name is argv[1]
            if (!RemOpenConnection(ConnectEthernet, 0, argv[1], &cHdl)) {
                cHdl = LOCAL_CONNECTION;
            }
        }
        if (!strcmp(argv[2], "archive")) {
            if (argc < 4) {
                printf("\nfile name missing\n");
                Usage(argv[0]);
                return DEMO_MISSING_FILENAME;
            }
            if (argc < 5) {
                result = ArchiveClip(cHdl, argv[3], "");
            }
            else {
                result = ArchiveClip(cHdl, argv[3], argv[4]);
            }
        }
        else if (!strcmp(argv[2], "restore")) {
            if (argc < 4) {
                printf("\nfile name missing\n");
                Usage(argv[0]);
                return DEMO_MISSING_FILENAME;
            }
            if (argc < 5) {
                result = RestoreClip(cHdl, argv[3], "");
            }
            else {
                result = RestoreClip(cHdl, argv[3], argv[4]);
            }

        }
        else if (!strcmp(argv[2], "format")) {
            if (argc < 4) {
                printf("\ncartridge label missing\n");
```

```
                Usage(argv[0]);
                return DEMO_MISSING_LABEL;
            }
            result = FormatCartridge(cHdl, argv[3]);
        }
        else if (!strcmp(argv[2], "inventory")) {
            if (argc < 4) {
                printf("\ncartridge label missing\n");
                Usage(argv[0]);
                return DEMO_MISSING_LABEL;
            }
            result = InventoryCartridge(cHdl, argv[3]);
        }
        else if (!strcmp(argv[2], "import")) {
            if (argc < 4) {
                printf("\ncartridge label missing\n");
                Usage(argv[0]);
                return DEMO_MISSING_LABEL;
            }
            result = ImportCartridge(cHdl, argv[3]);
        }
        else if (!strcmp(argv[2], "export")) {
            if (argc < 4) {
                printf("\ncartridge label missing\n");
                Usage(argv[0]);
                return DEMO_MISSING_LABEL;
            }
            result = ExportCartridge(cHdl, argv[3]);
        }
        else if (!strcmp(argv[2], "backup")) {
            if (argc < 4) {
                printf("\nbackup path missing\n");
                Usage(argv[0]);
                return DEMO_BACKUP_PATH_ERROR;
            }
            if (argc < 5) {
                printf("\nlocal device missing\n");
                Usage(argv[0]);
                return DEMO_BACKUP_DRIVE_ERROR;
            }
            result = BackUpLocalCatalog(cHdl, argv[3], argv[4]);
        }
        else {
            Usage(argv[0]);
            return DEMO_BAD_COMMAND_FORMAT;
        }
    }
    else {
        printf("\nImproperly formatted command.\n");
        Usage(argv[0]);
        return DEMO_BAD_COMMAND_FORMAT;
    }

    if (result) {
        printf("Command failed with error 0x%x\n", result);
    }

    return result;

} // main
```

TekPls extension invocation

Three of the commands described in *SDK Reference Manual* support **ProLink/ProNet** extension services. These commands are:

- 0E 00, *Create Externsion*;
- 0E FF, *Delete Extension*; and
- 0E xx, *Extension Command Execution*.

These commands are provided by the TekPls LIB/DLL for programs local to the Profile. The **ProLink/ProNet** extension services include the additional commands that **ProLink** supports on behalf of additional services which wish to "piggy-back" on top of the **Prolink** or **ProNet** session that is established. For example, the pair of programs *tekpls.exe* and *tekplsex.exe* provide the PLS serial protocol commands which "piggy-back" on the **Prolink** or **ProNet** session that was separately established.

The *tekpls.exe* program

The *tekpls.exe* program is invoked by using the ProLink/ProNet Extension Services **Create Extension** command. Specify the string "tekpls" as the command data. The *tekpls.exe* program will invoke the *tekplsex.exe* program and wait for the Profile to initialize before returning the CMD2 value to be used to access the extension. Normally the CMD2 value will be one but could be other values as additional extensions become possible. After the program *tekplsex.exe* completes initialization the *tekpls.exe* program exits.

The *tekplsex.exe* program

The *tekplsex.exe* program processes the PLS serial protocol commands. The serial protocol commands are sent as data in a **ProLink/ProNet** extension services **Extension Command Execution** commands. The serial protocol reply always contains at least four bytes. The first byte is the transaction number. The second byte is the PLS serial protocol library opcode being replied to. The third and fourth bytes of the reply are the library error code for the command execution that is documented in the *plserror.h* file. The *tekplsex.exe* programs terminates after the **ProLink/ProNet** extension services receives a **Delete Extension** command with the CMD2 value specified in the data byte. The Profile should beep as the *tekplsex.exe* program begins and terminates execution if the hardware and software are operating correctly.

Connecting to the TekPls extension

Here's how to connect to the TekPls extension via **ProNet**:

```
ConnectHandle ch = LOCAL_CONNECTION;
BOOL remote = RemOpenConnection (ConnectEthernet, 0, "remote_name", &ch);
UINT major = PlsGetMajorVersion(ch);
UINT minor = PlsGetMinorVersion(ch);
```

Here's how to connect to the TekPls extension via **ProLink**:

```
Cmd1 = 0E;
Cmd2 = 00;
Arg1 = "tekpls" // the PLS server extension
Arg2 = 02; // per EXT_TYPE_MACHINE

11:44:48..818--> CMD:  02 0A 0E 00 74 65 6B 70 6C 73 00 02 5D
11:44:48..818--> RSP:  04 00 00
11:44:48..949--> CMD:  02 03 00 00 00 00
11:44:48..949--> RSP:  04 00 00
```

```
11:44:50..451--> CMD:  02 03 00 00 00 00
11:44:50..451--> RSP:  04 00 00
11:44:50..551--> CMD:  02 03 00 00 00 00
11:44:50..551--> RSP:  02 04 0E 00 00 7E 74
```

Reply shows use Cmd2 = 7E for the extension.

```
11:45:18..832--> CMD:  02 03 0E 7E 01 73
11:45:18..832--> RSP:  04 00 01
11:45:18..862--> CMD:  02 03 00 00 01 FF
11:45:18..862--> RSP:  04 00 01
11:45:18..962--> CMD:  02 03 00 00 01 FF
11:45:18..962--> RSP:  02 08 0E 7E 01 01 00 00 01 00 71
```

Reply shows major version number is one.

```
11:45:26..322--> CMD:  02 03 0E 7E 02 72
11:45:26..332--> RSP:  04 00 02
11:45:26..352--> CMD:  02 03 00 00 02 FE
11:45:26..362--> RSP:  02 08 0E 7E 02 02 00 00 01 00 6F
```

Reply shows minor version number is one.

Obtaining a library handle

To obtain a library handle via **ProNet** after the extension has initialized, do the following:

```
PlsLibraryHandle hLib =
  PlsOpenLibrary(ch, libraryName, session, major, minor);
  while ((hLib == NULL) && (GetLastError() ==  PLS_AS_INIT_INCOMPLETE))
    {
      Sleep(1000);
      hLib = PlsOpenLibrary(ch, "", "test", 0, 0);
    }
```

To obtain a library handle via **ProLink** after the extension has initialized, do the following:

```
11:46:05..969--> CMD:  02 0D 0E 7E 03 00 00 00 00 00 74 65 73 74 00 B1
11:46:05..979--> RSP:  04 00 03
11:46:05..999--> CMD:  02 03 00 00 03 FD
11:46:05..999--> RSP:  02 0A 0E 7E 03 03 48 00 00 00 00 00 26
```

Reply shows PLS DLL not initialized.

```
11:47:41..377--> CMD:  02 0D 0E 7E 03 00 00 00 00 00 74 65 73 74 00 B1
11:47:41..377--> RSP:  04 00 04
11:47:41..407--> CMD:  02 03 00 00 04 FC
11:47:41..417--> RSP:  02 0A 0E 7E 04 03 48 00 00 00 00 00 25
```

Reply shows PLS DLL not initialized.

```
11:51:25..058--> CMD:  02 0D 0E 7E 03 00 00 00 00 00 74 65 73 74 00 B1
11:51:25..058--> RSP:  04 00 05
11:51:25..088--> CMD:  02 03 00 00 05 FB
11:51:25..088--> RSP:  02 0A 0E 7E 05 03 00 00 00 00 01 00 6B
```

Reply shows library handle of 0X00010000.

Archiving a file

You can archive a file using one of two methods:

- wait in DLL for archive operation to complete; or
- obtain a transaction handle and monitor archive operation.

To archive a file via **ProNet**, do the following:

```
// create file handle for archive
PlsFileHandle hFile1 = PlsConnectFile(hLib, NULL,0, "test");
PlsFileHandle hFile2 = PlsConnectFile(hLib, NULL,0, "test");
```

```
// issue command and wait for archive to take place
// (not possible on serial link)
if (PlsArchiveClip(hFile1, 0, 0, hTransport, &fileStored, NULL))
  printf("  archived to partition %i of cartridge \"%s\" ",
         fileStored.partitionNumber, fileStored.cartLabel);
else
  printf("  archive failed with error of 0x%lX", GetLastError());

// issue command and get transaction to monitor command
PlsTransactionHandle hTransaction;
if (PlsArchiveClip(hFile2, 0, 0, hTransport, &fileStored, &hTransaction))
  printf("  archive is transaction 0x%0lX ", hTransaction);
else
  printf("  archive failed with error of 0x%lX", GetLastError());

// wait till command done or failed
PlsCmdStatus cmd;
while ((PlsGetStatusCommand(hLib, &cmd, hTransaction)) &&
  (cmd.status) && (!(cmd.status & PlsCmdStatError)))
Sleep(100);

// get command result
PlsEvent event;
if (PlsGetCommandEvent(hLib, hTransaction, &event))
  printf("  archived to partition %i of cartridge \"%s\" ",
         event.e.cmd.res.fileStored.partitionNumber,
         event.e.cmd.res.fileStored.cartLabel);
else
  printf("  archive failed with error of 0x%lX", GetLastError());
```

To archive a file via **ProLink**, do the following:

Obtain a file handle:

```
12:03:02..761--> CMD:  02 12 0E 7E 50 00 00 01 00 00 00 00 00
                       00 00 74 65 73 74 00 63
12:03:02..761--> RSP:  04 00 0D
12:03:02..781--> CMD:  02 03 00 00 0D F3
12:03:02..781--> RSP:  02 0A 0E 7E 0D 50 00 00 02 00 01 00 14
```

Reply shows hFile of 0x00010002. Start archive:

```
12:03:23..381--> CMD:  02 13 0E 7E 53 02 00 01 00 00 00 00 00
                       00 00 00 00 00 00 00 00 1E
12:03:23..381--> RSP:  04 00 0E
12:03:23..401--> CMD:  02 03 00 00 0E F2
12:03:23..401--> RSP:  04 00 0E
12:03:23..501--> CMD:  02 03 00 00 0E F2
12:03:23..511--> RSP:  04 00 0E
12:03:23..601--> CMD:  02 03 00 00 0E F2
12:03:23..601--> RSP:  02 0D 0E 7E 0E 53 00 00 03 00 00 00 00
                       00 00 10
```

Reply shows hTransaction of 0x00000003. Check transaction status:

```
12:03:42..198--> CMD:  02 0B 0E 7E 7D 00 00 01 00 03 00 00 00
                       F3
12:03:42..198--> RSP:  04 00 0F
12:03:42..228--> CMD:  02 03 00 00 0F F1
12:03:42..228--> RSP:  02 0B 0E 7E 0F 7D 00 00 0C 00 00 00 00 DC
12:03:50..380--> CMD:  02 0B 0E 7E 7D 00 00 01 00 03 00 00 00 F3
12:03:50..380--> RSP:  04 00 10
12:03:50..400--> CMD:  02 03 00 00 10 F0
12:03:50..400--> RSP:  02 0B 0E 7E 10 7D 00 00 0C 00 00 80 00 5B
```

Reply shows 0x800000C.

```
12:03:59..163--> CMD:  02 0B 0E 7E 7D 00 00 01 00 03 00 00 00 F3
12:03:59..173--> RSP:  04 00 11
12:03:59..193--> CMD:  02 03 00 00 11 EF
12:03:59..193--> RSP:  02 0B 0E 7E 11 7D 00 00 0C 00 00 00 00 DA
12:04:43..406--> CMD:  02 0B 0E 7E 7D 00 00 01 00 03 00 00 00 F3
12:04:43..416--> RSP:  04 00 12
12:04:43..436--> CMD:  02 03 00 00 12 EE
12:04:43..436--> RSP:  02 0B 0E 7E 12 7D 00 00 0C 00 00 00 42 97
```

Reply shows 0x0000000C. Check transaction status:

```
12:05:12..208--> CMD:  02 0B 0E 7E 7D 00 00 01 00 03 00 00 00 F3
12:05:12..218--> RSP:  04 00 13
12:05:12..238--> CMD:  02 03 00 00 13 ED
12:05:12..238--> RSP:  02 0B 0E 7E 13 7D 00 00 00 00 00 00 00 E4
```

Reply shows 0x00000000 (DONE). Check transaction result:

```
12:05:17..215--> CMD:  02 0B 0E 7E 74 00 00 01 00 03 00 00 00 FC
12:05:17..215--> RSP:  04 00 14
12:05:17..245--> CMD:  02 03 00 00 14 EC
12:05:17..245--> RSP:  02 18 0E 7E 14 74 00 00 53 03 00 00 00
                       00 00 01 00 30 30 30 30 30 31 32 35 00 0D
```

Reply shows file stored at partition 1 of cartridge “00000125”.

Closing the library and connection

To close the library and connection via **ProNet**, do the following:

```
if (!PlsCloseLibrary(hLib))
  printf("  CloseLibrary failed - GetLastError() = 0x%lX\n",
         GetLastError());

if (!RemCloseConnection(ch))
  printf("  CloseConnection failed - GetLastError() = 0x%lX\n",
         GetLastError());
```

Close the library.

