StreamStor Real-Time Storage Controller

SDK 9

User's Guide

Copyright and Trademarks

Printed in the United States.

Manual version: 9.75

Publication date: August 21, 2012

Table of Contents

License Agreement and Limited Warranty

IMPORTANT: CAREFULLY READ THE TERMS AND CONDITIONS OF THIS AGREEMENT BEFORE USING THE PRODUCT. By installing or otherwise using the StreamStor Product, you agree to be bound by the terms of this Agreement. If you do not agree to the terms of this Agreement, do not install or use the StreamStor Product and return it to Conduant Corporation.

GRANT OF LICENSE. In consideration for your purchase of the StreamStor Product, Conduant Corporation hereby grants you a limited, non-exclusive, revocable license to use the software and firmware which controls the StreamStor Product (hereinafter the "Software") solely as part of and in connection with your use of the StreamStor Product. If you are authorized to resell the StreamStor Product, Conduant Corporation hereby grants you a limited non-exclusive license to transfer the Software only in conjunction with a sale or transfer by you of the StreamStor Product controlled by the Software, provided you retain no copies of the Software and the recipient agrees to be bound by the terms of this Agreement and you comply with the RESALE provision herein.

NO REVERSE ENGINEERING. You may not cause or permit, and must take all appropriate and reasonable steps necessary to prevent, the reverse engineering, decompilation, reverse assembly, modification, reconfiguration or creation of derivative works of the Software, in whole or in part.

OWNERSHIP. The Software is a proprietary product of Conduant Corporation which retains all title, rights and interest in and to the Software, including, but not limited to, all copyrights, trademarks, trade secrets, know-how and other proprietary information included or embodied in the Software. The Software is protected by national copyright laws and international copyright treaties.

TERM. This Agreement is effective from the date of receipt of the StreamStor Product and the Software. This Agreement will terminate automatically at any time, without prior notice to you, if you fail to comply with any of the provisions hereunder. Upon termination of this Agreement for any reason, you must return the StreamStor Product and Software in your possession or control to Conduant Corporation.

LIMITED WARRANTY. This Limited Warranty is void if failure of the StreamStor Product or the Software is due to accident, abuse or misuse.

Hardware: Conduant's terms of warranty on all manufactured products is one year from the date of shipment from our offices. After the warranty period, product support and repairs are available on a fee paid basis. Warranty on all third party materials sold through Conduant, such as chassis, disk drives, PCs, bus extenders, and drive carriers, is passed through with the original manufacturer's warranty. Conduant will provide no charge service for 90 days to replace or handle repair returns on third party materials. Any charges imposed by the original manufacturer will be passed through to the customer. After 90 days, Conduant will handle returns on third party material on a time and materials basis.

Software: The warranty on all software products is 90 days from the date of shipment from Conduant's offices. After 90 days, Conduant will provide product support and upgrades on a fee paid basis. Warranties on all third party software are passed through with the original manufacturer's warranty. Conduant will provide no charge service for 90 days to replace or handle repair returns on third party software. Any charges imposed by the manufacturer will be passed through to the customer.

DISCLAIMER OF WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CONDUANT CORPORATION DISCLAIMS ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT, WITH REGARD TO THE STREAMSTOR PRODUCT AND THE SOFTWARE.

SOLE REMEDIES. If the StreamStor Product or the Software do not meet Conduant Corporation's Limited Warranty and you return the StreamStor Product and the Software to Conduant Corporation, Conduant Corporation's entire liability and your exclusive remedy shall be at Conduant Corporation 's option, either (a) return of the price paid, if any, or (b) repair or replacement of the StreamStor Product or the Software. Any replacement Product or Software will be warranted for the remainder of the original warranty period.

LIMITATION OF LIABILITIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL CONDUANT CORPORATION BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE STREAMSTOR PRODUCT AND THE SOFTWARE. IN ANY CASE, CONDUANT CORPORATION'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS AGREEMENT SHALL BE LIMITED TO THE AMOUNT ACTUALLY PAID BY YOU FOR THE STREAMSTOR PRODUCT AND THE SOFTWARE. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

RESALE. If you are authorized to resell the StreamStor Product, you must distribute the StreamStor Product only in conjunction with and as part of your product that is designed, developed and tested to operate with and add significant functionality to the StreamStor Product; you may not permit further distribution or transfer of the StreamStor Product by your end-user customer; you must agree to indemnify, hold harmless and defend Conduant Corporation from and against any claims or lawsuits, including attorneys' fees, that arise or result from the use or distribution of your product; and you may not use Conduant Corporation's name, logos or trademarks to market your product without the prior written consent of Conduant Corporation.

ENTIRE AGREEMENT; SEVERABILITY. This Agreement constitutes the complete and exclusive agreement between you and Conduant Corporation with respect to the subject matter hereof and supersedes all prior written or oral agreements, understandings or communications. If