```
12:05:46..267--> CMD:  02 07 0E 7E 04 00 00 01 00 6F
12:05:46..267--> RSP:  04 00 15
12:05:46..297--> CMD:  02 03 00 00 15 EB
12:05:46..297--> RSP:  02 06 0E 7E 15 04 00 00 5B
```

Reply shows command succeeded.

PLS constants

The following tables list PLS events and opcodes, sorted by name and value.

Table 9. PLS events by name

PlsEventCmdAllocateCartridge	0x32	PlsEventCmdRemoveTransport	0x7B
PlsEventCmdAllocateTransport	0x21	PlsEventCmdRenameFile	0x57
PlsEventCmdArchiveClip	0x53	PlsEventCmdRestore	0x56
PlsEventCmdArchiveDataFile	0x54	PlsEventCmdRestoreDataFile	0x5F
PlsEventCmdCleanTransport	0x26	PlsEventCmdUnloadTransport	0x25
PlsEventCmdCopyCartridge	0x3A	PlsEventCmdUpdateCartridge	0x37
PlsEventCmdDeleteFile	0x58	PlsEventDeleteAll	0xE3
PlsEventCmdExportCartridge	0x3C	PlsEventDeleteFile	0xE5
PlsEventCmdFormatCartridge	0x38	PlsEventExport	0xE1
PlsEventCmdGetClipSize	0x63	PlsEventFormat	0xE0
PlsEventCmdGetPartitionMap	0x43	PlsEventImport	0xE2
PlsEventCmdHouseKeeping	0x7F	PlsEventLast	0xFF.
PlsEventCmdImportCartridge	0x3D	PlsEventNewFile	0xE4
PlsEventCmdImportLoadCartridge	0x42	PlsEventNoEvent	0x00
PlsEventCmdInventoryCartridge	0x36	PlsEventRenameFile	0xE6
PlsEventCmdInventoryLibrary	0x7C	PlsEventTransOffline	0xE8
PlsEventCmdLoadTransport	0x24	PlsEventTransOnline	0xE7

Table 10. PLS events by value

0x00	PlsEventNoEvent	0x57	PlsEventCmdRenameFile
0x21	PlsEventCmdAllocateTransport	0x58	PlsEventCmdDeleteFile
0x24	PlsEventCmdLoadTransport	0x5F	PlsEventCmdRestoreDataFile
0x25	PlsEventCmdUnloadTransport	0x63	PlsEventCmdGetClipSize
0x26	PlsEventCmdCleanTransport	0x7B	PlsEventCmdRemoveTransport
0x32	PlsEventCmdAllocateCartridge	0x7C	PlsEventCmdInventoryLibrary
0x36	PlsEventCmdInventoryCartridge	0x7F	PlsEventCmdHouseKeeping
0x37	PlsEventCmdUpdateCartridge	0xE0	PlsEventFormat
0x38	PlsEventCmdFormatCartridge	0xE1	PlsEventExport
0x3A	PlsEventCmdCopyCartridge	0xE2	PlsEventImport
0x3C	PlsEventCmdExportCartridge	0xE3	PlsEventDeleteAll
0x3D	PlsEventCmdImportCartridge	0xE4	PlsEventNewFile
0x42	PlsEventCmdImportLoadCartridge	0xE5	PlsEventDeleteFile
0x43	PlsEventCmdGetPartitionMap	0xE6	PlsEventRenameFile
0x53	PlsEventCmdArchiveClip	0xE7	PlsEventTransOnline
0x54	PlsEventCmdArchiveDataFile	0xE8	PlsEventTransOffline
0x56	PlsEventCmdRestore	0xFF	PlsEventLast

Table 11. PLS opcodes by name

PlsOpCodeAddTransport	0x7A	PlsOpCodeGetCartridgeStatus	0x35
PlsOpCodeAllocateCartridge	0x32	PlsOpCodeGetClipSize	0x63
PlsOpCodeAllocateTransport	0x21	PlsOpCodeGetCommandEvent	0x74
PlsOpCodeArchiveClip	0x53	PlsOpCodeGetEventMask	0x71
PlsOpCodeArchiveDataFile	0x54	PlsOpCodeGetFileDescription	0x5B
PlsOpCodeBackupCatalog	0x62	PlsOpCodeGetLibraryConfig	0x05
PlsOpCodeCancelCommand	0x7E	PlsOpCodeGetLibraryStatus	0x06
PlsOpCodeCleanTransport	0x26	PlsOpCodeGetLocationString	0x3E
PlsOpCodeCloseBinInfo	0x0C	PlsOpCodeGetMajorVersion	0x01
PlsOpCodeCloseCartridge	0x33	PlsOpCodeGetMinorVersion	0x02
PlsOpCodeCloseCartridgeInfo	0x0F	PlsOpCodeGetModes	0x76
PlsOpCodeCloseFile	0x51	PlsOpCodeGetPartitionMap	0x43
PlsOpCodeCloseFileInfo	0x12	PlsOpCodeGetPath	0x78
PlsOpCodeCloseFindHandle	0x15	PlsOpCodeGetStatusCommand	0x7D
PlsOpCodeCloseLibrary	0x04	PlsOpCodeGetTimeDate	0x79
PlsOpCodeCloseTransport	0x22	PlsOpCodeGetTransportStatus	0x23
PlsOpCodeCloseTransportConfig	0x09	PlsOpCodeHouseKeeping	0x7F
PlsOpCodeConnectCartridge	0x30	PlsOpCodeImportCartridge	0x3D
PlsOpCodeConnectFile	0x50	PlsOpCodeImportLoadCartridge	0x42
PlsOpCodeConnectTransport	0x20	PlsOpCodeInventoryCartridge	0x36
PlsOpCodeCopyCartridge	0x3A	PlsOpCodeInventoryLibrary	0x7C
PlsOpCodeDeleteFile	0x58	PlsOpCodeLastCommand	0xFF
PlsOpCodeExportCartridge	0x3C	PlsOpCodeLoadTransport	0x24
PlsOpCodeFindFirstBinInfo	0x0A	PlsOpCodeNoOp	0x00
PlsOpCodeFindFirstCartridgeInfo	0x0D	PlsOpCodeOpenLibrary	0x03
PlsOpCodeFindFirstFileInfo	0x10	PlsOpCodeRemoveTransport	0x7B
PlsOpCodeFindFirstHandle	0x13	PlsOpCodeRenameFile	0x57
PlsOpCodeFindFirstTransportConfig	0x07	PlsOpCodeRestore	0x56
PlsOpCodeFindNextBinInfo	0x0B	PlsOpCodeRestoreDataFile	0x5F
PlsOpCodeFindNextCartridgeInfo	0x0E	PlsOpCodeSetBackupDir	0x60
PlsOpCodeFindNextFileInfo	0x11	PlsOpCodeSetCartDescription	0x40
PlsOpCodeFindNextHandle	0x14	PlsOpCodeSetEventMask	0x70
PlsOpCodeFindNextTransportConfig	0x08	PlsOpCodeSetFileDescription	0x5A
PlsOpCodeFormatCartridge	0x38	PlsOpCodeSetLocationString	0x3F
PlsOpCodeGetAnyEvent	0x72	PlsOpCodeSetModes	0x75
PlsOpCodeGetAsynchEvent	0x73	PlsOpCodeSetPath	0x77
PlsOpCodeGetBackupDir	0x61	PlsOpCodeUnloadTransport	0x25
PlsOpCodeGetCartDescription	0x41	PlsOpCodeUpdateCartridge	0x37
PlsOpCodeGetCartridgeConfig	0x34		

Table 12. PLS opcodes by value

0x00	PlsOpCodeNoOp
0x01	PlsOpCodeGetMajorVersion
0x02	PlsOpCodeGetMinorVersion
0x03	PlsOpCodeOpenLibrary
0x04	PlsOpCodeCloseLibrary
0x05	PlsOpCodeGetLibraryConfig
0x06	PlsOpCodeGetLibraryStatus
0x07	PlsOpCodeFindFirstTransportConfig
0x08	PlsOpCodeFindNextTransportConfig
0x09	PlsOpCodeCloseTransportConfig
0x0A	PlsOpCodeFindFirstBinInfo
0x0B	PlsOpCodeFindNextBinInfo
0x0C	PlsOpCodeCloseBinInfo
0x0D	PlsOpCodeFindFirstCartridgeInfo
0x0E	PlsOpCodeFindNextCartridgeInfo
0x0F	PlsOpCodeCloseCartridgeInfo
0x10	PlsOpCodeFindFirstFileInfo
0x11	PlsOpCodeFindNextFileInfo
0x12	PlsOpCodeCloseFileInfo
0x13	PlsOpCodeFindFirstHandle
0x14	PlsOpCodeFindNextHandle
0x15	PlsOpCodeCloseFindHandle
0x20	PlsOpCodeConnectTransport
0x21	PlsOpCodeAllocateTransport
0x22	PlsOpCodeCloseTransport
0x23	PlsOpCodeGetTransportStatus
0x24	PlsOpCodeLoadTransport
0x25	PlsOpCodeUnloadTransport
0x26	PlsOpCodeCleanTransport
0x30	PlsOpCodeConnectCartridge
0x32	PlsOpCodeAllocateCartridge
0x33	PlsOpCodeCloseCartridge
0x34	PlsOpCodeGetCartridgeConfig
0x35	PlsOpCodeGetCartridgeStatus
0x36	PlsOpCodeInventoryCartridge
0x37	PlsOpCodeUpdateCartridge
0x38	PlsOpCodeFormatCartridge
0x3A	PlsOpCodeCopyCartridge
0x3C	PlsOpCodeExportCartridge
0x3D	PlsOpCodeImportCartridge
0x3E	PlsOpCodeGetLocationString
0x3F	PlsOpCodeSetLocationString
0x40	PlsOpCodeSetCartDescription
0x41	PlsOpCodeGetCartDescription
0x42	PlsOpCodeImportLoadCartridge
0x43	PlsOpCodeGetPartitionMap
0x50	PlsOpCodeConnectFile
0x51	PlsOpCodeCloseFile
0x53	PlsOpCodeArchiveClip
0x54	PlsOpCodeArchiveDataFile
0x56	PlsOpCodeRestore
0x57	PlsOpCodeRenameFile
0x58	PlsOpCodeDeleteFile
0x5A	PlsOpCodeSetFileDescription
0x5B	PlsOpCodeGetFileDescription
0x5F	PlsOpCodeRestoreDataFile
0x60	PlsOpCodeSetBackupDir
0x61	PlsOpCodeGetBackupDir
0x62	PlsOpCodeBackupCatalog
0x63	PlsOpCodeGetClipSize
0x70	PlsOpCodeSetEventMask
0x71	PlsOpCodeGetEventMask
0x72	PlsOpCodeGetAnyEvent
0x73	PlsOpCodeGetAsynchEvent
0x74	PlsOpCodeGetCommandEvent
0x75	PlsOpCodeSetModes
0x76	PlsOpCodeGetModes
0x77	PlsOpCodeSetPath
0x78	PlsOpCodeGetPath
0x79	PlsOpCodeGetTimeDate
0x7A	PlsOpCodeAddTransport
0x7B	PlsOpCodeRemoveTransport
0x7C	PlsOpCodeInventoryLibrary
0x7D	PlsOpCodeGetStatusCommand
0x7E	PlsOpCodeCancelCommand
0x7F	PlsOpCodeHouseKeeping
0xFF	PlsOpCodeLastCommand

PLS error codes by value

Archive server protocol errors returned to an application are of the form `0x20009<code>` where `code` is a 12-bit number from the table below. Error codes in serial protocols are a 12-bit code in a 16-bit field. The high-order 4 bits of the serial protocol error code are zeros.

Table 13. PLS serial error codes

Code	Error	Description
0x000	PLS_NO_ERROR	No error/success.
0x001	PLS_AS_INIT_FAIL	PLS classes initialization failed. One of the PLS classes failed to instantiate. (Call Grass Valley Group Customer Support.)
0x002	PLS_AS_LC_INIT_FAIL	Local catalog initialization failure. Local catalog failed to load in memory. (Call Grass Valley Group Customer Support.)
0x003	PLS_AS_RES_INIT_FAIL	OBSOLETE.
0x004	PLS_AS_TP_INIT_FAIL	Transport class initialization failure. Unable to instantiate a transport class or was not able to open the local catalog file, *C:\pls_lc\plstrans.lct*. (Call Grass Valley Group Customer Support.)
0x005	PLS_AS_LIB_INIT_FAIL	Library class initialization failure. Unable to instantiate robot class, initialize robot, or get local catalog data. (Call Grass Valley Group Customer Support.)
0x006	PLS_AS_INIT_FILE_MISSING	OBSOLETE.
0x007	PLS_AS_INIT_RECORD_FAIL	OBSOLETE.
0x008	PLS_AS_CONOP_RESOURCE_ERROR	Unable to allocate/deallocate desired resource (cartridge/transport). The desired resource is either in use or has been reserved, and queuing mode for that resource is not set. (See *PlsSetModes* on page 260 of the *SDK Reference Manual*.)
0x009	PLS_AS_LC_DELETE_FAIL	Unable to delete local catalog record. A delete was attempted on a local catalog record that no longer exists.
0x00A	PLS_AS_HND_DELETE_FAIL	OBSOLETE.
0x00B	PLS_AS_PATH_ERROR	Invalid PDR movie (clip) path. Make sure that the correct path is set (see *PlsGetPath* on page 240 and *PlsSetPath* on page 261 of the *SDK Reference Manual*) and that the desired movie exists within that path.
0x00C	PLS_AS_GET_MOVIE_ERROR	Unable to open desired movie (clip). Make sure that the movie (clip) exists and has not been opened "exclusive" (see *PdrOpenMovie* on page 162 of the *SDK Reference Manual*) by another application.
0x00D	PLS_AS_INSUFFICIENT_CAPACITY	Insufficient tape capacity to archive selected movie (clip). The target cartridge/partition does not have enough free space to accommodate the selected movie (clip). Try a different cartridge/partition pairing.
0x00E	PLS_AS_DATA_RETRIVAL_ERROR	Unable to retrieve desired local catalog record. The desired local catalog record no longer exists.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x00F	PLS_AS_I960_HANDLE_ERROR	OBSOLETE.
0x010	PLS_AS_INVALID_RECORD_ERROR	Desired local catalog record does not exist.
0x011	PLS_AS_MOVIE_CREATE_ERROR	Unable to create desired movie (clip) for restoring.
0x012	PLS_AS_CLOSE_MOVIE_ERROR	Unable to close archived/restored movie (clip). (See *PdrCloseMovie* on page 82 of the *SDK Reference Manual*.)
0x013	PLS_AS_FILE_DELETED_ERROR	Desired tape file is marked as deleted.
0x014	PLS_AS_HND_CREATE_FAIL	Unable to create handle. Generally due to a bad parameter. (Example: attempting to create a cartridge handle to a non-existent cartridge.)
0x015	PLS_AS_RECORD_CREATE_FAIL	One of the PLS log files could not be opened/created. (Call Grass Valley Group Customer Support.)
0x016	PLS_AS_LOG_INIT_FAIL	Unable to create a local catalog record. Generally this is caused when the memory needed for the new record is unavailable. The only exception is file records which will fail when a bad parameter is detected (cartridge label, partition number).
0x017	PLS_AS_LC_UPDATE_FAIL	Unable to update a local catalog record to the disk file. RAM copy of the record is still valid. However, once the PLS is restarted, that record's data may be invalid.
0x018	PLS_AS_UNFORMAT_FAIL	Unable to unformat a cartridge. The cartridge still contains old files and is in an unknown state.
0x019	PLS_AS_CARTRIDGE_MOUNTED	Unable to unmount cartridge from target transport. The command attempted to remove the cartridge from the target transport to clear it for allocation by a new transaction.
0x01A	PLS_AS_HANDLE_MISSING	A handle necessary for processing of this command is missing or invalid.
0x01B	PLS_AS_TRANSPORT_INUSE	Target transport is being used by another transaction. This will occur if a command tries to do an unload or import load to a busy transport. Also attempting to close a transport handle used to reserve a transport which is busy will cause this error.
0x01C	PLS_AS_TRANS_SLOT_OUT_OF_RANGE	Transport device number (IPM address) or robot location number is out of range.
0x01D	PLS_AS_UNFORMATTED	Cartridge must be formatted to perform this command.
0x01E	PLS_AS_INVALID_ACTION_CODE	The action code given for this command is invalid (out of range).
0x01F	PLS_AS_FILE_ALREADY_ARCHIVED	A file by the same name already exists in target partition.
0x020	PLS_AS_CARTRIDGE_NOT_SPECIFIED	Cartridge handle/label must be specified for this command.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x021	PLS_AS_EMPTY_TRANSPORT_ERROR	An command which expected a mounted cartridge found the transport empty. This error can only occur with the **PlsUnloadTransport** command or when commands such as **PlsArchiveClip** are sent to stand-alone transports.
0x022	PLS_AS_MISSING_LABEL	Cartridge label required for this command was missing.
0x023	PLS_AS_MISSING_FILENAME	File name required for this command was missing.
0x024	PLS_AS_FILE_NOT_ARCHIVED	Occurs when attempting to restore or rename a tape file, and the tape file has not been archived.
0x025	PLS_AS_WRONG_CARTRIDGE_TYPE	Cartridge is of the wrong type for the specified command. (Example: a media cartridge is specified in a **PlsCleanTransport** command.)
0x026	PLS_AS_ZERO_LENGTH_MOVIE	Movie (clip) has zero length. The movie (clip) exists, but has not recorded media. Can only occur during a **PlsArchiveClip** command.
0x027	PLS_AS_CARTRIDGE_OUT_OF_LIBRARY	The target cartridge is not in the robot library, or mounted in a stand-alone transport.
0x028	PLS_AS_COMMAND_NOT_IMPLEMENTED	This API command is not yet implemented.
0x029	PLS_AS_NO_FREE_TRANSPORT_SLOT	There is no free IPM slot (address) to add a new transport. There is currently a maximum of 4 archive IPM ports.
0x02A	PLS_AS_TRANSPORT_SLOT_IN_USE	The desired IPM slot (address) is in use.
0x02B	PLS_AS_TRANSPORT_NOT_CONNECTED	The transport requested does not exist.
0x02C	PLS_AS_EXCEEDED_MAX_NUM_HANDLES	The maximum number of handles for this handle type (library, resource, etc.) is in use. Close a handle of the desired type to proceed.
0x02D	PLS_AS_RENAME_SAME_NAME	Trying to rename a file with the same name.
0x02E	PLS_AS_HANDLE_INUSE	This handle is currently being used by another command. (Generally occurs when attempting to close a handle.)
0x02F	PLS_AS_INVALID_PARTITION	The partition number associated with the file is invalid, or the partition number is out of range. (The range a partition number can take is vendor-dependent.)
0x030	PLS_AS_MOVIE_ALREADY_EXISTS	The command is attempting a restore operation for an archived movie (clip) to a target path which already contains a movie (clip) of the same name. Renaming the tape file will solve this problem.
0x031	PLS_AS_BAD_IN_OUT_POINTS	The in/out points specified for this movie (clip) are invalid. The in point may be greater than the out point, or the in/out points are outside the recorded media.
0x032	PLS_AS_INVALID_SYSTEM_HANDLE	The primary handle for a given operation was invalid. The primary handle is the one required handle for the given command (generally a library handle).

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x033	PLS_AS_LOOP_COMPLETE	First/next search loop has completed.
0x034	PLS_AS_INVALID_LOOP_HANDLE	Invalid first/next search loop handle.
0x035	PLS_AS_NT_FILE_READ_ERROR	There was an error while attempting a Win32 read operation.
0x036	PLS_AS_NT_FILE_WRITE_ERROR	There was an error while attempting a Win32 write operation.
0x037	PLS_AS_NT_FILE_ALREADY_EXISTS	The command is attempting to restore a Windows NT file to a location when a file of that name already exists. Renaming the tape file is one way to solve this problem
0x038	PLS_AS_INVALID_NT_HANDLE	Windows NT returned a handle creation error.
0x039	PLS_AS_CART_TABLE_BACKUP_FAILED	Local catalog cartridge table backup failed. The local catalog table is still intact, and normal operation can proceed. The previous instance (if any) of a backup file still exists.
0x03A	PLS_AS_FILE_TABLE_BACKUP_FAILED	Local catalog file table backup failed. Local catalog cartridge table backup failed. The local catalog table is still intact, and normal operation can proceed. The previous instance (if any) of a backup file still exists.
0x03B	PLS_AS_PART_TABLE_BACKUP_FAILED	Local catalog partition table backup failed. Local catalog cartridge table backup failed. The local catalog table is still intact, and normal operation can proceed. The previous instance (if any) of a backup file still exists.
0x03C	PLS_AS_BACKUP_DIR_OPEN_FAILED	Unable to open remote target for local catalog backup. This is generally due to an invalid share name.
0x03D	PLS_AS_CARTRIDGE_ALLOC_ERROR	Unable to allocate desired cartridge. If the cartridge is not specified, then the command could find no free cartridge. This error only occurs when cartridge queuing is off. (See *PlsSetModes* on page 260 of the *SDK Reference Manual*.)
0x03E	PLS_AS_CARTRIDGE_DEALLOC_ERROR	Unable to deallocate a given cartridge. If this occurs, the cartridge could be unusable until the PLS is re-initialized.
0x03F	PLS_AS_TRANSPORT_ALLOC_ERROR	Unable to allocate desired transport. If the transport is not specified, then the operation could find no free transport. This error only occurs when transport queuing is off. (See *PlsSetModes* on page 260 of the *SDK Reference Manual*.)
0x040	PLS_AS_TRANSPORT_DEALLOC_ERROR	Unable to deallocate a given transport. If this occurs, the transport could be unusable until the PLS is re-initialized.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x041	PLS_AS_FILE_NOT_UNIQUE	File specified for restore is not unique. This occurs when a file handle does not specify the cartridge and/or the partition. The system is asked to find the correct file. If more than one file of that name exists based on the contents of the file handle. (i.e. Was cartridge label and/or partition specified?), this error will occur. Creating a file handle with the correct cartridge and partition and using it for the command will solve this problem.
0x042	PLS_AS_SOURCE_CARTRIDGE_EMPTY	Source cartridge for a Copy Cartridge command is empty. (See *PlsCopyCartridge* on page 207 of the *SDK Reference Manual*.)
0x043	PLS_AS_TARGET_CART_NOT_EMPTY	Target cartridge for a Copy Cartridge command is not empty. (See *PlsCopyCartridge* on page 207 of the *SDK Reference Manual*.)
0x044	PLS_AS_CART_FORMAT_MISMATCH	Target cartridge for a Copy Cartridge command is not the same format as the source cartridge. (See *PlsCopyCartridge* on page 207 of the *SDK Reference Manual*.)
0x045	PLS_AS_TARGET_CART_INUSE	Target cartridge for a Copy Cartridge command is currently in use. This causes a copy cartridge to fail, since the state of the target (destination) cartridge is changing.
0x046	PLS_AS_CANCEL_FAILED	Unable to cancel selected command. The command is either non-cancelable, or is in a state where canceling is not allowed.
0x047	PLS_AS_INVALID_CONNECTION	Application is not connected to the PLS library.
0x048	PLS_AS_INIT_INCOMPLETE	PLS library is still initializing. This happens when a PlsOpenLibrary is called before PLS initialization is complete. Check periodically once PLS initialization is complete. Either a library handle or different error will be returned.
0x049	PLS_AS_WRONG_BACKUP_DIR	Local device name points to wrong remote directory. The drive letter associated with backup command no longer points to the set share path. (See *PlsGetBackupDir* on page 225 and *PlsSetBackupDir* on page 255 of the *SDK Reference Manual*.)
0x04A	PLS_AS_CARTRIDGE_IN_LIBRARY	Occurs while attempting to import a cartridge with a label the same as one already physically in the library.
0x04B	PLS_AS_MEMORY_ALLOCATION_ERROR	Unable to allocate memory needed for the current command. This usually happens when the system is low on memory. Shutting down any un-needed processes can clear this up.
0x04C	PLS_AS_INVALID_DEVICE_ADDRESS	SCSI device address specified is not the one associated with the specified transport. Transport SCSI addresses are printed out in the PLS log (*pls.log*) during PLS initialization.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x04D	PLS_AS_THREAD_RESUME_FAILED	Unable to resume thread after suspension. The command belonging to this thread is lost. The command can be re-issued.
0x04E	PLS_AS_INVALID_MOVIE_TYPE	Movie (clip) is of a type that is invalid for this operation. This will occur if an attempt is made to archive movie type other than “simple”. (See *PdrGetMovieAttributes* on page 128 of the *SDK Reference Manual*.)
0x04F	PLS_AS_INVALID_TRANSPORT_NUMBER	Transport number was not in the range of 0-255.
0x050	PLS_AS_LOCAL_CATALOG_PURGE_FAILED	Unable to purge the local catalog.
0x101	PLS_AR_CONNECTION_FAILED	Internal state during initialization. Unable to locate SCSI library robot. (Call Grass Valley Group Customer Support.)
0x102	PLS_AR_INITIALIZATION_FAILED	Internal state during initialization. Unable to initialize library robot. (Call Grass Valley Group Customer Support.)
0x103	PLS_AR_INVENTORY_FAILED	Library robot unable to scan barcodes. (Hardware error. Call Grass Valley Group Customer Support.)
0x104	PLS_AR_INVALID_BIN	Requested bin does not exist in library robot. Attempting to find contents of a bin not available in currently connected robot.
0x105	PLS_AR_BARCODE_NOT_FOUND	Requested cartridge is not in the library robot or the cartridge label is not readable. Attempting to find location of a cartridge not currently in the robot. Use library inventory to update PLS knowledge of current robot state.
0x107	PLS_AR_DESTINATION_CONFLICT	Library robot bin or transport already occupied by another cartridge. Or, attempted to import another cartridge into a full library robot.
0x108	PLS_AR_SOURCE_CONFLICT	Requested cartridge not found in expected location. May occur if tape cartridge did not properly eject from the tape transport. Use library inventory to update PLS knowledge of current robot state.
0x109	PLS_AR_NOT_ADDED	Requested cartridge was not imported into the library robot. The operation timed out without the cartridge being added. Repeat the import operation.
0x10A	PLS_AR_NOT_REMOVED	Requested cartridge was not exported from the library robot. The operation timed out without the cartridge being removed. Repeat the export operation.
0x10C	PLS_AR_NOT_REQUESTED_BARCODE	Imported cartridge was not the cartridge requested. Operator has inserted the wrong cartridge. Request operator to insert the correct cartridge.
0x10E	PLS_AR_NO_FREE_ENTRY_PORTS	Cartridge import attempted with all entry ports occupied. Wait for previous import operation to complete and repeat the import operation.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x10F	PLS_AR_NO_FREE_EXIT_PORTS	Cartridge export attempted with all exit ports occupied. Wait for previous export operation to complete and repeat the export operation.
0x110	PLS_AR_CONNECTION_UNDERWAY	Internal state during initialization. Robot status while connecting to the SCSI library robot. (Call Grass Valley Group Customer Support.)
0x111	PLS_AR_INITIALIZATION_NEEDED	Internal state during initialization. Robot status before initializing the SCSI library robot. (Call Grass Valley Group Customer Support.)
0x112	PLS_AR_INITIALIZATION_UNDERWAY	Internal state during initialization. Robot status while initializing the SCSI library robot. (Call Grass Valley Group Customer Support.)
0x113	PLS_AR_NEW_CARTRIDGE	Operator error. Cartridge imported into library robot during cartridge export operation. Use library inventory to find label of cartridge.
0x114	PLS_AR_NO_ENTRY_PORTS	Cartridge import attempted with no entry ports installed in robot.
0x115	PLS_AR_NO_EXIT_PORTS	Cartridge export attempted with no exit ports installed in robot.
0x201	PLS_AT_INTERNAL	The real-time processor has detected an unexpected error which it does not know how to handle. Examine *profile.log*.
0x202	PLS_AT_BAD_HANDLE	The real-time processor was given a bad transaction handle by NT.
0x203	PLS_AT_NOT_IMPLEMENTED	A SCSI device returned an ILLEGAL REQUEST error.
0x205	PLS_AT_NBYTES_OUT_OF_RANGE	An fread or fwrite request from the Windows NT processor to the real-time processor had an illegal byte count argument.
0x206	PLS_AT_BAD_DEVICE_CODE	The real-time processor was given a request for an operation on a non-existent tape drive.
0x207	PLS_AT_BAD_INOUT_POINTS	An in-point specification was greater than the corresponding out-point.
0x208	PLS_AT_BAD_PARTITION	A request was made to position a tape to a non-existent partition.
0x209	PLS_AT_IPM_ERROR	An error occurred on the communication channel between the real-time and OS processors.
0x20A	PLS_AT_EIO	An unrecoverable I/O error occurred on a tape drive.
0x20B	PLS_AT_WRITEPROT	An attempt was made to write to a physically write-protected tape.
0x20C	PLS_AT_NOT_READY	The tape drive is not ready.
0x20D	PLS_AT_BAD_PARAMETER	A SCSI device reported an illegal parameter.
0x20E	PLS_AT_BLANK_CHECK	An attempt was made to read a portion of tape which has not been written, i.e. beyond end of data.
0x20F	PLS_AT_VOLUME_OVERFLOW	An attempt was made to write beyond the end of a partition.

Table 13. PLS serial error codes (Continued)

Code	Error	Description
0x210	PLS_AT_MISC_SCSI_ERROR	An unexpected error in the SCSI driver has occurred. See *profile.log*.
0x211	PLS_AT_NO_MEDIUM	There is no tape in the tape drive.
0x212	PLS_AT_UNKNOWN_FORMAT	The tape is of unknown format or has never been formatted.
0x2FE	PLS_AT_IPM_WRITE_FAIL	An error occurred writing to the communication channel between the the real-time and OS processors.
0x2FF	PLS_AT_IPM_READ_FAIL	An error occurred reading from the communication channel between the the real-time and OS processors.
0x301	PLS_REMOTE_DLL_CMD_NOT_AVAILABLE	PLS serial protocol does not support the PLS operation requested. Protocol may be expanded in a future release to include the command.
0x302	PLS_REMOTE_CMD_INVALID	PLS server does not support the opcode received. Command may be supported in a future release.
0x303	PLS_REMOTE_CMD_FAILURE	PLS server command failed without indicating a specific PLS error code.
0x304	PLS_REMOTE_CMD_INPUT_FAILURE	PLS server failed to receive all required information in the command packet.
0x305	PLS_REMOTE_CMD_OUTPUT_FAILURE	PLS server was not able to place all requested items in the result packet.
0x311	PLS_LOCAL_SERVER_FAILURE	Communication with the local PLS server has failed. Restart the local server.

Chapter 8

Programming with MPEG

Since limited bandwidth has always been a concern for those interested in sending video across a network, the engineers of the Motion Picture Experts Group designed the MPEG algorithm with a strong emphasis on compression. As a result, MPEG can double video storage capacity over JPEG-only storage at similar quality levels. Due to the smaller sizes, MPEG can also enable much faster data transfers over Fibre Channel.

Depending on the image, JPEG can achieve compression ratios of anywhere from 15:1 to 30:1. MPEG can achieve compression ratios that approach 200:1.

MPEG manages this by not duplicating video that remains static from one frame to the next. Since each frame in a sequence of motion picture frames is often very similar to the preceding and succeeding frames, MPEG only saves the changed information from the previous frame to reconstruct the present frame.

This use of backward and forward motion prediction involves GOPs (groups of pictures) consisting of I-frames, P-frames, and B-frames. An I-frame (also known as an I-picture or *intracoded* picture) is analogous to a single motion-JPEG frame, where all the data required to display the frame is stored in one picture. An I-frame is the heart of a GOP, the one picture on which the other pictures in the GOP base their predictive calculations. A P-frame (also called a P-picture or *predictive* picture) uses a motion vector to predict what will happen in the next frame and contains only the changed data rather than the entire frame of video. A B-frame (also known as a B-picture or *bidirectional* picture) relies on data from both backward and forward motion vectors to determine how a future frame will be composed.

A GOP consists of one I-frame and any number of P-frames and B-frames. In general, a longer GOP (with many P-frames and B-frames) will yield a more efficient MPEG video stream, at the possible expense of video quality; a shorter GOP (with fewer P-frames and B-frames separating the I-frames) will yield a less efficient video stream while improving video quality.

Compression/decompression algorithms

An algorithm for compressing and decompressing images is called a *codec*. The primary purpose of a codec is to reduce the number of bits required to represent an image in a digital, networked environment. Codecs are classified as *lossless* and *lossy.*

With *lossless* compression, no image information is lost, and reconstruction of the compressed image is identical to the original image. Lossless codecs are often used to compress sensitive medical images or scientific images that contain extremely important details. CCITT Group 3 and Group 4 are examples of lossless codecs.

With *lossy* compression, as the name implies, some of the information is lost, and reconstruction of a compressed image will not be identical to the original image. In most cases, however, the visual information lost is not noticeable. Both JPEG and MPEG are lossy codecs.

Some limitations to MPEG

MPEG-2 was designed for one-pass recording and one-pass playback starting with the first recorded pictures. In general, this is not good for editing, particularly linear insert editing. Because of this, linear editing is restricted to streams that have been recorded with GOPs that are I-frame only.

Other considerations:

- Off-speed playback of media stored in MPEG format will not produce the same quality and smoothness of motion produced by JPEG codecs. It is intended only as a way to visually locate material in a clip, and is not for on-air applications.
- Field dominance for field 1 only is supported at this time in MPEG. Field dominance for field 2 or variable dominance for fields 1 or 2 is not supported.
- Recording at a rate other than normal speed (1.0) is not allowed with MPEG.
- Jog recording with **VdrJog** is not allowed with MPEG.
- Loop recording only works properly if the loop length is a multiple of the GOP length, and the GOP encoding is closed-end. Otherwise, there will be a discontinuity at the beginning of the stream.

Other MPEG notes

A Profile MPEG encoder board transforms, quantizes, and encodes CCIR-601 video to MPEG-2 bitstream files for storage on disk. The MPEG decoder board, on the other hand, decodes, inverse-quantizes, and transforms MPEG-2 bitstream files stored on disk to CCIR-601 video. Input formats can be serial digital component, analog composite, or analog component. Output is either serial digital component or analog composite. The shortest MPEG clip that the Profile system can play is four GOPs or two seconds long, whichever is greater.

The PDR200 with an MPEG upgrade and the PDR300 from the factory both use the MPEG-2 4:2:2 @ Main Level encoder and decoder boards. The MPEG encoder offers both 4:2:2 and 4:2:0 chroma sampling, variable bitrates, and GOP structures from I-frame only to 16-frame GOPs.

Using MPEG functions

The TekVdr library has been extended with MPEG functionality. See the companion volume to this manual, *Profile SDK Reference Manual,* for a complete description of these functions, including parameters and parameter types. Also, see ***Chapter ,*** for instructions on general housekeeping for a Profile program, such as getting a resource handle, opening a port, and allocating resources.

Archiving and streaming

Profile supports MPEG archiving, which, from the applications standpoint, works just as it does for JPEG. Profile also supports streaming to/from the archive device for both MPEG and JPEG. This lets you start to play a file before it is recovered from the tape device. In this repsect, it is similar to the streaming capability with Fibre Channel.

PDR300s can stream MPEG and JPEG files to/from the PLS20, PLS200, and Ampex drives, in addition to the SGI server.

Bitrate

The bitrate, expressed in megabits per second (Mbps), sets the video quality. The higher the bitrate, the higher the video quality. However, higher bitrates require more disk space to store the data, limiting the number of hours of material you can store on disk. Use **VdrSetBitRate** to set the bitrate in the range of 4 to 45Mbps and **VdrGetBitRate** to query the system for the current bitrate.

Chrominance sampling

One early step in programming MPEG is setting the chrominance sampling (chroma format) to either 4:2:0 or 4:2:2. Call **VdrSetMpegChromaFormat** to set chroma sampling. **VdrGetMpegChromaFormat** queries the system for the current format, returning an enumerator of type MpegChromaFmt, such as MpegChroma422.

First and last line of encoding

You can select which of the incoming lines of video are encoded as MPEG. **VdrSetEncodingRange** allows you to set the first and last encoded line of video. For 525 (NTSC) systems, the starting and ending lines must be in the range 21 through 260, with an acceptable total of 512 or fewer lines per frame. For 625 (PAL), the range is 7 through 310, with an acceptable total of 608 lines per frame. **VdrGetEncodingRange** returns the first and last lines as they are currently set.

GOP structure

Another early step is to set the GOP structure, in other words, the number of P- and B-frames you want to use (up to sixteen total, minus one for the single I-frame that is part of each GOP). Use **VdrSetMpegGopStructure** to set the structure and **VdrGetMpegGopStructure** to query the system for the current GOP. These functions use an enumerator of type MpegGopEnd that determines how an MPEG video stream begins, either open (GopOpenEnd) or closed (GopClosedEnd). In an open GOP, the initial B pictures have a preceding I-frame that is part of the previous GOP. A closed GOP, on the other hand, has initial B-frames that have a preceding I-frame that is part of the same GOP.

MPEG closed caption technique

MPEG-2 compression works on 16-by-16 macro-blocks of pixels. If closed caption information is contained in the same macro-block as active video, the two will be compressed together, which will degrade the closed captions. By ensuring that only black is compressed with the closed caption information, you can minimize the compression degradation. Since closed caption information is placed on line 21 in 525-line video, you should follow these guidelines when recording video with closed caption information in MPEG format to help ensure optimum closed caption integrity, even at lower bitrates.

1. Set the MPEG encoder to begin encoding at line 6, and end at line 261.
2. Where possible, ensure that lines 6 to 20 contain black.

This technique will allow you to minimize the degradation of closed caption information at lower bitrates. You may also improve your results by using 4:2:0 encoding at very low bitrates.

Picture information

VdrGetCurrentPictureStatus returns the size, coding, and structure of the picture at the current position.

Sample program: MPEG encoding/decoding

Example 16, mpegdemo.c demonstrates the use of MPEG encoding and decoding. A specified number of seconds of audio and video is recorded in MPEG format to the given filename, then the recorded clip (simple movie) is played back. It is very similar to the JPEG record examples in ***Chapter ,*** , so you may want to become familiar with those first.

In this example, the first available MPEG encoder and first available MPEG decoder are used. Unlike JPEG, which uses a single codec for both recording and playback, separate resources must be allocated for MPEG encoding and decoding. The chroma format, GOP structure, and bitrate of the encoding are set up after the encoder is allocated. See the *Profile SDK Reference Manual* for further information about the parameters for the functions and how to vary them for the needs of your specific application.

As with the JPEG examples, the first inputs and outputs on the Profile are used for video and audio. Recording and playback are virtually the same once the resources have been set up correctly for MPEG recording. Cleanup is also similar except there is one more resource to free than in the JPEG example.

Example 16. mpegdemo.c

```
//
// This sample program records an MPEG clip of a specified time
// (measured in seconds) and then plays it back.
// Use with Profile release 2.4.2.3 or later.
//
// Usage: MpegDemo movieName -s recordTimeSeconds
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//

#include <stdio.h>
#include <windows.h>
#include <limits.h>
#include <tekrem.h>
#include <tekcfg.h>
#include <tekpdr.h>
#include <tekvdr.h>s

// Number of audio channels to use:
#define NUM_AUDIO_CHANNELS      2

#define NUM_INDICES(array)  (sizeof(array)/sizeof(*(array)))

// Handles to encoders, decoders, and codecs:
static ResourceHandle mpegEncoder = NULL;
static ResourceHandle mpegDecoder = NULL;
static ResourceHandle audioCodecs[NUM_AUDIO_CHANNELS] = {NULL};
// Table of all recorder resources:
static ResourceHandle recorders[2 + NUM_INDICES(audioCodecs)] = {NULL};

// Handles to inputs and outputs:
static ResourceHandle videoInput = NULL;
static ResourceHandle videoOutput = NULL;
static ResourceHandle audioOutputs[NUM_INDICES(audioCodecs)] = {NULL};

// Handle to the connection for the machine:
static ConnectHandle connectHandle = NULL;
```

```
// Handle to the port that controls the timeline:
static VdrHandle vdrHandle = NULL;

//
// Print out usage line.
//
void Usage(const char* progName)
{
    printf("Usage: %s movieName -s durationInSeconds\n", progName);

}

//
// Fxn: allocate any available resource of the given type
//
ResourceHandle AllocateAny(ConnectHandle connectHandle, ResourceType resourceType)
{
    ResourceHandle resourceHandle;
    UINT resourceNumber;
    UINT numResources;

    numResources = CfgGetNumResources(connectHandle, resourceType);
    for (resourceNumber = 0; resourceNumber < numResources; ++resourceNumber) {
        resourceHandle = VdrAllocateResource(vdrHandle, resourceType,
                resourceNumber);
        if (resourceHandle != NULL) {
            // This resource is available:
            return(resourceHandle);
        }
    }
    return(NULL);
}

// Resource number for first audio input and output (remaining audio
// channels are sequential):
#define FIRST_AUDIO_INPUT   0
#define FIRST_AUDIO_OUTPUT  0

// Number of video input which should be recorded:
#define VIDEO_INPUT_NUM     0

// Number of video output at which playback should be seen:
#define VIDEO_OUTPUT_NUM    0

//
// Initialize resources in order to be able to perform record and
// playback.  Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL ConfigureResources(void)
{
    UINT index;
    EventHandle eventHandle;
    ResourceHandle* recorderPtr;

    // Open the connection (can be used to open connection to a remote
    // machine):
    if (! RemOpenConnection(ConnectLocal, 0, 0, &connectHandle)) {
        printf("Cannot open local connection.\n");
        return(FALSE);
    }
```

```
// Open the port which controls the timeline:
    vdrHandle = VdrOpenPortConnection(connectHandle);
    if (vdrHandle == NULL) {
        printf("Cannot open port.\n");
        return(FALSE);
    }

    // Used to add codecs, encoders, and decoders to list of all record
    // resources:
    recorderPtr= recorders;

    // Allocate the first available MPEG encoder:
    mpegEncoder = AllocateAny(connectHandle, ResourceMpegEncoder);
    if (mpegEncoder == NULL) {
        printf("Cannot allocate MPEG encoder.\n");
        return(FALSE);
    }
    *(recorderPtr++) = mpegEncoder;

    // Set the chroma format used to encode video:
    if (! VdrSetMpegChromaFormat(mpegEncoder, MpegChroma422)) {
        printf("Cannot set the chroma format of encoder.\n");
        return(FALSE);
    }

    // Set the group-of-pictures structure of the encoder; the values shown
    // (5 P pictures per GOP, 2 B pictures per anchor picture, closed-end GOPs,
    // and frame-based compression) are typical.  The value for the third
    // argument (1) ensures future compatibility:
    if (! VdrSetMpegGopStructure(mpegEncoder, GopClosedEnd, 1, 5, 2,
        PixStructureFrame)) {
        printf("Cannot set GOP structure of encoder.\n");
        return(FALSE);
    }

    // Set the bit-rate to be used for encoding (36 Mbps):
    if (! VdrSetBitRate(mpegEncoder, 36.0e6)) {
        printf("Cannot set bit-rate of encoder.\n");
        return(FALSE);
    }

    // Allocate the first available MPEG decoder:
    mpegDecoder = AllocateAny(connectHandle, ResourceMpegDecoder);
    if (mpegDecoder == NULL) {
        printf("Cannot allocate MPEG decoder.\n");
        return(FALSE);
    }
    *(recorderPtr++) = mpegDecoder;


// Allocate audio codecs:
    for (index = 0; index < NUM_INDICES(audioCodecs); ++index) {
        audioCodecs[index] = VdrAllocateResource(vdrHandle, ResourceAudioCodec,
                FIRST_AUDIO_INPUT + index);
        if (audioCodecs[index] == NULL) {
            printf("Cannot allocate audio codec #%d.\n", FIRST_AUDIO_INPUT +
                index);
            return(FALSE);
        }
        *(recorderPtr++) = audioCodecs[index];
    }
```

```
    // Allocate the video output:
    videoOutput = VdrAllocateResource(vdrHandle, ResourceVideoOutput,
            VIDEO_OUTPUT_NUM);
    if (videoOutput == NULL) {
        printf("Cannot allocate video output.\n");
        return(FALSE);
    }

    // Get a handle used to connect to the video input:
    videoInput = VdrGetResourceConnectionHandle(vdrHandle, ResourceVideoInput,
            VIDEO_INPUT_NUM);
    if (videoInput == NULL) {
        printf("Cannot get video input.\n");
        return(FALSE);
    }

    // Allocate audio outputs:
    for (index = 0; index < NUM_INDICES(audioOutputs); ++index) {
        audioOutputs[index] = VdrAllocateResource(vdrHandle, ResourceAudioOutput,
                FIRST_AUDIO_OUTPUT + index);
        if (audioOutputs[index] == NULL) {
            printf("Cannot allocate audio output #%d.\n", FIRST_AUDIO_OUTPUT +
                index);
            return(FALSE);
        }
    }

    // Make the default crosspoint connections videoInput->videoOutput and
    // videoInput->mpegEncoder:
    if (! VdrDefaultEvent(vdrHandle, NULL, EventConnectResources, videoInput,
            videoOutput)) {
        printf("Cannot connect input->output.\n");
        return(FALSE);
    }

    if (! VdrDefaultEvent(vdrHandle, NULL, EventConnectResources, videoInput,
            mpegEncoder)) {
        printf("Cannot connect input->encoder.\n");
        return(FALSE);
    }

     // Make a crosspoint connection from mpegDecoder->videoOutput except when
    // idle or recording:
    eventHandle = VdrScheduleEvent(vdrHandle, INT_MIN, EventConnectResources,
            mpegDecoder, videoOutput);
    if (eventHandle == NULL) {
        printf("Cannot connect decoder->output.\n");
        return(FALSE);
    }

    return(TRUE);
}

//
// Fxn: Clean up by closing the control port
// Det: this also detaches any movies from the timeline, closes all handles
//      for the port, and frees any resources allocated to the port.
//
BOOL Cleanup(void)
{
    UINT index;

    if (! VdrClosePort(vdrHandle)) {
        printf("Cannot close port.\n");
```

```
        return(FALSE);
    }
    // NULL all local handles (not strictly necessary if the program
    // exits immediately after calling Cleanup, but a good habit):
    mpegEncoder = NULL;
    mpegDecoder = NULL;
    videoInput = NULL;
    videoOutput = NULL;
    connectHandle = NULL;
    vdrHandle = NULL;
    for (index= 0; index < NUM_INDICES(audioCodecs); ++index) {
        audioCodecs[index] = NULL;
    }
    for (index= 0; index < NUM_INDICES(recorders); ++index) {
        recorders[index] = NULL;
    }
    for (index= 0; index < NUM_INDICES(recorders); ++index) {
        audioOutputs[index]= NULL;
    }
    return(TRUE);
}

// Shuttle rate for normal play/record:
#define SHUTTLE_RATE  1.0

//
// Fxn: play the given movie
//
BOOL PlayMovie(VdrHandle vdrHandle, UINT durationSeconds)
{

    // Cue for playback:
    if (! VdrCuePlay(vdrHandle, SHUTTLE_RATE)) {
        printf("Cannot cue play.\n");
        return(FALSE);
    }

    // Add a 'Sleep(500)' here if it is desirable to have the system
    // still before playback ...

    // Play:
    if (! VdrShuttle(vdrHandle, SHUTTLE_RATE)) {
        printf("Cannot begin playback.\n");
        return(FALSE);
    }

    // Wait until the movie is done playing (add an extra 200 msec
    // to ensure that we don't stop playing early):
    Sleep((durationSeconds * 1000) + 200);

    return(TRUE);
}


//
// Fxn: record the given movie with the given duration (in seconds):
//
BOOL RecordMovie(VdrHandle vdrHandle, UINT durationSeconds)
{
    // Cue for record:
    if (! VdrCueRecord(vdrHandle)) {
        printf("Cannot cue record \n");
        return(FALSE);
```

```
    }

    // Record:
    if (! VdrShuttle(vdrHandle, SHUTTLE_RATE)) {
        printf("Cannot cue shuttle \n");
        return(FALSE);
    }

    // Wait until the movie is done recording (add an extra 200 msec
    // to ensure that we don't stop recording early):
    Sleep((durationSeconds * 1000) + 200);

    // Stop recording:
    if (! VdrIdle(vdrHandle)) {
        printf("Cannot idle.\n");
        return(FALSE);
    }

    return(TRUE);

}

//
// Fxn: convert seconds to number of fields:
//
UINT SecondsToFields(ConnectHandle connectHandle, UINT seconds)
{
    if (CfgGetStandard(connectHandle) == PCI_PAL_625_MODE) {
        // 625-line mode:
        return(seconds * 50);
    }
    else {
        // 525-line mode:
        return((UINT) ((seconds * 60) / 1.001));
    }
}

//
// The main entry point.
//
void main(int argc, char *argv[])
{
    MovieHandle movieHandle;
    UINT durationSeconds;
    const CHAR* movieName= NULL;
    const CHAR* arg;
    INT argIndex;

    // Process arguments:
    for (argIndex = 1; argIndex < argc; ++argIndex) {
        arg= argv[argIndex];
        if (arg[0] == '-') {
            // Option argument:
            switch (arg[1]) {
            case 's':
                ++argIndex;
                durationSeconds = atoi(argv[argIndex]);
                break;
            default:
                Usage(argv[0]);
                exit(1);
            }
        }
```

```
        else {
            movieName= arg;
        }
    }

    if (movieName == NULL) {
        Usage(argv[0]);
        exit(1);
    }

    // Check to see if the movie already exists.
    if (PdrMovieExists(connectHandle, movieName)) {
        printf("Movie name already exists.\n");
        exit(1);
    }

    printf("Configuring resources ...\n");
    if (! ConfigureResources()) {
        exit(1);
    }

    // Attach the movie to the timeline; the mark-out point is set so that
    // the duration of the clip matches the desired time:
    printf("Attaching movie \"%s\"...\n", movieName);
    movieHandle = VdrAttachMovieWithMarks(movieName, NUM_INDICES(recorders),
                recorders, NULL, ShiftAfter, MarkLongest, 0,
                SecondsToFields(connectHandle, durationSeconds));
    if (movieHandle == NULL) {
        printf("Cannot attach movie \"%s\".\n", movieName);
        exit(1);
    }

    printf("Recording movie ...\n");
    if (! RecordMovie(vdrHandle, durationSeconds)) {
        exit(1);
    }

    // Set the position back to the starting position for the movie:
    VdrSetPosition(vdrHandle, VdrGetMovieStartPosition(movieHandle,
                mpegDecoder));

    printf("Playing movie ...\n");
    PlayMovie(vdrHandle, durationSeconds);

    // Detach the movie from the timeline (not really necessary, since
    // Cleanup() will detach the movie):
    printf("Detaching movie ...\n");
    if (! VdrDetachMovie(movieHandle, ShiftAfter)) {
        printf("Cannot detach movie.\n");
        exit(1);
    }

    printf("Cleaning up ...\n");
    Cleanup();
}
```

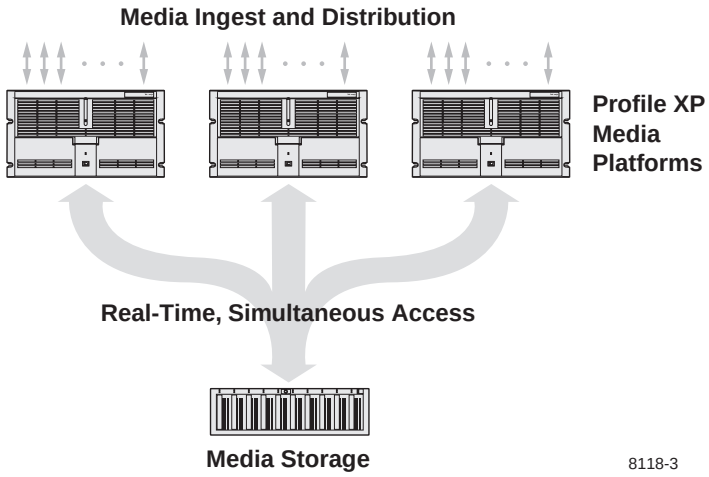
Chapter 9

The Media Area Network

This chapter provides an overview of the Grass Valley Group Media Area Network (MAN) for software application developers. It explains how the Media Area Network shared storage is different from locally attached Profile XP Media Platform storage, and how software applications must take these differences into account.

The Grass Valley Group *Media Area Network Instruction Manual* includes installation, operation, and service information for the Media Area Network.

The Media Area Network is Grass Valley Group's shared storage solution that gives multiple Profile XP Media Platforms access to a common pool of media, as illustrated in the following diagram.



The Media Area Network gives broadcast professionals the ability to do the following:

- Access media dynamically from multiple Profile XP Media Platforms
- Distribute media simultaneously from multiple Profile XP Media Platforms, even while ingest is ongoing.
- Store media more efficiently, without the additional space requirements for duplicate copies.
- Manage media simply without the need to track and update multiple copies.

Key features of the Media Area Network

The key features of the Media Area Network are as follows:

Many channels — Up to 48 channels of video and 300 channels of audio are available for play and record operations.

Large storage capacity — Up to 20,000 movies can be stored in the shared database.

Concurrent access — New media can begin playing out before recording is completed.

Simultaneous access — Multiple channels can play out at the same time from the same media.

Compatible with existing hardware and software — The Media Area Network works with Profile XP products and options, including the PFC500, the Profile Network Archive and the ContentShare platform.

No single-point-of-failure — You can set up your Media Area Network with full redundancy and failover capabilities.

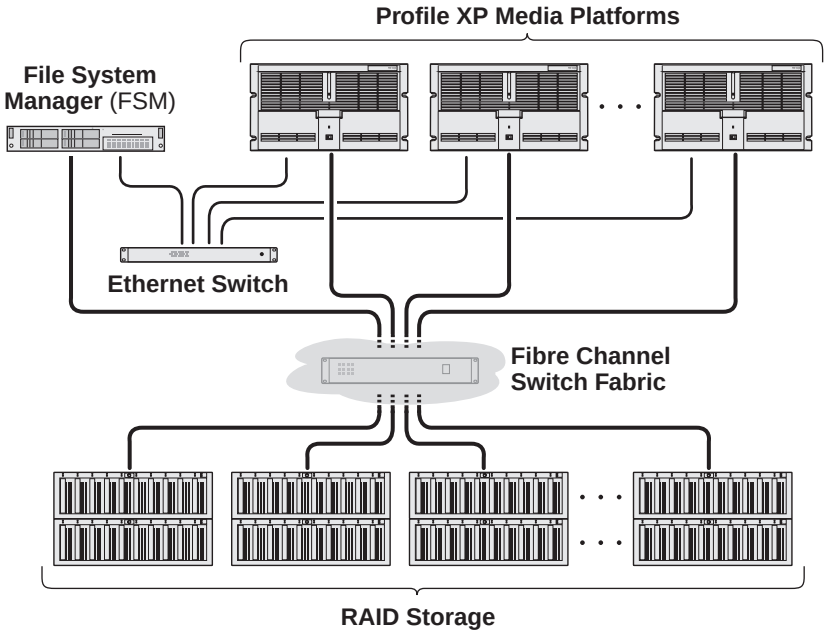
Automatic monitoring — The Media Area Network is ready to plug in to the NetCentral II system for monitoring and preemptive fault detection.

Overview of the Media Area Network

The primary functionality of the Media Area Network is described in the following topics. Please refer to the *Profile Media Area Network Instruction Manual* for detailed information on installing, operating, and maintaining a Media Area Network

- *Media Area Network hardware* — Illustrates devices and cabling with explanations of data flow for a basic system.
- *Media Area Network file system software* — Explains how the software components of the file system control data transactions.
- *Movie database software* — Explains how the Movie database makes media equally available from multiple Profile XP Media Platforms.
- *Fibre Channel redundancy* — Illustrates devices and cabling with explanations of failover mechanisms for a system with Fibre Channel redundancy only.
- *Fibre Channel failover* — Explains the sequence of failover mechanisms and failover states that maintain media access through the Fibre Channel switch fabric.
- *File System Manager redundancy* — Illustrates additional devices and cabling that extend redundancy to the file system.
- *File System Manager failover* — Explains how the hardware and software components of the failover system interact to enable continued operation.

Media Area Network hardware



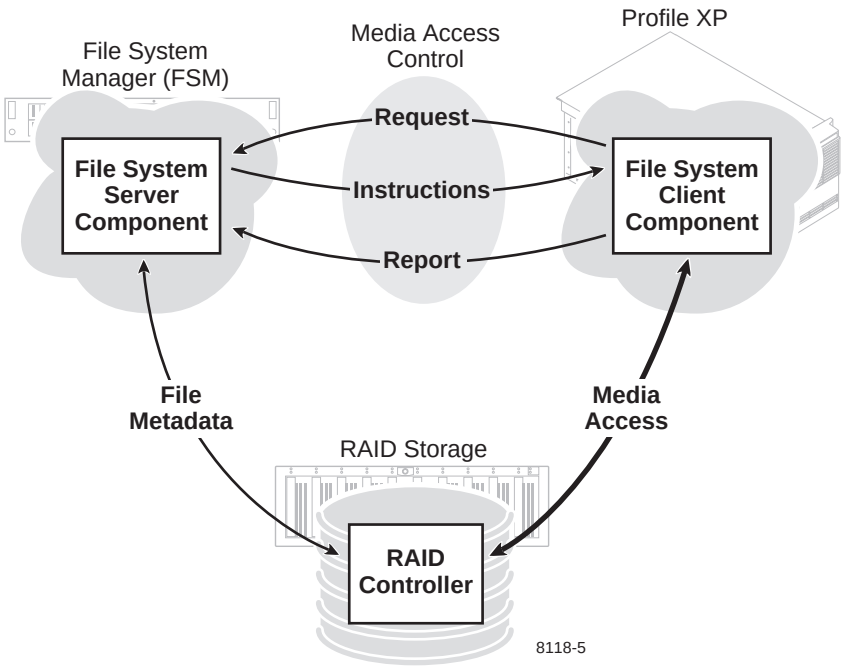
The Profile XP Media Platforms and the RAID arrays are connected to the Fibre Channel switch fabric. The Fibre Channel switch fabric is made up of one or more interconnected Fibre Channel switches, depending on the number of connections needed. All media access takes place over this Fibre Channel network, with the Fibre Channel switch fabric making the connection between the Profile XP accessing the media and the RAID array containing the media. The Fibre Channel network makes its maximum data rate available simultaneously to each Fibre Channel port.

The File System Manager (FSM) and the Profile XP Media Platforms are connected to the Ethernet switch. Control information flows through the Ethernet network as the File System Manager directs the reading and writing of data between the Profile XP systems and the shared storage. The File System Manager is also connected to the Fibre Channel network.

Media Area Network file system software

This section explains the software components that work together to control shared access to the RAID storage. These components include:

- File system server software component
- File system client software component
- RAID controller



The Media Area Network file system is a licensed third-party file system that is directly accessible to the Windows NT operating system. The Profile API has been modified to manage media files on the Media Area Network file system. Although any Windows NT application can directly access the media files, they must not move, modify, or delete media files. **All media file management must be accomplished through Profile XP applications or the Profile API.**

The Media Area Network file system has a client/server structure, with the server component residing on the File System Manager and the client component residing on the Profile XP Media Platform. These components run as Windows NT services. The server component acts as the centralized file system and interacts with the client components to control all media access within the Media Area Network.

On the File System Manager, the server component knows where the media is located, when the media is being accessed, and which Profile XP systems are accessing the media. No Profile XP Media Platform is allowed to access media without first obtaining instructions from the server. In this way the server enforces strict rules so that each Profile XP Media Platform can access media as it if “owned” the media, without regard for the access taking place on other Profile XP Media Platforms.

The file system server component can store metadata information about the media (the physical location of files on the disks) on the local hard drive of the FSM or on the RAID storage of the Media Area Network. When metadata is on the RAID storage of the Media Area Network, it is in a dedicated partition so that it is kept separate from the shared media.

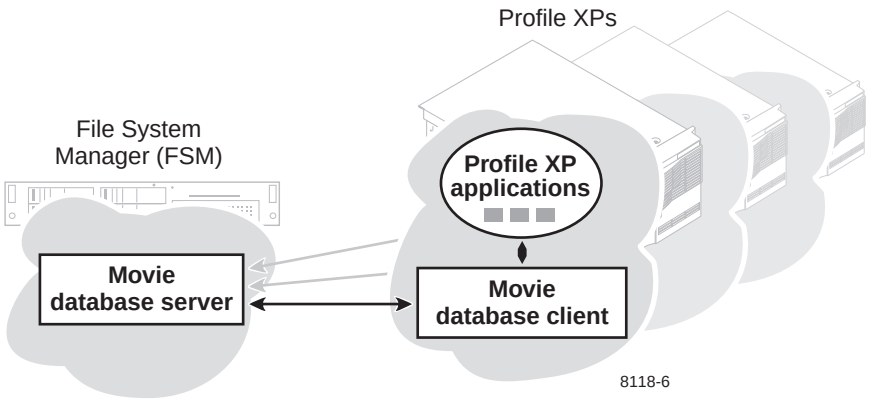
On each Profile XP Media Platform, the client component of the Media Area Network file system software implements all data read/write operations for the local Profile XP. The client dialogs with the server to request media, receive instructions, and report results as media is accessed by the local Profile XP.

On the RAID arrays, the RAID controller software receives the data read/write commands from the Profile XP Media Platform and implements them across the structure of the RAID array. The RAID controller software can also communicate with the server component of the Media Area Network file system software. This communication path is used when file system metadata is stored on the RAID arrays.

Movie database software

The movie database is built on an embedded version of the Microsoft SQL database. Third-party software applications must not access, modify, or configure the database directly. **All movie management must be accomplished through Profile XP applications or the Profile API.**

The Movie database manages native Profile XP media. This section explains how the Movie database works with the Media Area Network as media is created, shared and edited by applications on multiple Profile XP Media Platforms.

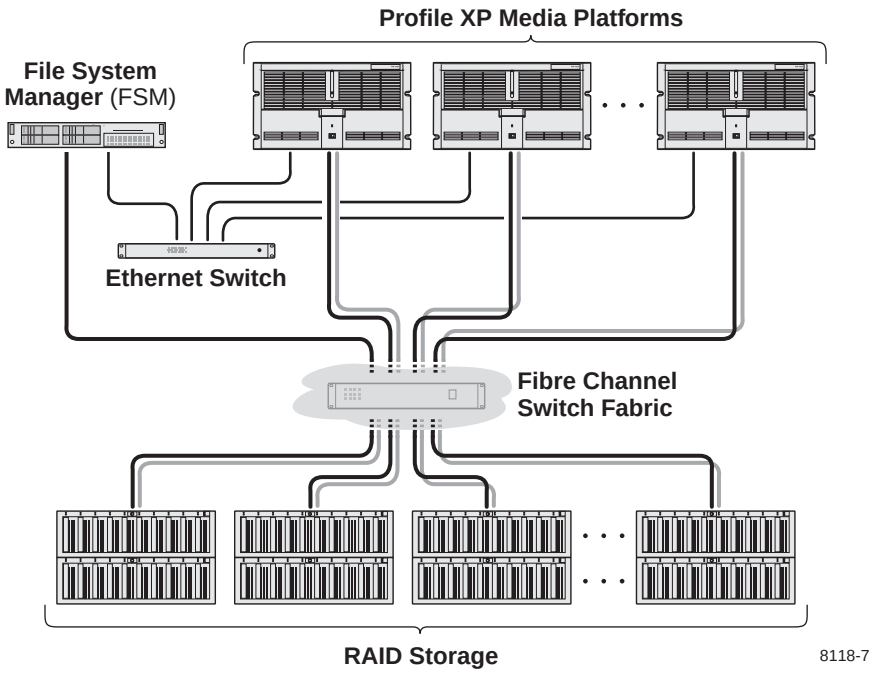


Like the Media Area Network file system, the Movie database is structured as a client/server application with the client residing on each Profile XP Media Platform and the server on the File System Manager. While the file system views Profile XP media as generic data files, the Movie database views Profile XP media as “Movies” and maintains a database record for each Movie. The record describes the tracks and timing information that define the Movie.

The server component of the Movie database maintains a centralized and coherent view of the Movies in the Media Area Network and propagates this view to the client components. In this way Profile XP applications “see” the Movies through their client component and can interact with the Movie database as if they alone “owned” the media.

Fibre Channel redundancy

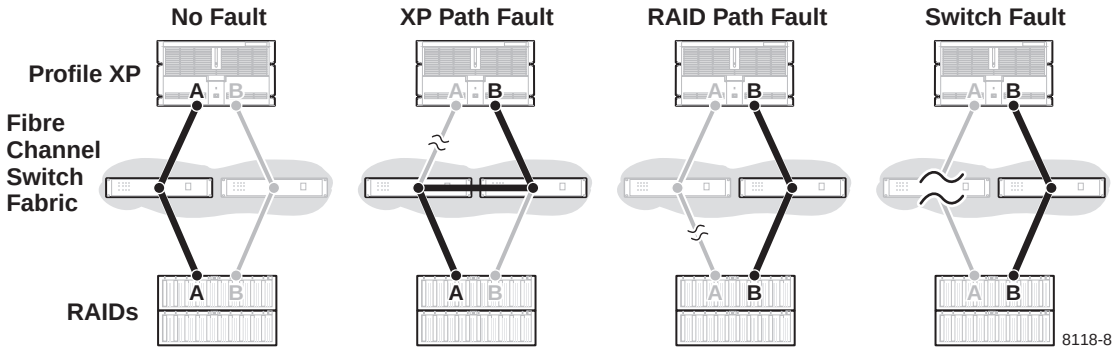
This section explains the hardware for a Media Area Network with redundancy in the Fibre Channel network only. Read “File System Manager redundancy” on page 198 to learn about the fully redundant Media Area Network.



A Media Area Network with Fibre Channel redundancy requires dual Fibre Channel connections for every device and a Fibre Channel switch fabric made up of two or more switches, as required to support the number of devices. Each Profile XP Media Platform is connected to two different switches. The same is true for each RAID storage device. The interconnections of the Fibre Channel switch fabric are thereby able to negotiate alternate pathways in the event of a fault. Read the next section, *Fibre Channel failover*for more information.

Fibre Channel failover

This section explains the sequence of failover mechanisms that work together to maintain access between a host and the RAID storage through the Fibre Channel switch fabric. The following diagrams use a Profile XP Media Platform as an example of the host. The diagrams illustrate a simplified view of the different states in which access can continue successfully, depending on the type of the fault.



For a system in which no fault has yet occurred, a Profile XP system uses its primary (A) Fibre Channel port for media access to and from the primary (A) controller on a RAID array. This access takes place over the Fibre Channel switch fabric. When a fault occurs along the Fibre Channel pathway and media access fails, the Profile XP system fails over to its backup (B) Fibre Channel port and tries the media access again.

If the fault that caused the Profile XP failover is in the path on the Profile XP side of the Fibre Channel switch, the Fibre Channel switch fabric is able to find an alternate pathway between the Profile XP system's backup (B) port and the primary (A) RAID controller. This allows the media access to be completed successfully without requiring a RAID controller failover.

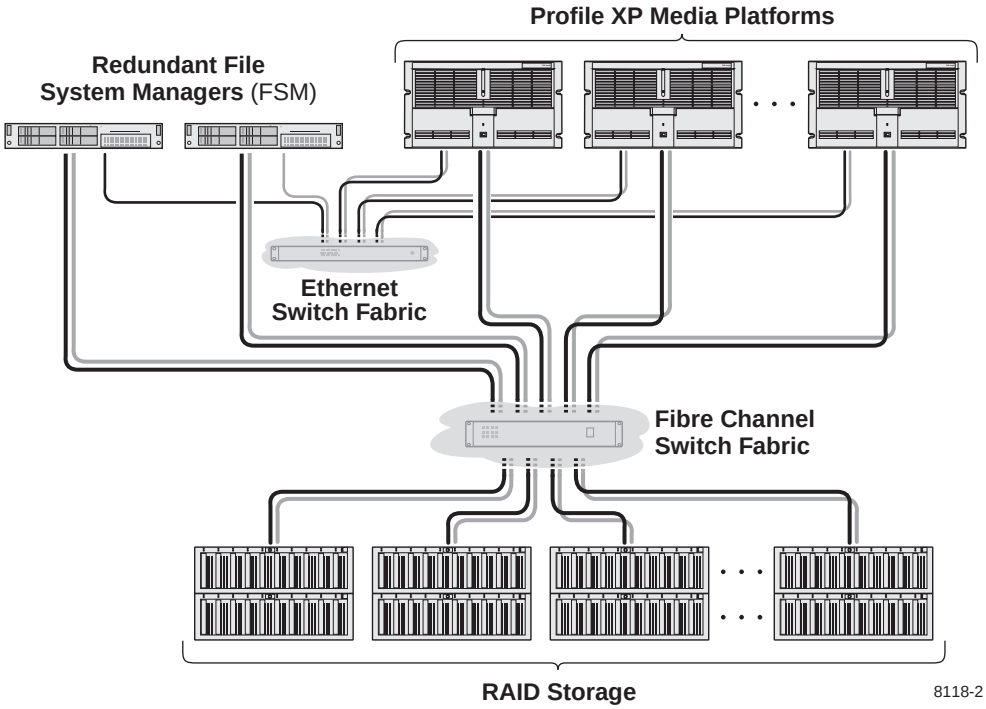
If the fault that caused the Profile XP failover is in the path on the RAID side of the Fibre Channel switch or if the Fibre Channel switch itself is faulty, the Fibre Channel switch fabric is unable to find an alternate pathway between the Profile XP system's backup (B) port and the primary (A) RAID controller. This causes the media access to fail again, in which case the Profile XP system uses its backup (B) Fibre Channel port to send failover commands to the RAID array. These commands instruct the RAID array to failover to its backup (B) controller. Once more the Profile XP system tries the media access. This time the Fibre Channel switch fabric is able to find the alternate pathway between the Profile XP system's backup (B) port and the backup (B) RAID controller.

These alternate paths remain active for as long as they operate without a fault, as there are no persistent primary/back-up relationships within the Fibre Channel network. When the fault is repaired on the original path, that path then becomes available for use as an alternate path in the event of subsequent faults.

NetCentral reports all faults, whether the fault occurs on a pathway in active use for current media access or whether the fault occurs on a reserve pathway that will be needed in the event of a failover.

File System Manager redundancy

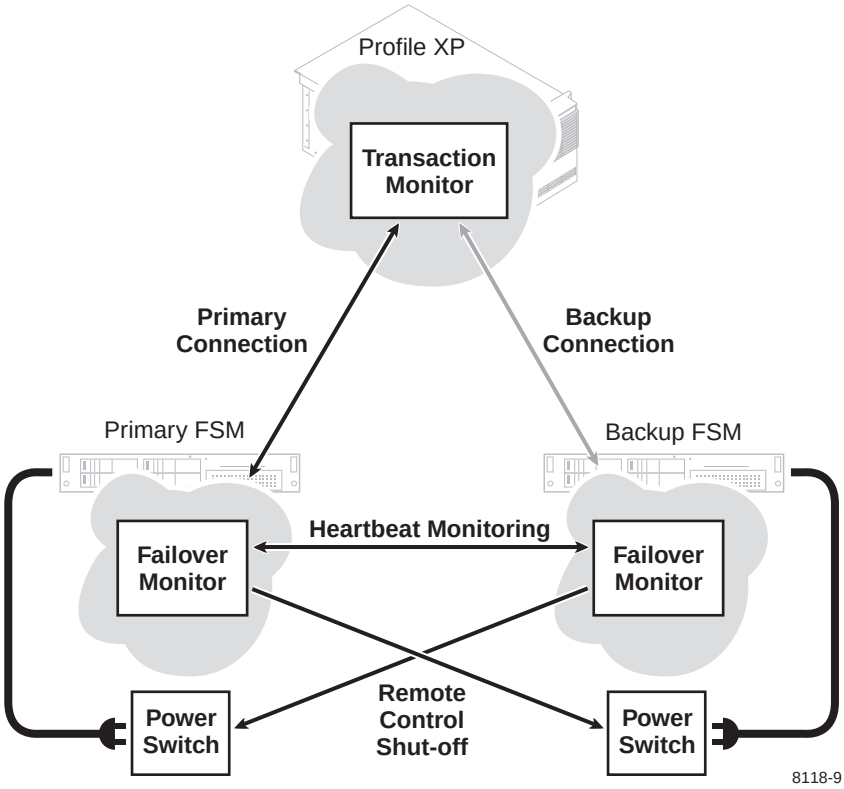
This section explains the hardware that provides redundancy for the File System Manager and its Media Area Network management mechanisms, such as the MAN file system and Movie database. This hardware, combined with the redundant Fibre Channel explained earlier in this chapter, supports a fully redundant Media Area Network with no single-point-of-failure throughout the entire system.



A Media Area Network with redundant File System Managers requires two FSMs that are interconnected through the cabling and devices of the failover system, as described in “File System Manager failover” on page 199. The FSMs have redundant connections to the Fibre Channel switch fabric and use failover mechanisms similar to those described in “Fibre Channel failover” on page 197. The FSMs each have one connection to the Ethernet switch fabric, while the Profile XP Media Platforms each have two connections to the Ethernet switch fabric. Redundant pathways are thereby created so that management information can continue to flow between the active FSM and the Profile XP Media Platforms in spite of faults along an Ethernet pathway, a Fibre Channel pathway, or in the event of a FSM failover.

File System Manager failover

This section explains the software and hardware mechanisms that work together to provide failover capabilities for the File System Manager's Media Area Network management mechanisms.



On each File System Manager, the Fail-over Monitor software component runs as a Windows NT service. The software monitors the status of its own FSM by constantly checking the FSM's Movie database, MAN file system and network connectivity. If everything checks out OK, the software sends a heartbeat pulse over a dedicated serial cable to the other FSM. If the software detects a critical problem, it stops the heartbeat pulse.

When Fail-over Monitor software running on the current backup FSM detects that the heartbeat coming from the current primary FSM has stopped, it triggers a failover in which the backup FSM takes control of the Media Area Network. The software then sends a shut-off command over a dedicated serial cable to the power switch of the primary FSM. This shut-off is necessary to ensure that the two File System Managers do not attempt to simultaneously manage the Media Area Network.

When Fail-over Monitor software running on the current primary FSM detects that the heartbeat coming from the current backup FSM has stopped, it does not trigger a failover. Rather, the software notifies the client devices of the Media Area Network that the backup FSM is unavailable.

On the Profile XP Media Platforms, Transaction Monitor software monitors Movie database transactions. If a transaction takes too long, the software checks with the backup FSM to verify if a failover has occurred. If a failover has occurred, the software switches the Profile XP

system over to the backup FSM. If a failover has not occurred, the software continues to monitor Movie database transactions, but waits longer before trying to verify a failover. This allows any momentary, non-critical pauses in media access to clear without triggering a failover.

When a failover occurs, the original backup FSM becomes the new primary FSM. Likewise, when the original primary FSM is restored, it becomes the new backup FSM. Thereafter, the new primary FSM remains active as the primary for as long as it operates continuously without a fault, as there are no persistent primary/back-up relationships in the operation of the file system.

NetCentral reports all faults, whether the fault occurs on a pathway in active use for current media access or whether the fault occurs on a reserve pathway that will be needed in the event of a failover.

Specifications

The following specification are preliminary and subject to change. Please consult with your Grass Valley Group representative to obtain the most recent specifications.

- Number of movies (masters) - 20,000, assuming the average movie references six media files
- Movies per group - 8,000
- Shortest clip length - 5 seconds
- Back to back clip play-out - 5 to 7 seconds
- Record to play-out time - 10 seconds
- Simultaneous play-outs of same clip - TBD
- Queue time vs. number of clips - TBD
- Serial protocol foreground (synchronous) response time - 10 milliseconds
- Serial protocol backgroud (asynchronous) response time - network dependent, typically 90 milliseconds, up to 1 second on a busy network
- File System Manager failover - up to 20 seconds. Playout in progress continues during the failover, although records fail.

Number of channels and bandwidth specifications are configuration dependent. Please consult with your Grass Valley Group representative for Media Area Network design criteria and guidance.

Developing Media Area Network software applications

From a programmer’s point of view, a Profile XP Media Platform connected to a Media Area Network is very similar to a Profile XP Media Platform with local storage. Software applications can run directly on the local system processor through the Profile API or the Event Scheduler Engine, on a remote PC using PortServer and the TekRem library, or control a channel through VDCP, Odetics, BVW, or Profile protocol by assigning the channel to an RS-422 port in VdrPanel. Many existing Profile applications will require little or no change to operate in a Media Area Network environment. Some movie ownership changes in the TekPdr

library imposed by the use of a shared Pdr database may require minor adjustments in some applications. Movie ownership becomes more critical in a Media Area Network. Refer to “TekPdr changes” on page 201 for more information.

The most significant difference in Profile applications running on Profile XP Media Platforms connected to a Media Area Network is that many of the previously required Video Network (Fibre Channel or Ethernet) transfers are no longer necessary. All material recorded by any channel is immediately available to all other channels without any intermediate operations. Software applications must support this new model of material management.

Total system bandwidth is defined by the number of storage processors, drives, and Inter-Switch Links. Grass Valley Group works closely with Media Area Network to design systems that meet customer bandwidth and failover needs. Bandwidth must be managed on each Profile XP Media Platform to ensure that the total system bandwidth is not exceeded. Control applications must not use more bandwidth than allocated in the system design. The application must manage the Profile XP Media Platform bandwidth including record and playback of video through traditional I/O and SDTI, as well as streaming transfers over a the Video Network. High-bandwidth operations such as off-speed play, jog, shuttle, and scrub audio may require up to 50% more bandwidth per channel. Both the allocated bandwith on each Profile XP Media Platform and the total Media Area Network bandwidth must be respected.

Software applications must take into account failover events. Timeouts may need to be lengthened, since requests will generally complete after the failover event. Timing-critical operations need to be closely analysed and optimized in a Media Area Network environment.

File system changes

The file system used in a Media Area Network is accessible from the Windows NT environment.

Dataset name

By convention, the shared drive is mounted as V for video on all the clients. This drive letter is used as the dataset identifier, replacing the INT or EXT of the Media File System. Applications using XfrRequest() or FTP to stream movies must use the new dataset name. The dataset name must be a single letter in the Media Area Network.

Application access to the file system

In this release, no applications may access media files directly on the Windows NT-accessible file system. All manipulation of media files must be accomplished through the Profile API.

TekPdr changes

TekPdr functions have been slightly modified to accommodate the database used in the Media Area Network. The following sections describe the main diffences in TekPdr behavior.

Different Open Modes

When opening clips with the modified TekPdr, the following open modes are in effect.

PdrOpenExclusive: Unchanged from earlier releases.

PdrOpenMultiple/PdrOpenReadWrite: PdrOpenReadWrite is highly preferred over PdrOpenMultiple. The first instance of the movie opened in PdrOpenReadWrite mode is the only one that can read/write to the Movie Database. Until that instance is closed, no other instance will be able to record changes.

PdrOpenReadOnly: This new mode allows a movie to be opened as read-only, and will not allow a write.

PdrDeleteMovie is not synchronous

When a movie is deleted with PdrDeleteMovie, the corresponding media files are moved to V:\attic\ for subsequent deletion by the garbage collection mechanism.

PdrSetMovieReadOnly() and PdrSetMovieReadWrite()

These functions are now equivalent to PdrSetMovieLocked() and PdrSetMovieUnlocked(), respectively. They set and reset the same attribute, **PdrLocked**. Media files are not set read-only, only the movie attributes. Only the subject movie is affected. Other movies that reference the same media files will still be ReadWrite.

PdrControlRO bit

This bit in the movie attributes is now obsolete.

PdrReadOnly

This movie attribute is now purely informational, indicating that a movie was opened in the read-only mode, or that this instance of a movie was not the first instance to be opened "multiple".

PdrExtensions

PdrExtensions API functions will not be compatible with VIP applications. User data is now a special case of PdrExtensions. Therefore using PdrFindFirst/NextExtensionPos() will also enumerate the userdata records.

Obsolete functions

The following functions are not supported in Profile System Software version 5.0 and higher.

- PdrCopyMovie
- PdrGetRegistry
- PdrSetRegistry

Common Movie Format database access

Although the Common Movie Format remains the same, movie definitions are now stored in the centralized movie database, and not in a *.cmf* file. Access to movie definitions through the Profile API remain unchanged, although response time may be lengthened due to the network link to the database.

Tools and utilities that accessed the *.cmf* file directly can no longer be used. Access to the database must occur through the Profile API, or through tools and utilities provided by Grass Valley Group.

Direct third-party access to the movie database is not permitted due to licensing requirements.

Recommendations for verifying applications

It is usually best to begin verification of software application operation on a very basic Media Area Network consisting of two Profile XP Media Platforms. By limiting the complexity of the environment, it is easier to simulate error conditions and to verify appropriate responses.

Once basic functionality is confirmed, verification of operation on a "busy" system should include a larger number of clients, more active software applications, etc. This verification might test the application's sensitivity to PDR response time, for example.

Finally, the software application should be verified on a very large Media Area Network to discover scaling issues, for example.

Chapter 10

PdrMovie Extensions

PdrMovie extensions make use of the existing user data mechanism but allows new, user-specific extension data to be added to and associated with the standard PdrMovie structure, thus taking advantage of existing networking and archive features. Any Profile software developer can make use of the PdrMovie extensions. Since all extensions use a standard header, applications may exchange nonmovie data if they understand the underlying data structure identified by the extension header. For a listing of the C data structure for the extensions, refer to "The PdrExtensionInfo Data Structure" on page 221.

NOTE: Though user data is used to store extension data structures, the functions PdrSetUserData() and PdrGetUserData() cannot be used with these extensions.

Like all user data, extensions may be associated with a movie, track, or media segment. This is known as the extension's location. Extensions differ from regular TekPdr user data in the following ways:

- No size limit.
- A single extension may be stored across multiple user data blocks.
- Larger, more meaningful header.
- More sophisticated lookup mechanism. Extensions may be enumerated using specific search/filter criteria.
- No location restriction.
- User data can only be associated with a track if it contains segments. An extension may be associated with a track without regard to the presence of segments on that track. Multiple extensions may also be associated with the same location and time.

A new bit is added to the movie state attributes mask to identify movies that have been extended. Both simple and complex movies may have the extension bit set. The extension bit is set and cleared automatically by the system when extensions are inserted or deleted. The Media Manager application will distinguish between movies with extensions and without, but will not show any specific vendor information in the contents list.

A PdrMovie extension contains the elements listed in Table 14.

Table 14. PdrMovie extension elements

Element	Description
fieldOffset (int)	Extensions influence a single point of time or a range of time starting at fieldOffset.
vendorName (char[32])	Name of the software vendor which defined the specific dataClass and dataType. For example: "Lightworks," "EditStar," or "Grass Valley Group."

Table 14. PdrMovie extension elements

dataClass (char[32])	Class this type of extension data falls into. This field may be used by client applications to exclude certain classes of extensions. For example, an extension whose category is “AudioMixEffect” may be ignored by clients that don’t use audio.
dataType (char[80])	Identifies the specific extension data structure.
dataName (char[80])	This string allows you to give the extension a meaningful name.
dataVersion (char[32])	Anticipates future changes to a particular extension data structure.
dataLen (UINT)	Number of bytes in the extension data structure.
data (variable length)	The actual extension data structure that follows the header.

This version of the Profile SDK includes eight functions that control PdrMovie extensions:

- **PdrInsertExtension()** copies data into a movie’s list of extensions, and **PdrDeleteExtensionAtPos()** removes an extension from the movie structure.
- **PdrReadExtension()** returns a data buffer that corresponds to the named movie data structure, and **PdrFreeExtension()** frees the memory that was allocated with **PdrReadExtension()**.
- **PdrFindFirstExtensionPos()** initiates the enumeration of extensions in a particular movie. The enumeration begins by choosing which extension to use as search criteria. **PdrFindNextExtensionPos()** continues the enumeration that **PdrFindFirstExtensionPos()** began, returning the next extension that matches the given search criteria.
- **PdrGetExtensionAtPos()** returns the extension data for the given extension position while **PdrGetExtensionIntoAtPos()** returns only header information from an extension.

For more specifics on these functions, see the *Profile Family Software Development Kit Reference Manual*.

Grass Valley Group Common Extensions

The TekPdr library defines several extension structures that you may use to store edit and hardware setup information that doesn't appear in the basic movie structure. These are referred to as the Grass Valley Group Common Extensions. Currently, there are common extensions available for digital video effects (DVE), audio, motion, General Purpose Interface (GPI), and metadata:

- Simple dissolves
- Advanced dissolves
- Simple wipes
- Advanced wipes
- Keys
- Video fade-to-matte
- Audio mix
- Audio levels
- Source motion effects
- General Purpose Interface (GPI)
- Movie metadata
- Segment metadata

More discussions on these extensions follow in the next pages.

Video Mix Effects Extensions

This class of extension is used to control the Profile mix effects board.

Simple Dissolve

The most basic video mix-effect extension is the dissolve. This extension allows you to dissolve from one video source to another across time.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "Dissolve"
fieldOffset = +-<0..n>
```

Table 15. Simple dissolve extension elements

Element	Description
numFields (int)	Duration of the dissolve in fields.
endingMix (int)	Value between 0 and VME_MAX_PROGRESS. Video is dissolved in the background to foreground direction with 0 being fully background and VME_MAX_PROGRESS fully foreground. The mix value persists beyond the duration of the dissolve, therefore if the endingMix is less than VME_MAX_PROGRESS, the mix-effects output will continue to show a combination of the foreground and background video sources.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
backgroundSrc (int)	Logical, zero-based video track number associated with the mixer's background input. This value allows the system to connect the appropriate codec resource to the mix-effect background input when the movie is attached to a port.

Advanced Dissolve

The advanced dissolve extension adds fields for modulating the dissolve using a key signal.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "AdvancedDissolve"
fieldOffset = +-<0..n>
```

Table 16. Advanced dissolve elements

Element	Description
numFields (int)	Duration of the dissolve in fields.
endingMix (int)	Value range 0 thru VME_MAX_PROGRESS. Video is dissolved in the background to foreground direction with 0 being fully background and VME_MAX_PROGRESS fully foreground. The mix value persists beyond the duration of the dissolve, therefore if the endingMix is less than VME_MAX_PROGRESS, the mix-effects output will continue to show a combination of the foreground and background video sources.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
backgroundSrc (int)	Logical, zero-based video track number associated with the mixer's background input. This value allows the system to connect the appropriate codec resource to the mix-effect background input when the movie is attached to a port.
keySrc (int)	Logical, zero-based video track number associated with the mixer's key input. This value allows the system to connect the appropriate codec resource to the mix-effect key input when the movie is attached to a port.
nClipLevel (int)	A clip level in range 0 thru VME_MAX_CLIP_LEVEL.
eGain (float)	A gain value for the key in range VME_MIN_GAIN_LEVEL thru VME_MAX_GAIN_LEVEL
invertMode (int)	Enumerated type specifying the key invert mode. Must be one of **VmeNormalKey** or **VmeInvertedKey**.
edgeMode (int)	Enumerated type specifying the key edge mode. Must be one of **VmeAdditiveKey** or **VmeMultiplativeKey**.

Simple Wipe

The simple wipe uses these fields to perform a simple, background to foreground wipe.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "Wipe"
fieldOffset = +-<0..n>
```

Table 17. Simple wipe extension elements

Element	Description
numFields (int)	Duration of the wipe transition in fields.
endingMix (int)	Value range 0 thru VME_MAX_PROGRESS. Video is wiped from background to foreground with 0 being fully background and VME_MAX_PROGRESS fully foreground. The mix value persists beyond the duration of the dissolve, therefore if the endingMix is less than VME_MAX_PROGRESS, the mix-effects output will continue to show a combination of the foreground and background video sources.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
backgroundSrc (int)	Logical, zero-based video track number associated with the mixer's background input. This value allows the system to connect the appropriate codec resource to the mix-effect background input when the movie is attached to a port.
wipeShape (int)	SMPTE wipe code. All wipe codes may not be supported.

Advanced Wipe

The advanced wipe extension adds fields for rotation and position transforms, horizontal and vertical wipe modulation, border matte and edge control, and modulating the wipe using a key signal. Rotation and position settings are assumed to be valid when bTransform equals 1. If bTransform equals 0, these settings are ignored. Horizontal and vertical wipe modulation settings are valid when bHVMod equals 1. Keyer modulation settings are valid when bKeyer equals 1. Border settings are valid when bBorder equals 1.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "AdvancedWipe"
fieldOffset = +-<0..n>
```

Table 18. Advanced wipe extension elements

Element	Description
numFields (int)	Duration of the wipe transition in fields.
endingMix (int)	Value range 0 thru VME_MAX_PROGRESS. Video is wiped from background to foreground with 0 being fully background and VME_MAX_PROGRESS fully foreground. The mix value persists beyond the duration of the dissolve, therefore if the endingMix is less than VME_MAX_PROGRESS, the mix-effects output will continue to show a combination of the foreground and background video sources.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
backgroundSrc (int)	Logical, zero-based video track number associated with the mixer's background input. This value allows the system to connect the appropriate codec resource to the mix-effect background input when the movie is attached to a port.
wipeShape (int)	SMPTE wipe code. All wipe codes may not be supported.
bTransform (byte)	1 to indicate valid transform parameters, 0 to ignore transform parameters.
nX (int)	The position of the end of the transition, range: –10 * VME_MAX_SCREEN_HEIGHT thru 10 * VME_MAX_SCREEN_HEIGHT.
nY (int)	The position of the end of the transition, range: –10 * VME_MAX_SCREEN_HEIGHT thru 10 * VME_MAX_SCREEN_HEIGHT.
rotation (int)	The integer number mapped to the range: 0 thru VME_MAX_ROTATION.
nHMult (int)	The horizontal integer multiplier value. Limit to range 1 thru VME_MAX_MULTIPLIER.
nVMult (int)	The vertical integer multiplier value. Limit to range 1 thru VME_MAX_MULTIPLIER.
bKeyer (byte)	1 to turn key modulation on, 0 to turn the key modulation off.

Table 18. Advanced wipe extension elements

keySrc (int)	Logical, zero-based video track number associated with the mixer's key input. This value allows the system to connect the appropriate codec resource to the mix-effect key input when the movie is attached to a port.
nClipLevel (int)	A clip level in the range: 0 thru VME_MAX_CLIP_LEVEL.
eGain (float)	A gain value for the key in range VME_MIN_GAIN_VALUE thru VME_MAX_GAIN_VALUE.
invertMode (int)	Enumerated type specifying the key invert mode. Must be one of **VmeNormalKey** or **VmeInvertedKey**.
edgeMode (int)	Enumerated type specifying the key edge mode. Must be one of **VmeAdditiveKey** or **VmeMultiplativeKey**.
bHVMod (byte)	1 to turn horizontal/vertical modulation on, 0 to turn modulation off.
modTypeH (int)	The modulation waveform type.
nHAmp (int)	The peak-to-peak amplitude of the waveform. Units match other screen units where **VmeMaxHeight** is full screen height.
eHFreq (float)	A frequency value for the waveform.
nHPhase (int)	The starting phase of the waveform.
nHDeltaPhase (int)	The change in phase from frame to frame.
modTypeV (int)	The modulation waveform type.
nVAmp (int)	The peak-to-peak amplitude of the waveform. Units match other screen units where **VmeMaxHeight** is full screen height.
eVFreq (float)	A frequency value for the waveform.
nVPhase (int)	The starting phase of the waveform.
nVDeltaPhase (int)	The change in phase from frame to frame.
bBorder (byte)	1 to turn border on, 0 to turn the border off.
borderWidth (int)	The integer width of the border.
nSoftness (int)	The integer which specifies the edge softness.
nHue (int)	The border hue value in degrees.
nChroma (int)	The border chroma saturation value.
nLuma (int)	The border luminance value.

Key

The key extension allows you to use an arbitrary video source as a key to reveal a background video source.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "Key"
fieldOffset = +-<0..n>
```

Table 19. Key extension elements

Element	Description
bKeyer (byte)	1 to turn the keyer on. 0 to turn the keyer off.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
backgroundSrc (int)	Logical, zero-based video track number associated with the mixer's 'background' input. This value allows the system to connect the appropriate codec resource to the mix-effect background input when the movie is attached to a port.
keySrc (int)	Logical, zero-based video track number associated with the mixer's 'key' input. This value allows the system to connect the appropriate codec resource to the mix-effect key input when the movie is attached to a port.
nClipLevel (int)	A clip level in range 0..VME_MAX_CLIP_LEVEL
eGain (float)	A gain value for the key in range VME_MIN_GAIN_LEVEL..VME_MAX_GAIN_LEVEL
invertMode (int)	Enumerated type specifying the key invert mode. Must be one of **VmeNormalKey** or **VmeInvertedKey**.
edgeMode (int)	Enumerated type specifying the key edge mode. Must be one of **VmeAdditiveKey** or **VmeMultiplativeKey**.

Video Fade-to-Matte

The video fade extensions allows you to control the video mixer's fade-to-matte hardware which is independent of its dissolve/wipe capabilities.

```
vendorName = "Grass Valley Group"
dataClass = "VideoMixEffect"
dataType = "FadeToMatte"
fieldOffset = +-<0..n>
```

Table 20. Fade-tomatte extension elements

Element	Description
numFields (int)	Length of the fade in number of video fields.
endingMix (int)	Value between 0 and VME_MAX_PROGRESS, where 0 is the pre-fade state and VME_MAX_PROGRESS is the post-fade state. The mix value persists beyond the duration of the fade, therefore if the endingMix is less than VME_MAX_PROGRESS, the mix-effects output will continue to show a combination of the video and matte sources.
videoOutput (int)	Logical video output number connected to the mix-effects output. This refers to the output resource that will be used when the movie is later attached to a port.
fadeDirection (int)	The enumerated type specifying the direction of the fade. This must be **VmeFadeDown** or **VmeFadeUp**.
nHue (int)	The hue value in degrees.
nChroma (int)	The chroma saturation value.
nLuma (int)	The luminance value.

Audio Mix Effects

This class off extensions are used to control the Profile audio mixing hardware.

Audio Mix

Settings include audio source to destination routing and mix coefficients. A single AudioMix extension is used to describe a single audio mix adjustment. An audio mix may involve multiple audio inputs mixing to a common audio output.

```
vendorName = "Grass Valley Group"
dataClass = "AudioMixEffect"
dataType = "AudioMix"
fieldOffset = +-<0..n>
```

Table 21. Audio mix extension elements

Element	Description
numFields (int)	Duration of the audio transition in fields.
audioOutput (int)	Logical audio output number. This refers to the output resource that will be used when the movie is later attached to a port.
numSources (int)	The number of audio inputs that contribute to the audio output.
audioSrc (int)	Logical, zero-based audio track number. This value allows the system to adjust the appropriate physical mix coefficient when the movie is attached to a port.
endingMix (double)	Level at which the audio source should be mixed into the audio output.
audioSrc (int)	Logical, zero-based audio track number. This value allows the system to adjust the appropriate physical mix coefficient when the movie is attached to a port.
endingMix (double)	Level at which the audio source should be mixed into the audio output.

Audio Level

This extension is used to control the Profile audio mixing hardware (current level values are multiplied with the current audio mix state). A single AudioLevel extension may be used to make any number of level adjustments, even on a field-by-field basis.

```
vendorName = "Grass Valley Group"
dataClass = "AudioMixEffect"
dataType = "AudioLevel"
fieldOffset = +-<0..n>
```

Table 22. Audio level extension elements

Element	Description
numLevelChanges (int)	Level at which the audio source should be mixed into the audio output.
fieldPos (int)	Number of fields after the extension field offset the audio level should match thze specified level.
level (double)	Level at which the audio source should be mixed into the audio output. Audio levels and mix values are additive.
fieldPos (int)	Number of fields after the extension field offset the audio level should match the specified level.
level (double)	Level at which the audio source should be mixed into the audio output. Audio levels and mix values are additive.

Motion Effects

Currently, there is only one kind of motion effect: source effects.

Source Effects

The source effects extension is used to control slow and fast motion playback as well as indicate extra “source handles” to be transferred with the movie when it is copied to a new location. This allows edits to be trimmed after transfer.

```
vendorName = "Grass Valley Group"
dataClass = "MotionEffect"
dataType = "SourceEffect"
fieldOffset = undefined
```

Table 23. Source effect extension elements

Element	Description
srcFieldOffset (int)	Indicates number of fields after the movie segment’s media mark-in value represent playable material. This value is used to transfer extra source material when the movie is copied to a new location.
srcNumFields (int)	Length, in fields, of playable material after the srcField-Offset.
targetNumFields (int)	Number of fields in which to execute the playable material. A targetNumFields shorter than srcNumFields indicates fast-motion playback. A targetNumFields value longer than srcNumFields indicates slow-motion playback.

External Control

This class of extensions is used to control devices external to the Profile video server.

GPI

This extension allows programmers to assert General Purpose Interface (GPI) triggers to control a device external to the Profile server. Triggered devices are indicated by name. The Profile configuration will associate the device name with a port, output number, trigger voltage and period, and pre-roll offset.

```
vendorName = "Grass Valley Group"
dataClass = "ExternalControl"
dataType = "GPI"
fieldOffset = +-<0..n>
```

Table 24. GPI extension element

Element	Description
deviceName (char[80])	Logical GPI device name.

Meta Data Extensions

There are two meta data extensions, MovieData and SegmentData.

MovieData

This extension is used for media management. Strings stored in this extension are assumed to be in Unicode. Since Grass Valley Group cannot anticipate customer specific meta data, the MovieData extension provides generic tag/value string pairs.

```
vendorName = "Grass Valley Group"
dataClass = "MetaData"
dataType = "MovieData"
fieldOffset = undefined
```

Table 25. MovieData extension elements

Element	Description
movieID (GUID)	Globally unique movie identifier. This is currently just a place-holder.
nextEditNum (unsigned int)	Integer that is incremented each time a new edit adds media references to the movie.
commentsLen (int)	Number of Unicode characters in comments string.
comments(wchar_t[commentsLen])	Comments
tag (wchar_t[80])	MetaData label string.
valueLen (int)	Number of unicode characters in value string.
value (wchar_t [valueLen])	MetaData value string.
tag (wchar_t[80])	MetaData label string.
valueLen (int)	Number of unicode characters in value string.
value (wchar_t [valueLen])	MetaData value string.

SegmentData

This extension is used for media management applications and to support editing clients.

```
vendorName = "Grass Valley Group"
category = "MetaData"
dataType = "SegmentData"
fieldOffset = undefined
```

Table 26. SegmentData extension elements

Element	Description
mediaID (GUID)	Globally unique media identifier. This is currently just a placeholder.
name (char[80])	Segment name string.
movieName(char[80])	Name of the movie that originally contained this segment.
editNum (unsigned int)	ID used to identify synchronous segments.

The PdrExtensionInfo Data Structure

```
#define PDR_EXTENSION_LABEL_LEN 32
#define PDR_EXTENSION_DESCRIPTION_LEN 80

typedef struct
{
MovieToken movie; /* The token for the movie with which the extension is to be regis-
                  tered. */

TrackToken track; /* The token for a track with which the extension is to be associ-
                  ated. The data can be associated with the movie itself by using 0
                  for the track value. */

MediaToken media; /* A media segment with which the extension is to be associated.
                  The data can be associated with the movie or track by using
                  PdrNullMediaToken for the media value. */

int fieldOffset; /* Number of fields from the start of the segment/track/movie.
                  Extensions influence a single point of time or a range of time
                  starting at fieldOffset. */

char vendorName[PDR_EXTENSION_LABEL_LEN+1]; /* Name of the software vendor which
                  defined the specific dataClass and dataType. For example: "Light-
                  works", "EditStar", or "Grass Valley Group". */

char dataClass[PDR_EXTENSION_LABEL_LEN+1]; /* Class this type of extension data falls
                  into. This field may be used by client applications to exclude
                  certain classes of extensions. For example an extension whose cat-
                  egory is "AudioMixEffect" may be ignored by clients that don't use
                  audio. */

char dataType[PDR_EXTENSION_DESCRIPTION_LEN+1]; /* Identifies the extension data
                  structure */

char dataName[PDR_EXTENSION_DESCRIPTION_LEN+1]; /* This string allows you to give the
                  extension a meaningful name. */

char dataVersion[PDR_EXTENSION_LABEL_LEN+1]; /* Anticipates future changes to a par-
                  ticular extension data structure. */

UINT dataLen; /* Number of bytes in the extension data structure.*/

} PdrExtensionInfo;
```

Chapter **11**

Profile RS-422 Serial Control

This chapter looks at programs that perform serial communication by using wrapper functions that implement the RS-422-based Profile serial protocol. The functions with the prefix *Pp* are wrappers that contain the serial commands. For reference information on the serial communication calls, see ***Chapter 3,*** *Profile Serial Communication Protocol* of the *Profile SDK Reference Manual.*

The programs in this chapter demonstrate how to:

- browse a Profile file system remotely;
- play a movie remotely;
- send packets over a network;
- receive packets; and
- perform packet communications.

Near the end of this chapter is a listing of the RS-422 communication header file, *ppheader.h.* The function declarations in that header file specify the serial utility functions. The definitions of these functions are contained in the following examples:

- *Example 19, ppsend.c* on page 234;
- *Example 20, ppreply.c* on page 254; and
- *Example 21, ppcomm.c* on page 268.

Browsing a remote Profile file system

Example 17, ppbrowse.c demonstrates the use of the serial Pdr inventory API calls to output an entire inventory on a remote Profile video server using a RS-422 connection. This code is similar to the browse sample operating on a local Profile or a remote Profile using Ethernet. (See *Browsing a remote Profile file system* on page 223.)

Example 17 makes use of various serial programming utility functions provided in *Example 19*, *Example 20*, and *Example 21*. Tokens for dataset, group and clip enumeration are used in the low-level support utilities.

Once the connection is established with **PpOpenComm** (see *Packet communication* on page 268), the sample code function IdentifyInventory performs the majority of the work. **PpFindFirstDataset** and **PpFindNextDataset** walk through all of the available datasets. The TekPdr library equivalents of these functions are **PdrFindFirstDataset** and **PdrFindNextDataset**, respectively.

PpFindFirstGroup and **PpFindNextGroup** walk through all of the available groups within each dataset. **PdrFindFirstGroup** and **PdrFindNextGroup** are equivalents.

Similarly, **PpFindFirstMovie** (like **PdrFindFirstMovie**) and **PpFindNextMovie** (similar to **PdrFindNextMovie**) walk through each movie within a group. **PpGetMovieState** retrieves a PdrMovieState structure, which it then uses to format some basic information about a movie. The walk-through of datasets, movies and groups is straightforward, demonstrated in three nested control loops.

Example 17. ppbrowse.c

```
//
// File: ppbrowse.c
// This is a part of the Grass Valley Group Profile Source Code Samples.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// This source code is only intended as a supplement to
// Profile Development Tools and documentation of Native Protocol.
//
// Sample code to list all the movie clips in the inventory.
//

#include <stdio.h>
#include <<stdlib.h>
#include <windows.h>
#include <limits.h>
#include <string.h>
#include <tekvdr.h>
#include <profcmd.h>
#include "ppheader.h"

// Module global variables.
static VdrHandle sPort;

// String that specifies which comm port to use.
char *comm;

// Fxn: convert a FILETIME to a string.
static const char *sMonth[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

//
// Format the FILETIME date for printout.
//
void MakeTimeString(FILETIME *ft, char *string)
{
    SYSTEMTIME st;
    WORD thisYear, thisMonth;

    GetLocalTime(&st);
    thisYear = st.wYear;
    thisMonth= st.wMonth;
    FileTimeToSystemTime(ft, &st);
    if ((thisYear == st.wYear) ||
        ((thisYear == (st.wYear+1)) && (thisMonth < st.wMonth))) {
        sprintf(string, "%3s %2d %02d:%02d",
            sMonth[st.wMonth-1], st.wDay, st.wHour, st.wMinute);
    }
    else {
```

```
            sprintf(string, "%3s %2d, %4d", sMonth[st.wMonth-1], st.wDay, st.wYear);
        }

    } // MakeTimeString

    //
    // Print out usage line.
    //
    void Usage(const char* progName)
    {
        printf("Usage: %s comm_port\n", progName);

    } // Usage

    //
    // initialize the PDR100.  Report any anomalies.
    //
    BOOL SetupResources(void)
    {
        // Open the connection and port.
        if (!PpOpenComm()) {
            printf("Open comm failed.\n");
            exit(1);
        }
        if (!PpOpenPort(0, &sPort)) {
            printf("Could not open port.\n");
            exit(1);
        }
        return TRUE;

    } // SetupResources
    //
    // Walk through the movie inventory and print it out.
    //
    void IdentifyInventory(void)
    {
        char create[16];
        char last[16];
        BOOL  dsetGood, grpGood;

        static char dataset[PDR_MAX_DSET_NAME_LEN+1];
        static char name[PDR_MAX_MOVIE_NAME_LEN+1];
        static char group[PDR_MAX_GROUP_NAME_LEN+1];
        static char startName[PDR_MAX_COMPLEX_MEDIA_NAME_LEN+1];

        //
        // Find the first known dataset.  The name is placed in the buffer dataset.
        // process the dataset information
        //
        if (!PpFindFirstDataset(dataset)) {
            printf("Cannot get first dataset.\n");
            return;
        }
        dsetGood = TRUE;
        while (ppDsetTok > 0 && dsetGood) {

            // Now look for groups within this dataset.
            grpGood = TRUE;
            strcpy(startName, dataset);

            // Find the first group within the dataset.
            // Returns the group name in the buffer "group".
            if (!PpFindFirstGroup(startName, group)) {
```

```
    printf("Cannot get the first group in %s\n", startName);
    return;
}

// Process the group information.
while (ppGrpTok > 0 && grpGood) {
    printf("%s/%s\n", dataset, group);

    // Start build the complex movie name.
    sprintf(startName, "%s/%s/", dataset, group);

    // Find the first movie name within this group.
    if (!PpFindFirstMovie(startName, name)) {
        printf("Cannot get the first movie name.\n");
        return;
    }

    // Process the movie information.
    while (ppMovieTok > 0) {
        // State structure found in pdrtypes.h
        PdrMovieState state;

        printf("         /%s\n", name);

        // Complete building the complex movie name.
        sprintf(startName, "%s/%s/%s", dataset, group, name);

        // Get information about a stored movie without opening it.
        // StartName is now the complex Movie name.
        if (!PpGetMovieState(startName, &state)) {
            printf("Cannot get state of %s\n", startName);
            return;
        }

        // Print out the status of the movie.
        printf("           Length- min %d, max %d\n",
                state.minLength, state.maxLength);
        MakeTimeString(&state.createTime, create);
        MakeTimeString(&state.lastChangedTime, last);
        printf("           Time- create %s, changed %s\n", create, last);
        printf("           #Tracks- v %d, a %d, tc %d\n",
               state.numV, state.numA, state.numT);
        printf("           Attr.- OpenExclusive %s,\n",
               (state.attributes & PdrOpenExclusive)? "Yes":"No");
        printf("           RO %s, Cntlr %s, Locked %s,\n",
               (state.attributes & PdrReadOnly)? "Yes" : "No",
               (state.attributes & PdrControlRO)? "Yes" : "No",
               (state.attributes & PdrLocked)? "Yes" : "No");

        // Find the next movie in this group.
        if (!PpFindNextMovie(name)) {
            printf("Cannot get next movie.\n");
            return;
        }
        if (!name[0]) {
            break;
        }

    }

    // Release the movie enumeration token used to track the current
    // position within the an enumeration list.
    if (!PpCloseFind(ppMovieTok)) {
```

```
                printf("Cannot close movie find\n");
                return;
            }
            if (!PpFindNextGroup(group)) {
                printf("Cannot find next group\n");
                return;
            }
            if (!group[0]) {
                grpGood = FALSE;
            }
        }

        printf(" \n");
        if (!PpCloseFind(ppGrpTok)) {
            printf("Cannot close group find\n");
            return;
        }

        // Look for the next dataset.
        if (!PpFindNextDataset(dataset)) {
            printf("Can't find next dataset\n");
            return;
        }
        if (!dataset[0]) {
            dsetGood = FALSE;
        }
    }

    if (!PpCloseFind(ppDsetTok)) {
        printf("Cannot close dataset find\n");
    }

} // IdentifyInventory


//
// The main entry point.
//
void main(int argc, char *argv[])
{
    int i = 1;

    // Read in the required comm_port.
    if (argv[i] && argv[i][0] != '-') {
        comm = argv[i];
    } else {
        Usage(argv[0]);
        exit(1);
    }

    if (SetupResources()) {
        IdentifyInventory();

        if (!PpClosePort(sPort)) {
            printf("Could not close port\n");
            exit(1);
        }
    }

} // main
```

Playing a movie remotely

Example 18, ppplay.c shows you how to play a movie using RS-422 serial communication with a Profile video server. In this example, the compression format of the video must be JPEG, and optional mark-in and mark-out points may be specified on the command line. *Example 18* is similar to the nonserial code in *Example 2, play.c* on page 58, which plays to a local Profile or over the Ethernet connection (see).

First, the appropriate resources must be allocated for later use. A connection to the remote machine is established over RS-422 with **PpOpenComm**, and a port is opened using a call to the function **PpOpenPort**.

Next, a JPEG codec resource is allocated and attached to the port with **PpAllocateResource**. The codec allows video decoding to occur on the specified port. Similarly, two audio codecs are allocated and attached to the port, also with **PpAllocateResource**, one for each stereo channel. Likewise, the video input and output resources are then allocated and attached to the port. Finally, default and scheduled events (defined with **PpDefaultEvent** and **PpScheduledEvent**) are set up to describe the connections that occur when the port is in various states.

Once the resources are obtained, we can perform the playback. The movie is opened with **PpOpenMovie** and attached using the wrapper **PpAttachOpenMovie**. Optional mark-in and mark-out points are set with the functions **PpSetMovieMarkIn** and **PpSetMovieMarkOut**.

After this, the program cues the movie for play (**PpCuePlay**). It begins playing when the video server is instructed to shuttle (**PpShuttlePlay**). After the specified duration has passed, idle mode is set with **PpIdlePort** and the movie is detached using **PpDetachMovie** and closed with **PpCloseMovie**. Finally, the port is closed (**PpClosePort**). This completes the process and the program is stopped.

If an error occurs at any step of the way, an appropriate error message is sent to the display to help troubleshoot.

Example 18. ppplay.c

```
//
// File: ppplay.c
// This sample program plays a specified JPEG clip, using
// serial communications in Profile Protocol over RS422.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// Usage: play movie_name [-i markin] [-o markout]
//
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <limits.h>
#include <string.h>
#include <tekvdr.h>
#include <profcmd.h>
#include "ppheader.h"

#define SHUTTLE_RATE  1.0
#define NUM_INPUT     0
#define NUM_OUTPUT    0
#define NUM_CODEC     0

// For demo application, we will have several resources.  Enumerate
// them for use as indeces into an array for AttachMovie calls.
// First three are Codecs.
enum CodecResEnum { VCOD, ACOD1, ACOD2, MAX_CODEC };
enum ResEnum { VIN = MAX_CODEC, VOUT, AOUT1, AOUT2, MAX_RSRC };

// Module static variables.
static VdrHandle      sPort;
static ResourceHandle sResHdls[MAX_RSRC];
static char*          spMovieName;
static UINT           sMarkIn = 0;
static UINT           sMarkOut = 0;

// String that specifies which comm port to use.
char *comm;

//
// Print out usage line.
//
void Usage(const char* progName)
{
    printf("Usage: %s movie_name comm_port [-i markin] [-o markout]\n", progName);

} // Usage


//
// Initialize the Profile.  Report any anomalies.
//
// Return TRUE if successful, otherwise FALSE.
//
BOOL SetupResources(void)
{
    printf("Starting setup...\n");

    // Open the communications port.
```

```
if (!PpOpenComm()) {
    printf("Open comm failed.\n");
    return FALSE;
}
if (!PpOpenPort(0, &sPort)) {
    printf("Could not open port.\n");
    return FALSE;
}

//
// Now, get the necessary resources for the demo.
//

// Allocate a Video Codec.
if (!PpAllocateResource(sPort, ResourceVideoCodec, NUM_CODEC, &sResHdls[VCOD])) {
    printf("Could not allocate video port.\n");
    return FALSE;
}

// Allocate two Audio Codecs.
if (!PpAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT, &sResHdls[ACOD1])) {
    printf("Could not allocate the first audio codec.\n");
    return FALSE;
}
if (!PpAllocateResource(sPort, ResourceAudioCodec, NUM_INPUT+1, &sResHdls[ACOD2])) {
    printf("Could not allocate the second audio codec.\n");
    return FALSE;
}

// Get video out and in resources.
if (!PpAllocateResource(sPort, ResourceVideoOutput, NUM_OUTPUT, &sResHdls[VOUT])) {
    printf("Could not allocate video out.\n");
    return FALSE;
}
if (!PpGetResourceConnectHandle(sPort, ResourceVideoInput, NUM_INPUT, &sResHdls[VIN])) {
    printf("Could not allocate video in.\n");
    return FALSE;
}

// Get audio resources.
if (!PpAllocateResource(sPort, ResourceAudioOutput, NUM_INPUT, &sResHdls[AOUT1])) {
    printf("Could not allocate the first audio output.\n");
    return FALSE;
}
if (!PpAllocateResource(sPort, ResourceAudioOutput, NUM_INPUT+1, &sResHdls[AOUT2])) {
    printf("Could not allocate the first audio output.\n");
    return FALSE;
}

// Set the default event.
if (!PpDefaultEvent(sPort, EventConnectResources, sResHdls[VIN],
            sResHdls[VOUT], 0, 0)) {
    printf("Could not set default event.\n");
    return FALSE;
}

// Schedule the event.
if (!PpScheduleEvent(sPort, INT_MIN, EventConnectResources, sResHdls[VCOD],
                    sResHdls[VOUT], 0, 0)) {
    printf("Could not schedule event.\n");
    return FALSE;
}
return TRUE;
```

```
} // SetupResources


//
// Play the movie clip.
//
void StartPlay(void)
{
    INT oldpos, newpos;
    MovieToken movieTok;
    MovieHandle movieHdl;
    UINT portStatusArray[8];
    UINT mask;

    // Open and attach the movie.
    if (!PpOpenMovie(spMovieName, 0, &movieTok)) {
        printf("Movie %s does not exist.\n", spMovieName);
        return;
    }
    if (!PpAttachOpenMovie(movieTok, MAX_CODEC, sResHdls, 0, ShiftAfter, &movieHdl)) {
        printf("Cannot attach the open movie.\n");
        return;
    }

    // Set markin and markout if required.
    if (sMarkIn) {
        if (!PpSetMovieMarkIn(movieHdl, sMarkIn)) {
            printf("Could not set markin.\n");
            return;
        }
    }
    if (sMarkOut) {
        if (!PpSetMovieMarkOut(movieHdl, sMarkOut)) {
            printf("Could not set markout.\n");
            return;
        }
    }

    // Cue up playback of media attached with PpAttachOpenMovie.
    if (!PpCuePlay(sPort)) {
        printf("Cannot cue play.\n");
        return;
    }

    // Begin motion playback.
    if (!PpShuttlePlay(sPort, SHUTTLE_RATE)) {
        printf("Cannot begin playback.\n");
        return;
    }

    printf("Starting playback...\n");

    // Wait while movie plays.
    // When newpos and oldpos are the same, we're done playing out.
    newpos = 0;
    mask = Dat1_GetPosition;
    do  {
        oldpos = newpos;
        Sleep(100);                         // wait 1/10th second
        if (!PpGetPortStatus(sPort, &mask, portStatusArray, 6)) {
            printf("Can not get Port status.\n");
            return;
```

```
        }
        newpos = portStatusArray[1];

    }  while (newpos > oldpos);

    // Cease play back.
    if (!PpIdlePort(sPort)) {
        printf("Cannot move to idle.\n");
        return;
    }

    // Detach the movie handle from the channel.
    if (!PpDetachMovie(movieHdl)) {
        printf("Cannot detach movie.\n");
        return;
    }

    // Close the movie.
    if (!PpCloseMovie(movieTok)) {
        printf("Cannot close movie.\n");
        return;
    }

} // StartPlay

//
// Cleanup by releasing resources and closing the control port.
//
void Cleanup(void)
{
    printf("Starting cleanup...\n");

    if (!PpClosePort(sPort)) {
        printf("Cannot close port.\n");
        return;
    }
    sPort = 0;

} // Cleanup
//
// The main entry point.
//
void main(int argc, char *argv[])
{

    BOOL rtn = TRUE;
    int i=1;

    // Read in the required movie name.
    i = 1;
    if (argv[i]) {
        spMovieName = argv[i];
    }
    else {
        Usage(argv[0]);
        exit(1);
    }
    i++;

    // Read in the required communications port.
    if (argv[i] && argv[i][0] != '-') {
        comm = argv[i];
    }
```

```
        else {
            Usage(argv[0]);
            exit(1);
        }
        i++;

        // Process optional markin and markout points.
        while (i < argc) {
            if (argv[i][0] == '-')
                switch (argv[i][1]) {
                case 'i':
                    i++;
                    sMarkIn = atoi(argv[i++]);
                    break;
                case 'o':
                    i++;
                    sMarkOut = atoi(argv[i++]);
                    break;
                default:
                    Usage(argv[0]);
                    exit(1);
                }
            else {
                Usage(argv[0]);
                exit(1);
            }
        }

        if (SetupResources()) {
            StartPlay();
        }

        Cleanup();

    } // main
```

Sending packets

Example 19, ppsend.c contains various utility functions for sending packets in Profile protocol over a serial RS-422 connection. The function declarations are shown in *ppheader.h*.

The majority of functions is this group act as wrappers around specific Profile API calls and use the lower-level reply functions, defined in *Example 20, ppreply.c*. For example, **PpGetPortStatus** issues the command to get the port status, then calls **PpGetPortStatusReply** to parse the specific reply to that command. Each wrapper function returns TRUE upon success, otherwise FALSE.

Example 19. ppsend.c

```
//
// File: ppsend.c
// This is a part of the Grass Valley Group Profile Source Code Samples.
//
/// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// This source code is only intended as a supplement to
// Profile Development Tools and documentation of Native Protocol.
//
#include <stdio.h>
#include <windows.h>
#include <tekvdr.h>
#include "ppcomm.h"
#include <profhdr.h>
#include <profcmd.h>
#include "sample.h"

#define MAX_CLIP        5

extern MovieHandle   moviehandle[MAX_CLIP];
extern EventHandle   eventhandle;
extern ConnectHandle connection;

#define MAX_RESOURCE    32

//
// Get Transaction Reply
//     cmd:
//         data1: transaction number; the command will return the reply for the given
//              transaction if it is available.
//     reply:
//         If the given transaction reply is ready, the "transaction reply" for the
//       command which started the transaction is returned (see details for each
//      command).
//      If the given transaction reply is unavailable, the following is returned:
//             byte0: 0x02 (Stx)
//             byte1: byte count (2)
//             byte2: 0x00
//             byte3: 0xff
//             byte4: checksum
//
BOOL PpGetTransactionReply(UINT xactno)
{
    char buffer[256];

    buffer[PRO_DATA1_INDEX] = xactno;
```

```
    if (!PpSendCommand(buffer, 1, Cmd1_SystemAccess, Cmd2_GetTransactionReply)) {
        return FALSE;
    }

    return TRUE;

} // PpGetTransactionReply

//
// Get Port Status
//     cmd:
//         data1-4: handle of port for which status is to be returned.
//         data5-6: mask indicating what status information is to be returned:
//             bit0: play/record state
//             bit1: current position (field number) along timeline
//             bit2: movie at current position
//             bit3: motion-play mode
//             bit4: still-play mode
//             bit5: current play rate
//             bit6 ..  bit15: reserved
//     reply:
//         data1-2: a copy of the request mask from command data5-6
//             (if the bit is not implemented, it will be cleared)
//         data3-n: status values requested
//             (value corresponding to the lowest bit in the mask first):
//             Play/record state (1 byte): one of the following values:
//                 2: record-cued
//                 3: jog-play
//                 4: jog-record
//                 5: shuttle-play
//                 6: shuttle-record
//             Current position along timeline (integer, 4 bytes)
//             Current movie on timeline (4 bytes)
//             Motion-play mode (1 byte): one of the following values:
//                 0: normal play
//                 1: loop play
//                 2: bounce play
//                 3: limited play
//                 4: slaved play
//             Still-play mode (1 byte): one of the following values:
//                 0: still play by field
//                 1: still play by frame
//             Current play rate (float, 4 bytes)
//
BOOL PpGetPortStatus(VdrHandle portHandle, UINT* mask, UINT values[], UINT numvals)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    *((USHORT *)(&buffer[PRO_DATA1_INDEX+4])) = (USHORT)(*mask);
    if (!PpSendCommand(buffer, 6, Cmd1_PortAccess, Cmd2_GetPortStatus)) {
        return FALSE;
    }

    if (!PpGetPortStatusReply(mask, values, numvals)) {
        return FALSE;
    }

    return TRUE;

} // PpGetPortStatus
```

```
//
// Open Port
//     cmd:
//         (no agruments are needed for this command.  The following is optional
//         data1-8: Named Configuration Space
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5: handle for port.  If port cannot be opened, this value will be 0.
//
BOOL PpOpenPort(char* ncsname, VdrHandle* portHandle)
{
    char buffer[16];
    UINT len;

    if (ncsname) {
        strcpy(&buffer[PRO_DATA1_INDEX], ncsname);
        len = strlen(ncsname)+1;
    }
    else {
        len = 0;
    }

    if (!PpSendCommand(buffer, len, Cmd1_PortAccess, Cmd2_OpenPort)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpOpenPortReply(portHandle)) {
        return FALSE;
    }

    return TRUE;

} // PpOpenPort


//
// Close Port
//     cmd:
//         data1-4: handle of port
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpClosePort(VdrHandle portHandle)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;

    if (!PpSendCommand(buffer, 4, Cmd1_PortAccess, Cmd2_ClosePort)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_PortAccess, Cmd2_ClosePort)) {
        return FALSE;
```

```
    }

    return TRUE;

} // PpClosePort


//
// Close Movie
//
//     cmd:
//         data1-4:    movie token
//
//     reply:
//         0:    OK
//         1-255 (error condition)
//
//
BOOL PpCloseMovie(MovieToken movie)
{
    char buffer[16];

    *((MovieToken *)(&buffer[PRO_DATA1_INDEX])) = movie;

    if (!PpSendCommand(buffer, 4, Cmd1_PdrMovieAccess, Cmd2_PdrCloseMovie)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpPdrErrorNumReply(Cmd1_PdrMovieAccess, Cmd2_PdrCloseMovie)) {
        return FALSE;
    }
    return TRUE;

} // PpCloseMovie

//
// Idle Port
//     cmd:
//         data1-4: handle of port
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpIdlePort(VdrHandle portHandle)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    if (!PpSendCommand(buffer, 4, Cmd1_PortAccess, Cmd2_Idle)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_PortAccess, Cmd2_Idle)) {
        return FALSE;
    }
    return TRUE;

} // PpIdlePort
```

```
//
// Cue Play
//     cmd:
//         data1-4: handle of port
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpCuePlay(VdrHandle portHandle)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    if (!PpSendCommand(buffer, 4, Cmd1_PortAccess, Cmd2_CuePlay)) {
        return FALSE;
    }

    // Synchronous operation.
    if ( !PpErrorNumReply(Cmd1_PortAccess, Cmd2_CuePlay)) {
        return FALSE;
    }
    return TRUE;

} // PpCuePlay


//
// Shuttle Play
//     cmd:
//         data1-4: handle of port
//         data5-8: shuttle speed (float); if no bytes 5-8, speed 1.0 assumed.
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpShuttlePlay(VdrHandle portHandle, double rate)
{
    char    buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    *((float *)(&buffer[PRO_DATA1_INDEX+4])) = (float)rate;
    if (!PpSendCommand(buffer, 8, Cmd1_PortAccess, Cmd2_ShuttlePlay)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_PortAccess, Cmd2_ShuttlePlay)) {
        return FALSE;
    }
    return TRUE;

} // PpShuttlePlay
```

```
//
// Set Movie Mark In
//     cmd:
//         data1-4:  movie handle
//         data5-data8:  mark-in field number
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpSetMovieMarkIn(MovieHandle movieHandle, UINT markin)
{
    char buffer[16];

    *((MovieHandle *)(&buffer[PRO_DATA1_INDEX])) = movieHandle;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+4])) = markin;

    if (!PpSendCommand(buffer, 8, Cmd1_AttachedMovieAccess, Cmd2_SetMovieMarkIn)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_AttachedMovieAccess, Cmd2_SetMovieMarkIn)) {
        return FALSE;
    }
    return TRUE;

} // PpSetMovieMarkIn

//
// Set Movie Mark Out
//     cmd:
//         data1-4:  movie handle
//         data5-data8:  mark-out field number
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpSetMovieMarkOut(MovieHandle movieHandle, UINT markout)
{
    char buffer[16];

    *((MovieHandle *)(&buffer[PRO_DATA1_INDEX])) = movieHandle;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+4])) = markout;

    if (!PpSendCommand(buffer, 8, Cmd1_AttachedMovieAccess, Cmd2_SetMovieMarkOut)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_AttachedMovieAccess, Cmd2_SetMovieMarkOut)) {
        return FALSE;
    }

    return TRUE;

} // PpSetMovieMarkOut
//
```

```
// Attach Movie with Marks
//     cmd:
//         data1-n:  ASCIZ name
//         data(n+1)-(n+4):  mark-in field number
//         data(n+5)-(n+8):  mark-out field number
//         data(n+9)-(n+13):  handle of port to which movie is to be attaached
//         --- if no more arguments ->  attached at end of list w/ appropriate shift
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5:  handle of attached movie (0 if failure)
//
BOOL PpAttachMovie_Marks(VdrHandle portHandle, char* nambuf, UINT markin,
                         UINT markout, MovieHandle* movieHandle)
{
    char buffer[256];
    // Include trailing null.
    UINT len = strlen(nambuf) + 1;

    if (len > 235)  {
        // Too much.
        return FALSE;
    }
    strcpy(&buffer[PRO_DATA1_INDEX], nambuf);
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len])) = markin;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+4])) = markout;
    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX+len+8])) = portHandle;

    if (!PpSendCommand(buffer, len+12, Cmd1_AttachedMovieAccess,
                       Cmd2_AttachMovieWithMarks)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpAttachMovieReply(movieHandle, Cmd2_AttachMovieWithMarks)) {
        return FALSE;
    }

    return TRUE;

} // PpAttachMovie_Marks


//
// Detach Movie
//     cmd:
//         data1-4:  movie handle
//         --- if no more arguments ->  use appropriate shift
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5:  handle of attached movie (0 if failure)
//
BOOL PpDetachMovie(MovieHandle movieHandle)
{
    char buffer[16];

    *((MovieHandle *)(&buffer[PRO_DATA1_INDEX])) = movieHandle;
    if (!PpSendCommand(buffer, 4, Cmd1_AttachedMovieAccess, Cmd2_DetachMovie)) {
        return FALSE;
    }
```

```
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_AttachedMovieAccess, Cmd2_DetachMovie)) {
        return FALSE;
    }
    return TRUE;

} // PpDetachMovie


//
// Detach Media
//     cmd:
//         data1-4:  movie handle
//         --- if no more arguments ->  use appropriate shift
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5:  handle of attached movie (0 if failure)
//
BOOL PpDetachMedia(MediaHandle mediaHandle, UINT shift)
{
    char buffer[16];

    *((MovieHandle *)(&buffer[PRO_DATA1_INDEX])) = mediaHandle;
    buffer[PRO_DATA1_INDEX+4] = (UCHAR)shift;
    if (!PpSendCommand(buffer, 5, Cmd1_AttachedMediaAccess, Cmd2_DetachMedia)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_AttachedMediaAccess, Cmd2_DetachMedia)) {
        return FALSE;
    }
    return TRUE;

} // PpDetachMedia


//
// Allocate Resource
//     cmd:
//         data1-4: handle of port
//         data5: type of resource to be allocated; one of
//             0: audio codec
//             1: video codec
//             2: video input
//             3: video output
//             4: LTC input
//             5: VITC input
//             6: timecode generator
//             7: LTC output
//             8: VITC output
//             9: timecode recorder
//             a: audio output
//         data6: physical number of resource to be allocated.
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5: handle to allocated resource (0 if resource could not be allocated)
//
BOOL PpAllocateResource(VdrHandle portHandle, UCHAR type, UINT num, ResourceHandle* rscHandle)
{
```

```
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    buffer[PRO_DATA1_INDEX+4] = type;
    buffer[PRO_DATA1_INDEX+5] = (UCHAR)num;

    if (!PpSendCommand(buffer, 6, Cmd1_ResourceAccess, Cmd2_AllocateResource)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpResourceHandleReply(rscHandle, Cmd2_AllocateResource)) {
        return FALSE;
    }

    return TRUE;

} // PpAllocateResource


//
// Get Resource Connect Handle
//     cmd:
//         data1-4: handle of port
//         data5: type of resource; one of
//             0: audio codec
//             1: video codec
//             2: video input
//             3: video output
//             4: LTC input
//             5: VITC input
//             6: timecode generator
//             7: LTC output
//             8: VITC output
//             9: timecode recorder
//             a: audio output
//         data6: physical number of resource to be allocated.
//     reply:
//         Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5: connect handle to resource (0 if resource has no connection capability)
//
BOOL PpGetResourceConnectHandle(VdrHandle portHandle, UCHAR type, UINT num, ResourceHandle*
rscHandle)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = portHandle;
    buffer[PRO_DATA1_INDEX+4] = type;
    buffer[PRO_DATA1_INDEX+5] = (UCHAR)num;

    if (!PpSendCommand(buffer, 6, Cmd1_ResourceAccess, Cmd2_GetResConnectHandle)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpResourceHandleReply(rscHandle, Cmd2_GetResConnectHandle)) {
        return FALSE;
    }
    return TRUE;

} // PpGetResourceConnectHandle
```

```
//
// Default Event
//
//     cmd:
//          data1-4:    Port handle
//          data-5:     Event type:
//                            2: mix audio
//                            3: connect Resources
//                            4: set timecode generator time
//                            5:  set timecode generator user bits
//          data6-9   source resource handle
//              [This section is dependant on the type of event as follows:]
//          data 10-13: Connect Resource:    destination resource handle
//              Mix Audio:       destination resource handle
//                        14-17  mix level (0.0...1.0)
//                        18-21: number of fields for duration of mix
//              Set TC Gen Time: timecode
//                        14-17: field number at which to activate
//              Set TC Gen bits: use bits pattern
//
BOOL DefaultEvent(VdrHandle vdrHandle, UINT evttyp, ResourceHandle srcHandle, ResourceHandle
destHandle, double mix, UINT noflds)
{
    char buffer[24];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = vdrHandle;
    buffer[PRO_DATA1_INDEX+4] = evttyp;
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+5])) = srcHandle;
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+9])) = destHandle;

    if (!PpSendCommand(buffer, 13, Cmd1_EventAccess, Cmd2_DefaultEvent)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_EventAccess, Cmd2_DefaultEvent)) {
        return FALSE;
    }

    return TRUE;

} // PpDefaultEvent
//
// Schedule Event
//     cmd:
//         data 1-4:    port handle
//              5-8:    field number for the event timing
//              9:        Event type broken down as follows:
//                         3: Connect Resource
//                         2: Mix Audio
//                         4: Set Timecode Generator Time
//                         5: Set Timecode Generator User bits
//            10-13:    source resource handle
//                     [this next section is dependaant on the type of event as follows]
//            14-17:    Connect Resource: destination resource handle
//                    Mix Audio:          destination resource handle
//                      18-21:            mix level (0.0...1.0)
//                      22-25:            number of fields for duration of mix
//                    Set TC Gen Time:    timecode
//                    Set TC Gen bits:    use bits pattern
//
//
BOOL PpScheduleEvent(VdrHandle vdrHandle, INT fldno, UINT evttyp,
                     ResourceHandle srcHandle, ResourceHandle destHandle,
```

```
                    double mix, UINT noflds)
{
    char buffer[24];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = vdrHandle;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+4])) = fldno;
    buffer[PRO_DATA1_INDEX+8] = evttyp;
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+9])) = srcHandle;
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+13])) = destHandle;

    if (!PpSendCommand(buffer, 17, Cmd1_EventAccess, Cmd2_ScheduleEvent)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpGetScheduleEventReply(&eventhandle)) {
        return FALSE;
    }
    return TRUE;

} // PpScheduleEvent


//
// Open Movie
//
//     cmd:
//         data 1-n:    complex move name
//         data (n+1)-(n+4): flags
//                           PdrExclusive - 0x0040 (else 0)
//
BOOL PpOpenMovie(char *moviename, UINT flags, MovieToken* movietoken)
{
    char buffer[256];
    UINT len = strlen(moviename) + 1;

    strcpy(&buffer[PRO_DATA1_INDEX], moviename);
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len])) = flags;

    if (!PpSendCommand(buffer, len+4, Cmd1_PdrMovieAccess, Cmd2_PdrOpenMovie)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpGetOpenMovieReply(movietoken)) {
        return FALSE;
    }
    return TRUE;

} // PpOpenMovie


//
// Attach Open Movie
//
//     cmd:
//         data1-4:    movie token from OpenMovie
//         data5:        number of resources to use
//         data6-(n):    resource handles to use (maxiumum of 32)
//             [if there are no arguments after the port handle, the movie will be
//            attached at the end of the list with a shift appropriate to the current position]
//         data(n+1)-(n+4): handle of movie before which this movie should be attached
//                     Use value 0 to attach at the end of the list of movies.
//         data(n+5):    shift move, one of
```

```
//                          0: shift previously attached movies before newly attached movie
//                          1: shift previously attached movies after newly attached movie
//
BOOL PpAttachOpenMovie(MovieToken movietoken, UINT num, ResourceHandle* rsrchands,
        UINT after, UINT shift, MovieHandle* moviehandle)
{
    char buffer[256];
    int len;
    UINT i;

    *((MovieToken *)(&buffer[PRO_DATA1_INDEX])) = movietoken;
    if (num > MAX_RESOURCE) {
        printf("Resources exceed maximun number.\n");
        return FALSE;
    }
    buffer[PRO_DATA1_INDEX+4] = num;
    len = 5;
    for (i = 0; i < num; i++) {
        *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+len])) = (ResourceHandle)(*rsrchands++);
        len = len+4;
    }
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len])) = after;
    buffer[PRO_DATA1_INDEX+len+4] = (UCHAR)shift;
    buffer[PRO_DATA1_INDEX+len+5] = MarkLongest;                // mark mode

    if (!PpSendCommand(buffer, len+6, Cmd1_AttachedMovieAccess, Cmd2_AttachOpenMovie)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpGetAttachOpenMovieReply(moviehandle)) {
        return FALSE;
    }
    return TRUE;

} // PpAttachOpenMovie


//
// Get Standard
//
//     cmd:
//         (no data bytes)
//     reply:
//         data1:     the system standard
//
BOOL PpGetStandard(ConnectHandle* connection)
{
    char buffer[16];

    if (!PpSendCommand(buffer, 0, Cmd1_SystemAccess, Cmd2_GetStandard)) {
        return FALSE;
    }
    if (!PpGetStandardReply(connection)) {
        return FALSE;
    }
    return TRUE;

} // PpGetStandard


//
// Cue Record
```

```
//
// cmd:
//     data1-4:    Port handle
//
BOOL PpCueRecord(VdrHandle porthandle)
{
    char buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = porthandle;
    if (!PpSendCommand(buffer, 4, Cmd1_PortAccess, Cmd2_CueRecord)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_PortAccess, Cmd2_CueRecord)) {
        return FALSE;
    }
    return TRUE;

} // PpCueRecord


//
// Shuttle Record
//
// cmd:
//     data1-4:    Port handle
//     data5-8:    shuttle speed (float); if no bytes 5-8, assume speed 1.0
//
BOOL PpShuttleRecord(VdrHandle porthandle, double rate)
{
    char    buffer[16];

    *((VdrHandle *)(&buffer[PRO_DATA1_INDEX])) = porthandle;
    *((float *)(&buffer[PRO_DATA1_INDEX+4])) = (float)rate;

    if (!PpSendCommand(buffer, 8, Cmd1_PortAccess, Cmd2_ShuttleRecord)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpErrorNumReply(Cmd1_PortAccess, Cmd2_ShuttleRecord) ) {
        return FALSE;
    }
    return TRUE;

} // PpShuttleRecord


//
// Close Find
// cmd:
//      data1-4:    token from Find First Dataset, Group, or Movie calls
//
BOOL PpCloseFind(EnumToken tok)
{
    char buffer[16];

    *((EnumToken *)(&buffer[PRO_DATA1_INDEX])) = tok;

    if (!PpSendCommand(buffer, 4, Cmd1_PdrMovieAccess, Cmd2_PdrCloseFind)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpPdrErrorNumReply(Cmd1_PdrMovieAccess, Cmd2_PdrCloseFind)) {
```

```
        return FALSE;
    }

    return TRUE;

} // PpCloseFind


//
// Movie Exist
//
// cmd:
//     data1-n:    complex movie name
//
BOOL PpMovieExist(char* moviename, UINT* exist)
{
    char buffer[256];
    UINT len = strlen(moviename) + 1;

    strcpy(&buffer[PRO_DATA1_INDEX], moviename);

    if (!PpSendCommand(buffer, len+4, Cmd1_PdrMovieAccess, Cmd2_PdrMovieExist)) {
        return FALSE;
    }

    // Synchronous operation.
    if (!PpGetMovieExistReply(exist)) {
        return FALSE;
    }
    return TRUE;

} // PpMovieExis


//
// Find First Dataset
//
// cmd:
//     (not data bytes)
//
//
BOOL PpFindFirstDataset(char* dataset)
{
    char buffer[16];

    if (!PpSendCommand(buffer, 0, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstDataset)) {
        return FALSE;
    }
    if (!PpGetFirstDatasetReply(dataset)) {
        return FALSE;
    }
    return TRUE;

} // PpFindFirstDataset


//
//
// Find First Group
//
// cmd:
//     data1-n:    Datset name to walk through
//
```

```
BOOL PpFindFirstGroup(char* dataset, char* group)
{
    char buffer[32];
    UINT len = strlen(dataset) + 1;

    strcpy(&buffer[PRO_DATA1_INDEX], dataset);

    if (!PpSendCommand(buffer, len, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstGroup)) {
        return FALSE;
    }
    if (!PpGetFirstGroupReply(group)) {
        return FALSE;
    }
    return TRUE;

} // PpFindFirstGroup


//
// Find First Movie
//
// cmd:
//     data1-n:    dataset/group name to walk through
//
BOOL PpFindFirstMovie(char* datagrp, char* name)
{
    char buffer[256];
    UINT len = strlen(datagrp) + 1;

    strcpy(&buffer[PRO_DATA1_INDEX], datagrp);

    if (!PpSendCommand(buffer, len, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstMovie)) {
        return FALSE;
    }
    if (!PpGetFirstMovieReply(name)) {
        return FALSE;
    }
    return TRUE;

} // PpFindFirstMovie


//
// Find Next Dataset
//
// cmd:
//     data1-4:    token from subsquent calls
//
BOOL PpFindNextDataset(char* dataset)
{
    char buffer[16];

    *((EnumToken *)(&buffer[PRO_DATA1_INDEX])) = dset;

    if (!PpSendCommand(buffer, 4, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextDataset)) {
        return FALSE;
    }
    if (!PpGetNextDatasetReply(dataset)) {
        return FALSE;
    }

    return TRUE;
```

```
} // PpFindNextDataset


//
// Find Next Group
//
// cmd:
//     data1-4:    token from subsquent calls
//
BOOL PpFindNextGroup(char* group)
{
    char buffer[32];

    *((EnumToken *)(&buffer[PRO_DATA1_INDEX])) = grp;

    if (!PpSendCommand(buffer, 4, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextGroup)) {
        return FALSE;
    }
    if (!PpGetNextGroupReply(group)) {
        return FALSE;
    }
    return TRUE;

} // PpFindNextGroup


//
// Find Next Movie
//
// cmd:
//     data1-4:    token from subsquent calls
//
BOOL PpFindNextMovie(char* name)
{
    char buffer[16];

    *((EnumToken *)(&buffer[PRO_DATA1_INDEX])) = movie;

    if (!PpSendCommand(buffer, 4, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextMovie)) {
        return FALSE;
    }
    if (!PpGetNextMovieReply(name)) {
        return FALSE;
    }
    return TRUE;

} // PpFindNextMovie


//
// Get Movie State
//
// cmd:
//     data1-n:    Complex movie name.
//
BOOL PpGetMovieState(char* name, PdrMovieState* state)
{
    char    buffer[256];
    UINT    len = strlen(name) + 1;

    strcpy(&buffer[PRO_DATA1_INDEX], name);

    if (!PpSendCommand(buffer, len, Cmd1_PdrMovieAccess, Cmd2_PdrGetMovieState)) {
```

```
        return FALSE;
    }
    if (!PpGetMovieStateReply(state)) {
        return FALSE;
    }
    return TRUE;

} // PpGetMovieState


//
// Get Next Track Token
//
// cmd:
//     data1-4:    Movie token.
//     data5-6:    Track Token.
//
// reply:
//     data1-2:    Next track token.
//
BOOL PpGetNextTrack(MovieToken movietoken, TrackToken* tracktoken)
{
    char buffer[16];

    *((MovieToken *)(&buffer[PRO_DATA1_INDEX])) = movietoken;
    *((TrackToken *)(&buffer[PRO_DATA1_INDEX]+4)) = (USHORT)(*tracktoken);

    if (!PpSendCommand(buffer, 6, Cmd1_StoredMediaAccess, Cmd2_GetNextTrack)) {
        return FALSE;
    }
    if (!PpGetNextTrackReply(tracktoken)) {
        return FALSE;
    }
    return TRUE;

} // PpGetNextTrack

//
// Get Null Media Token
//
// cmd:
//     (No command data bytes required
//
// reply:
//     data1-8:    Null media token
//
BOOL PpGetNullMediaToken(MediaToken* mediatoken)
{
    char buffer[16];

    if (!PpSendCommand(buffer, 0, Cmd1_StoredMediaAccess, Cmd2_GetNullMediaToken)) {
        return FALSE;
    }
    if (!PpGetNullMediaTokenReply(mediatoken)) {
        return FALSE;
    }
    return TRUE;

} // PpGetNullMediaToken
//
// Get Media File Path
//
BOOL PpGetMediaPath(UINT* mask, MediaToken mediatoken, char* mediapath)
```

```
{
    char buffer[16];

    *((USHORT *)(&buffer[PRO_DATA1_INDEX])) = (USHORT)(*mask);
    *((MediaToken *)(&buffer[PRO_DATA1_INDEX+2])) = mediatoken;

    if (!PpSendCommand(buffer, 10, Cmd1_StoredMediaAccess, Cmd2_GetStoredMediaStatus)) {
        return FALSE;
    }
    if (!PpGetMediaPathReply(mask, mediapath)) {
        return FALSE;
    }
    return TRUE;

} // PpGetMediaPath


//
// Attach Media
//
//  cmd:
//      data1-n:            name of media (Null terminated)
//      data(n+1)-(n+4):    handle of resource to wich media is to be attached
//      data(n+5)-(n+8):    Optional number of fields duration
//      data(n+9)-(n+12):    handle of media before which this media is attached
//      data(n+13):           Shiftmode
//
//  reply:
//      data1:        transaction number
//      data2-5:    handle of attached media (0, if media could not be attachted
//
//
//
BOOL PpAttachMedia(char* namebuf, ResourceHandle rsrc, UINT duration,
                   MediaHandle* mediahandle, UINT after, UINT shift)
{
    char buffer[256];

    // Include trailing null.
    UINT len = strlen(namebuf) + 1;

    if (len > 235) {
        // Too much.
        return FALSE;
    }

    strcpy(&buffer[PRO_DATA1_INDEX], namebuf);
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+len])) = rsrc;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+4])) = duration;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+8])) = after;
    buffer[PRO_DATA1_INDEX+len+12] = (UCHAR)shift;

    if (!PpSendCommand(buffer, len+13, Cmd1_AttachedMediaAccess, Cmd2_AttachMedia)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpAttachMediaReply(mediahandle, Cmd2_AttachMedia)) {
        return FALSE;
    }
    return TRUE;

} // PpAttachMedia
```

```
//
// Attach Media With Marks
//
// cmd:
//     data1-n:            name of media (Null terminated)
//     data(n+1)-(n+4):    mark-in
//     data(n+5)-(n+8):    mark-out
//     data(n+9)-(n+12):   handle of resource to wich media is to be attached
//     data(n+13)-(n+16):  Optional number of fields duration
//     data(n+17)-(n+20):  handle of media before which this media is attached
//     data(n+21):         Shiftmode
//
// reply:
//     data1:   transaction number
//     data2-5: handle of attached media (0, if media could not be attachted
//
BOOL PpAttachMediaWithMarks(char* namebuf, ResourceHandle rsrc, UINT duration,
                            MediaHandle* mediahandle, UINT after, UINT shift,
                            UINT markin, UINT markout)
{
    char buffer[256];

    // Include trailing null.
    UINT len = strlen(namebuf) + 1;

    if (len > 235) {
        // Too much.
        return FALSE;
    }

    strcpy(&buffer[PRO_DATA1_INDEX], namebuf);
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len])) = markin;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+4])) = markout;
    *((ResourceHandle *)(&buffer[PRO_DATA1_INDEX+len+8])) = rsrc;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+12])) = duration;
    *((UINT *)(&buffer[PRO_DATA1_INDEX+len+16])) = after;
    buffer[PRO_DATA1_INDEX+len+20] = (UCHAR)shift;

    if (!PpSendCommand(buffer, len+21, Cmd1_AttachedMediaAccess, Cmd2_AttachMediaWithMarks)) {
        return FALSE;
    }
    // Synchronous operation.
    if (!PpAttachMediaReply(mediahandle, Cmd2_AttachMediaWithMarks)) {
        return FALSE;
    }
    return TRUE;

} // PpAttachMediaWithMarks


//
// Get Next Media Token
//
// cmd:
//     data1-4:    Movie token.
//     data5-6:    Track Token
//     data7-14:    Media Token
//
// reply:
//     data1-8:    Next Media token
//
BOOL PpGetNextMediaToken(MovieToken movietoken, TrackToken tracktoken,
```

```
                        MediaToken* mediatoken)
{
    char buffer[32];

    *((MovieToken *)(&buffer[PRO_DATA1_INDEX])) = movietoken;
    *((TrackToken *)(&buffer[PRO_DATA1_INDEX]+4)) = tracktoken;
    *((MediaToken *)(&buffer[PRO_DATA1_INDEX]+6)) = *mediatoken;

    if (!PpSendCommand(buffer, 14, Cmd1_StoredMediaAccess, Cmd2_GetNextMediaToken)) {
        return FALSE;
    }
    if (!PpGetNextMediaReply(mediatoken)) {
        return FALSE;
    }
    return TRUE;

} // PpGetNextMediaToken


//
// Is Media Token Null
//
// cmd:
//     data1-8:    Media token.
//
// reply:
//     data1:      0= True, it is a null media token
//                 1= False, it is not a null media Token Next
//
BOOL PpIsMediaTokenNull(MediaToken* mediatoken)
{
    char buffer[16];

    *((MediaToken *)(&buffer[PRO_DATA1_INDEX])) = *mediatoken;

    if (!PpSendCommand(buffer, 8, Cmd1_StoredMediaAccess, Cmd2_GetNullMediaToken)) {
        return FALSE;
    }
    if (!PpIsMediaTokenNullReply()) {
        return FALSE;
    }
    return TRUE;

} // PpIsMediaTokenNull
```

Receiving packets

Example 20, ppreply.c contains various utility functions to facilitate the reception of packets in Profile protocol over a serial RS-422 connection. The function declarations are shown in *ppheader.h*. **PpGetReply** uses the generic function PpRcvReply (defined in *ppcomm.c*) to get a synchronous transaction reply. As in *Example 19, ppsend.c*, the majority of functions in this group act as wrappers around specific Profile API calls.

Example 20. ppreply.c

```
//
// File: ppreply.c
// This is a part of the Grass Valley Group Profile Source Code Samples.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// This source code is only intended as a supplement to
// Profile Development Tools and documentation of Native Protocol.
//
#include <stdio.h>
#include <windows.h>
#include <tekvdr.h>
#include "ppcomm.h"
#include "vdrtypes.h"
#include <profhdr.h>
#include <profcmd.h>
#include "sample.h"

//
// Get the reply packet.
//
BOOL PpGetReply(UCHAR* buffer, UINT* bcnt, UINT cmd1, UINT cmd2)
{
    UINT tno;
    UCHAR tbuf[8];

    if (!PpRcvReply(buffer, bcnt, cmd1, cmd2)) {
        return FALSE;
    }

    if (buffer[PRO_STX_INDEX] != Stx_Ack) {
        return TRUE;
    }
    // This is only for synchronous receive operations.

    // Pickup transaction number.
    tno = (UINT)buffer[PRO_CMD1_INDEX];
    tbuf[PRO_DATA1_INDEX] = tno;
    do  {
        if (buffer[PRO_CMD1_INDEX] != tno) {
            printf("Error on Synchronous Transaction Number Mismatch\n");
            return FALSE;
        }
        if (!PpSendCommand(tbuf, 1, Cmd1_SystemAccess, Cmd2_GetTransactionReply)) {
            printf("Error on Synchronous Transaction Request\n");
            return FALSE;
        }
        Sleep(15);  // Allow time for reply.
```

```
        if (!PpRcvReply(buffer, bcnt, cmd1, cmd2)) {
            printf("Error on Synchronous Transaction Reply\n");
            return FALSE;
        }
    } while (buffer[0] == Stx_Ack);

    return TRUE;

} // PpGetReply

//
// Get Transaction Reply
//
// If the given transaction reply is ready, the "transaction reply" for the
// command that started the transaction is returned (see details for each
// command). If the transaction reply is unavailable, the following is returned:
//     byte0: 0x02 (Stx)
//     byte1: byte count (2)
//     byte2: 0x00
//     byte3: 0x00
//     byte4: checksum
//
BOOL PpTransactionReply(UINT xactno)
{
    char buffer[256];
    UINT len;

    buffer[PRO_DATA1_INDEX] = xactno;

    if (!PpGetReply(buffer, &len, Cmd1_SystemAccess, Cmd2_GetTransactionReply)) {
        return FALSE;
    }
    return TRUE;

} // PpTransactionReply
//
// Get Port Status Reply
//
//     data1-2: a copy of the request mask from command data5-6
//         (if the bit is not implemented, it will be cleared)
//         bit0: play/record state
//         bit1: current position (field number) along timeline
//         bit2: movie at current position
//         bit3: motion-play mode
//         bit4: still-play mode
//         bit5: current play rate
//         bit6 ..  bit15: reserved
//
//     data3-n: status values requested
//         (value corresponding to the lowest bit in the mask first):
//         Play/record state (1 byte): one of the following values:
//             0: idle
//             1: play-cued
//             2: record-cued
//             3: jog-play
//             4: jog-record
//             5: shuttle-play
//             6: shuttle-record
//         Current position along timeline (integer, 4 bytes)
//         Current movie on timeline (4 bytes)
//         Motion-play mode (1 byte): one of the following values:
//             0: normal play
//             1: loop play
```

```
//             2: bounce play
//             3: limited play
//             4: slaved play
//         Still-play mode (1 byte): one of the following values:
//             0: still play by field
//             1: still play by frame
//         Current play rate (float, 4 bytes)
//
BOOL PpGetPortStatusReply(UINT* mask, UINT values[], UINT numvals)
{
    char buffer[256];
    UINT len, i;
    UINT bits = 0;

    if (!PpGetReply(buffer, &len, Cmd1_PortAccess, Cmd2_GetPortStatus)) {
        return FALSE;
    }

    if ((UINT)((USHORT *)(buffer[PRO_DATA1_INDEX])) != *mask) {
        printf("Port Status changed mask\n");
    }

    *mask = (UINT)((UINT *)(buffer[PRO_DATA1_INDEX]));

    for (i = 0; i < 16; i++) {
        if (*mask & (1<<i)) {
            bits++;
        }
    }
    if (bits > numvals) {
        printf("Port Status not enough room for reply\n");
        return FALSE;
    }

    for (i = 0, len = Bytes_RequestMask; i < numvals; i++) {
        if (*mask & (1<<i)) {
            switch (1<<i) {
            case Dat1_GetPlayRecordState:
                values[i] = (UINT)buffer[PRO_DATA1_INDEX+len];
                len += Bytes_PlayRecordState;
                break;
            case Dat1_GetPosition:
                values[i] = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+len])));
                len += Bytes_Position;
                break;
            case Dat1_GetCurrentMovie:
                values[i] = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+len])));
                len += Bytes_MovieHandle;
                break;
            case Dat1_GetMotionPlayMode:
                values[i] = (UINT)buffer[PRO_DATA1_INDEX+len];
                len += Bytes_MotionPlayMode;
                break;
            case Dat1_GetStillPlayMode:
                values[i] = (UINT)buffer[PRO_DATA1_INDEX+len];
                len += Bytes_StillPlayMode;
                break;
            case Dat1_GetCurrentRate:
                values[i] = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+len])));
                len += Bytes_CurrentRate;
                break;
            }
        }
```

```
    }
    return TRUE;

} // PpGetPortStatusReply


//
// Open Port
//
// Reply:
//     Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5: handle for open port.  If port cannot be opened, this value will be 0.
//
BOOL PpOpenPortReply(VdrHandle* portHandle)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PortAccess, Cmd2_OpenPort))
        return FALSE;

    *portHandle = (VdrHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    return TRUE;

} // PpOpenPortReply

//
// Generic Error Number reply routine
//
// Reply:
//     Ack + transaction number
//     xact:
//         data1: transaction number
//         data2: one of the following:
//             0: OK
//             1-255: (error conditions)
//
BOOL PpErrorNumReply(UINT cmd1, UINT cmd2)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, cmd1, cmd2)) {
        return FALSE;
    }
    if (buffer[PRO_DATA1_INDEX+1] == '\0') {
        return TRUE;
    }

    printf("PpErrorNumReply received %02.2X", buffer[PRO_DATA1_INDEX+1]);
    return FALSE;

} // PpErrorNumReply
//
// Specific Error Number reply routine
//
// Reply:
//     data1: one of the following:
//         0: OK
//         1-255: (error conditions)
//
```

```
BOOL PpPdrErrorNumReply(UINT cmd1, UINT cmd2)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, cmd1, cmd2)) {
        return FALSE;
    }
    if (buffer[PRO_DATA1_INDEX] == '\0') {
        return TRUE;
    }

    printf("PpErrorNumReply received %02.2X", buffer[PRO_DATA1_INDEX]);
    return FALSE;

} // PpPdrErrorNumReply


//
// PpAllocateResource
// PpGetResourceConnectHandle
//
// Reply:
//     data1:  transaction number
//     data2-5: resource handle
//
BOOL PpPortHandleReply(VdrHandle* portHandle, UINT cmd2)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_ResourceAccess, cmd2)) {
        return FALSE;
    }
    *portHandle = (VdrHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    return TRUE;

} // PpPortHandleReply


//
// Attach Movie
// Attach Movie with Marks
//
// Reply:
//     Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5:  handle of attached movie (0 if failure)
//
BOOL PpAttachMovieReply(MovieHandle* movieHandle, UINT cmd2)
{
    char buffer[256];
    UINT len;

    // Synchronous operation.
    if (!PpGetReply(buffer, &len, Cmd1_AttachedMovieAccess, cmd2)) {
        return FALSE;
    }
    *movieHandle = (MovieHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    return TRUE;

} // PpAttachMovieReply
```

```
//
// Get Resource Status Reply
//
//     data1-2: a copy of the request mask from command data5-6 (any request which is
//         not implemented will cause the corresponding bit in the mask to be cleared)
//     data3-n: status values requested (value corresponding to the lowest bit in the
//         mask first:
//         (video codecs only:)
//         Current field size (integer, 4 bytes)
//         Current luminance Q factor (float, 4 bytes)
//         (audio codecs only:)
//         Current audio level (integer, 4 bytes)
//         (timecode recorder/generators only:)
//         Current time code (time code, 4 bytes)
//         Current user bits (integer, 4 bytes)
//
BOOL PpGetResourceStatusReply(UINT* mask, UINT values[], UINT numvals)
{
    char buffer[256];
    UINT len, i;
    UINT bits = 0;

    if (!PpGetReply(buffer, &len, Cmd1_ResourceAccess, Cmd2_GetResourceStatus)) {
        return FALSE;
    }

    if ((UINT)((USHORT *)(buffer[PRO_DATA1_INDEX])) != *mask) {
        printf("Get Resource Status changed mask\n");
    }

    *mask = (UINT)((UINT *)(buffer[PRO_DATA1_INDEX]));

    for (i = 0; i < 16; i++) {
        if (*mask & (1<<i)) {
            bits++;
        }
    }
    if (bits > numvals) {
        printf("Get Resource Status not enough room for reply\n");
        return FALSE;
    }

    for (i = 0, len = Bytes_RequestMask; i < numvals; i++) {
        if (*mask & (1<<i)) {
            switch (1<<i) {
            case Dat1_GetCurrentFieldSize:
                values[i] = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+len])));
                // All codec have 4 bytes for first value.
                len += Bytes_FieldSize;
                break;
            case Dat1_GetCurrentLumQFactor:
                values[i] = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+len])));
                len += Bytes_Position;
                break;
            }
        }
    }
    return TRUE;

} // PpGetResourceStatusReply
```

```
//
// PpAllocateResource
// PpGetResourceConnectHandle
//
// Reply:
//     data1:   transaction number
//     data2-5: resource handle
//
BOOL PpResourceHandleReply(ResourceHandle* rscHandle, UINT cmd2)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_ResourceAccess, cmd2)) {
        return FALSE;
    }

    *rscHandle = (ResourceHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));

    if (*rscHandle == NULL) {
        return FALSE;
    }

    return TRUE;

} // PpResourceHandleReply


//
// Get Standard Reply
//
//     data1: the system standard
//
BOOL PpGetStandardReply(ConnectHandle* connection)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_SystemAccess, Cmd2_GetStandard)) {
        return FALSE;
    }

    *connection = (ConnectHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (*connection == NULL) {
        return FALSE;
    }
    return TRUE;

} // PpGetStandardReply
//
// Get Open Movie Reply
//
//     data1:   transaction number
//     data2-5: movie token for new movie (Null if error)
//
BOOL PpGetOpenMovieReply(UINT* movietoken)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrOpenMovie)) {
        return FALSE;
```

```
    }
    *movietoken = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (!movietoken) {
        return FALSE;
    }
    return TRUE;

} // PpGetOpenMovieReply


//
// Get Attach Open Movie Reply
//
//     data1:   transaction number
//     data2-5: movie handle for new movie (Null if error)
//
BOOL PpGetAttachOpenMovieReply(MovieHandle* moviehandle)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_AttachedMovieAccess, Cmd2_AttachOpenMovie)) {
        return FALSE;
    }
    *moviehandle = (MovieHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));

    if (*moviehandle == NULL) {
        return FALSE;
    }
    return TRUE;

} // PpGetAttachMovieReply


//
// Get Schedule Event Reply
//
//     data1:   transaction number
//     data2-5: event handle for new movie (Null if error)
//
BOOL PpGetScheduleEventReply(EventHandle* eventhandle)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_EventAccess, Cmd2_ScheduleEvent)) {
        return FALSE;
    }
    *eventhandle = (EventHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));

    if (*eventhandle == NULL) {
        return FALSE;
    }
    return TRUE;

} // PpGetScheduleEventReply


//
// Get First Dataset Reply
//
//     data1:   transaction number
//     data2-5: token to use fro subsquent calls
```

```
//     data6-n: name of the first dataset
//
BOOL PpGetFirstDatasetReply(char* dataset)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstDataset)) {
        return FALSE;
    }
    dset = (EnumToken)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (!dset) {
        return FALSE;
    }
    strcpy(dataset, &buffer[PRO_DATA1_INDEX+5]);
    return TRUE;

} // PpGetFirstDatasetReply


//
// Get First Group Reply
//
//     data1:   transaction number
//     data2-5: token to use for subsquent calls
//     data6-n: name of the first group
//
BOOL PpGetFirstGroupReply(char* group)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstGroup)) {
        return FALSE;
    }
    grp = (EnumToken)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (!grp) {
        return FALSE;
    }
    strcpy(group, &buffer[PRO_DATA1_INDEX+5]);
    return TRUE;

} // PpGetFirstGroupReply


//
// Get First Movie Reply
//
//     data1:   transaction number
//     data2-5: token to use for subsquent calls
//     data6-n: name of the first movie
//
BOOL PpGetFirstMovieReply(char* name)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindFirstMovie)) {
        printf("Error, get first movie\n");
        return FALSE;
    }
    movie = (EnumToken)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (!movie) {
```

```
        return FALSE;
    }
    strcpy(name, &buffer[PRO_DATA1_INDEX+5]);
    return TRUE;

} // PpGetFirstMovieReply
//
// Get Next Dataset Reply
//
// data1:   transaction number
// data2-n: name of the next dataset
//
BOOL PpGetNextDatasetReply(char* dataset)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextDataset)) {
        printf("Error, get next dataset\n");
        return FALSE;
    }
    strcpy(dataset, &buffer[PRO_DATA1_INDEX+1]);
    return TRUE;

} // PpGetNextDatasetReply


//
// Get Next Group Reply
//
//     data1:   transaction number
//     data2-n: name of the next group
//
BOOL PpGetNextGroupReply(char* group)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextGroup)) {
        printf("Error, get next grouop\n");
        return FALSE;
    }
    strcpy(group, &buffer[PRO_DATA1_INDEX+1]);
    return TRUE;

} // PpGetNextGroupReply


//
// Get Next Movie Reply
//
//     data1:   transaction number
//     data2-n: name of the next movie
//
BOOL PpGetNextMovieReply(char* name)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrFindNextMovie)) {
        printf("Error, get next movie\n");
        return FALSE;
    }
```

```
    strcpy(name, &buffer[PRO_DATA1_INDEX+1]);
    return TRUE;

} // PpGetNextMovieReply


//
// Get Movie State Reply
//
//     data1:     status of call (0-ok, else error)
//     data2-5:   Movie attributes
//     data6-9:   Minimum length
//     data10-13: Maximum length
//     data14-17: Movie creation time (date, time 2 bytes each)
//     data18-21: Movie last modification time (date, time 2 bytes each)
//     data22:    number of video tracks
//     data23:    number of audio tracks
//     data24:    number of timecode tracks
//
BOOL PpGetMovieStateReply(PdrMovieState* state)
{
    char    buffer[256];
    UINT    len;
    FILETIME time;
    WORD    dosDate;
    WORD    dosTime;

    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrGetMovieState)) {
        printf("Get movie status error\n");
        return FALSE;
    }
    // Fill in the state structure.
    state->attributes = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    state->minLength = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+5])));
    state->maxLength = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+9])));

    // Get the create time.
    // Read it in MS-DOS date and time and turn it into 64 bit Filetime.
    dosDate = (USHORT)(*((USHORT *)(&buffer[PRO_DATA1_INDEX+13])));
    dosTime = (USHORT)(*((USHORT *)(&buffer[PRO_DATA1_INDEX+15])));
    if (!DosDateTimeToFileTime(dosDate, dosTime, &time)) {
        printf("DOS TIME\n");
        return FALSE;
    }
    state->createTime = time;

    // Get the last change time.
    dosDate = (USHORT)(*((USHORT *)(&buffer[PRO_DATA1_INDEX+17])));
    dosTime = (USHORT)(*((USHORT *)(&buffer[PRO_DATA1_INDEX+19])));
    if (!DosDateTimeToFileTime(dosDate, dosTime, &time)) {
        printf("DOS TIME\n");
        return FALSE;
    }
    state->lastChangedTime = time;

    state->numV = buffer[PRO_DATA1_INDEX+21];
    state->numA = buffer[PRO_DATA1_INDEX+22];
    state->numT = buffer[PRO_DATA1_INDEX+23];
    state->exclusivePID = (UINT)(*((UINT *)(&buffer[PRO_DATA1_INDEX+24])));

    return TRUE;
```

```
} // PpGetMovieStateReply


//
// Get Next Track Reply
//
//     data1-2: Next Track Token
//
BOOL PpGetNextTrackReply(TrackToken* tracktoken)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_StoredMediaAccess, Cmd2_GetNextTrack)) {
        return FALSE;
    }
    *tracktoken = (USHORT)(*((USHORT *)(&buffer[PRO_DATA1_INDEX])));
    if (*tracktoken == 0) {
        printf("Can not get tracktoken\n");
        return FALSE;
    }
    return TRUE;

} // PpGetNextTrackReply


//
// Get Next Media Token Reply
//
//     data1-8: media token
//
BOOL PpGetNextMediaReply(MediaToken* mediatoken)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_StoredMediaAccess, Cmd2_GetNextMediaToken)) {
        return FALSE;
    }
    *mediatoken = (*((MediaToken *)(&buffer[PRO_DATA1_INDEX])));
    if (!mediatoken) {
        return FALSE;
    }
    return TRUE;

} // PpGetNextMediaReply


//
// Get Null Media Token Reply
//
//     data1-8: track token
//
BOOL PpGetNullMediaTokenReply(MediaToken* mediatoken)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_StoredMediaAccess, Cmd2_GetNullMediaToken)) {
        return FALSE;
    }

    *mediatoken = (*((MediaToken *)(&buffer[PRO_DATA1_INDEX])));
```

```
    if (PpIsMediaTokenNull(mediatoken)) {
        return FALSE;
    }
    return TRUE;

} // PpGetNullMediaTokenReply


//
// Get Media Path Reply
//
//     data1-2: mask
//     data1-n: media path
//
BOOL PpGetMediaPathReply(UINT* mask, char* mediapath)
{
    char buffer[256];
    UINT len;

    if (!PpGetReply(buffer, &len, Cmd1_StoredMediaAccess, Cmd2_GetStoredMediaStatus)) {
        return FALSE;
    }

    strcpy(mediapath, &buffer[PRO_DATA1_INDEX+2]);

    return TRUE;

} // PpGetMediaPathReply


//
// Attach Media Reply
// Attach Media with Marks
//
// Reply:
//     Ack + transaction number
//     xact:
//         data1: transaction number
//         data2-5:  handle of attached media (0 if failure)
//
BOOL PpAttachMediaReply(MediaHandle* mediahandle, UINT cmd2)
{
    char buffer[256];
    UINT len;

    // ssynchronous operation.
    if (!PpGetReply(buffer, &len, Cmd1_AttachedMediaAccess, cmd2)) {
        return FALSE;
    }

    *mediahandle = (MediaHandle)(*((UINT *)(&buffer[PRO_DATA1_INDEX+1])));
    if (*mediahandle == 0) {
        printf("Can not get mediahandle\n");
        return FALSE;
    }
    return TRUE;

} // PpAttachMediaReply

//
// Get Movie Exist Reply
//
BOOL PpGetMovieExistReply(UINT* exist)
```

```
{
    char buffer[256];
    UINT len;

    // Synchronous operation.
    if (!PpGetReply(buffer, &len, Cmd1_PdrMovieAccess, Cmd2_PdrMovieExist)) {
        return FALSE;
    }
    *exist = (UINT)(*((UCHAR *)(&buffer[PRO_DATA1_INDEX])));
    if (!exist) {
        return FALSE;
    }

    return TRUE;

} // PpGetMovieExistReply
//
// Is Media Token Null Reply
//
BOOL PpIsMediaTokenNullReply(void)
{
    char buffer[256];
    UINT len;
    UCHAR isnull;

    if (!PpGetReply(buffer, &len, Cmd1_StoredMediaAccess, Cmd2_IsMediaTokenNull)) {
        return FALSE;
    }
    isnull = buffer[PRO_DATA1_INDEX];
    if (isnull == 1) {
        return FALSE;
    }
    return TRUE;

} // PpIsMediaTokenNullReply
```

Packet communication

Example 21, *ppcomm.c* contains various generic utility functions to enable the communication of packets in Profile protocol over a serial RS-422 connection. These are common functions used by all of the Profile serial programming sample applications.

The function declarations described here are shown in the file *ppheader.h*:

- **PpOpenComm** opens a serial COM port using Windows SDK calls (Win16/Win32). It also sets up appropriate parameters for flow control of the RS-422 connection.
- **PpAddChksum** computes the required checksum for a given buffer.
- **PpValidateChksum** ensures that a return packet contains a valid checksum.
- **PpSendCommand** sends a command buffer out the RS-422 serial port, using the Win16/ Win32 function WriteFile with the appropriate communication handle.
- **PpRcvReply** receives a reply packet from the remote Profile.

Example 21. ppcomm.c

```
//
// File: ppcomm.c
// This is a part of the Grass Valley Group Profile Source Code Samples.
//
// Copyright (c) Grass Valley Group Inc.  This program, or portions thereof,
// is protected as an unpublished work under the copyright laws of
// the United States.
//
// This source code is only intended as a supplement to
// Profile Development Tools and documentation of Native Protocol.
//

#include <stdio.h>
#include <windows.h>
#include <tekvdr.h>
#include "ppcomm.h"
#include <profhdr.h>
#include <profcmd.h>

static HANDLE sCommHdl;

extern char* comm;

//
// Add a checksum to the buffer.
//
void PpAddChksum(UCHAR* buffer, UINT bcnt)
{
    UINT i;
    UCHAR cksum = 0;

    // Skip the header.
    bcnt += PRO_CMD1_INDEX;

    for (i = PRO_CMD1_INDEX; i < bcnt; i++) {
        cksum += buffer[i];
    }
    cksum = (~cksum) + 1;
    buffer[bcnt] = cksum;
```

```
} // PpAddChksum


//
// Send out the command.
//
BOOL PpSendCommand(UCHAR* buffer, UINT bcnt, UINT cmd1, UINT cmd2)
{
    DWORD wrcnt;

    // Bump for cmds.
    bcnt += NUM_PRO_HEADER_BYTES;
    if (bcnt > 0xff) {
        // If illegal count - return error.
        return FALSE;
    }

    buffer[PRO_STX_INDEX] = PRO_STX_HEADER; // Add the STX, and...
    buffer[PRO_BYTE_COUNT_INDEX] = bcnt;    // the byte count from cmd1 to end.
    buffer[PRO_CMD1_INDEX] = cmd1;          // Put down the commands,
    buffer[PRO_CMD2_INDEX] = cmd2;
    PpAddChksum(buffer, bcnt);              // and tack on the checksum.

    // Do the transaction (len bcnt+NUM_PRO_HEADER_BYTES+NUM_PRO_CHECKSUM_BYTES).
    bcnt += NUM_PRO_HEADER_BYTES + NUM_PRO_CHECKSUM_BYTES;
    if (!WriteFile(sCommHdl, buffer, (DWORD)bcnt, &wrcnt, NULL)) {
        printf("Error writing to Comm Device \n");
        return FALSE;
    }
    return TRUE;
} // PpSendCommand

//
// Validate the checksum in the buffer
//
BOOL PpValidateChksum(UCHAR* buffer)
{
    UINT i, bcnt;
    UCHAR cksum = 0;

    bcnt = buffer[PRO_BYTE_COUNT_INDEX];    // Get the byte count.
    bcnt += PRO_CMD1_INDEX;                 // Skip the header.
    for (i = PRO_CMD1_INDEX; i < bcnt; i++) {
        cksum += buffer[i];
    }
    cksum = (~cksum) + 1;
    if (buffer[bcnt] != cksum) {
        return FALSE;
    }
    return TRUE;

} // PpValidateChksum
//
// Reply - this buffer must be 256 bytes in length
// Input:
//   buffer is unsigned char pointer of at least 256 bytes.
//   bcnt is unsigned int pointer to where the return length should be supplied.
//   cmd1 and cmd2 are the commands expected if data is available
//
BOOL PpRcvReply(UCHAR* buffer, UINT* bcnt, UINT cmd1, UINT cmd2)
{
    BOOL retstat;
```

```
    DWORD rdcnt;

    // First get the reply byte and byte count or status.
    if (!ReadFile(sCommHdl, buffer, (DWORD)2, &rdcnt, NULL)) {
        return FALSE;
    }
    if (rdcnt != 2) {
        return FALSE;
    }

    switch (buffer[0])  {
    case Stx_Nak:
        // if NAK just show cause.
        *bcnt = 2;
        switch (buffer[PRO_BYTE_COUNT_INDEX]) {
        case Dat1_UndefinedError:
            printf("Undefined Error\n");
            break;
        case Dat1_CheckSumError:
            printf("NAK - Checksum Error\n");
            break;
        case Dat1_ParityError:
            printf("NAK - Parity Error\n");
            break;
        case Dat1_OverRun:
            printf("NAK - Overrun Error\n");
            break;
        case Dat1_FramingError:
            printf("UNAK - Framing Error\n");
            break;
        case Dat1_TimeOut:
            printf("UNAK - Timeout Error\n");
            break;
        default:
            printf("Unspecified Error\n");
            break;
        }
        retstat = FALSE;
        break;

    case Stx_Stx:
        // if ACK use bytecount+1 for next read to include reply + checksum
        rdcnt = *bcnt = (UINT)buffer[1] + 1;

        if (!ReadFile(sCommHdl, &buffer[PRO_CMD1_INDEX], (DWORD)rdcnt, &rdcnt, NULL) || *bcnt
!= rdcnt) {
            retstat = FALSE;
            break;
        }

        // Now validate the checksum and the assumed command return.
        if (!PpValidateChksum(buffer)) {
            retstat = FALSE;
        }
        else {
            if (buffer[PRO_CMD1_INDEX] == cmd1 && buffer[PRO_CMD2_INDEX] == cmd2) {
                retstat = TRUE;
            }
            else {
                retstat = FALSE;
            }
        }
        *bcnt += 2;
```

```
            break;

        case Stx_Ack:
            // If transaction then read the transaction number.
            if (!ReadFile(sCommHdl, &buffer[PRO_CMD1_INDEX], (DWORD)1, &rdcnt, NULL)
                    || rdcnt != 1) {
                return FALSE;
            }
            *bcnt = 3;

            // First check to see if all is fine.
            if (buffer[PRO_BYTE_COUNT_INDEX] == Dat1_Ok) {
                retstat = TRUE;
            }
            else {
                switch (buffer[PRO_BYTE_COUNT_INDEX]) {
                case Dat1_NotImplemented:
                    printf("Command Not Implemented\n");
                    break;
                case Dat1_PortBusy:
                    printf("Port Busy\n");
                    break;
                case Dat1_IncorrectNumBytes:
                    printf("Incorrect Number of Bytes\n");
                    break;
                }
                retstat = FALSE;
            }
            break;

        default:
            // Unknown response received.
            sprintf(buffer, "UNKNOWN reply %02.2X"<, buffer[0]);
            retstat = FALSE;
            *bcnt = 1;
            break;
        }

        return retstat;

    } // PpRcvReply

    //
    // Open the communication port
    //
    BOOL PpOpenComm(void)
    {
        COMMTIMEOUTS CommTimeOuts;
        DCB dcb;

        if (sCommHdl) {
            CloseHandle(sCommHdl);
        }

        sCommHdl = CreateFile(comm, GENERIC_READ | GENERIC_WRITE, 0, NULL,
             OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (!sCommHdl) {
            return FALSE;
        }

        CommTimeOuts.ReadIntervalTimeout = 25;
        CommTimeOuts.ReadTotalTimeoutMultiplier = 25;
        CommTimeOuts.ReadTotalTimeoutConstant = 300;
```

```
    CommTimeOuts.WriteTotalTimeoutMultiplier = 0;
    CommTimeOuts.WriteTotalTimeoutConstant = 0;

    if (!SetCommTimeouts(sCommHdl, &CommTimeOuts)) {
        return FALSE;
    }

    dcb.DCBlength = sizeof(DCB);

    if (!GetCommState(sCommHdl, &dcb)) {
        return FALSE;
    }

    dcb.BaudRate      = CBR_38400;
    dcb.ByteSize      = DATABITS_8;
    dcb.Parity        = ODDPARITY;
    dcb.StopBits      = ONESTOPBIT;
    dcb.fBinary       = TRUE;
    dcb.fOutxCtsFlow  = FALSE;
    dcb.fOutxDsrFlow  = FALSE;
    dcb.fDtrControl   = FALSE;
    dcb.fOutX         = FALSE;
    dcb.fInX          = FALSE;
    dcb.fRtsControl   = FALSE;
    dcb.fAbortOnError = TRUE;

    // Setup hardware flow control.
    dcb.fOutxDsrFlow = FALSE;
    dcb.fDtrControl = DTR_CONTROL_DISABLE;

    dcb.fOutxCtsFlow = FALSE;
    dcb.fRtsControl = RTS_CONTROL_DISABLE;

    // Setup software flow control.
    dcb.fInX = dcb.fOutX = FALSE;

    // Setup other various settings.
    dcb.fParity = TRUE;

    if (!SetCommState(sCommHdl, &dcb)) {
        return FALSE;
    }

    if (!SetCommMask(sCommHdl, EV_RXCHAR)) {
        return FALSE;
    }

    return TRUE;

} // PpOpenComm
```

Index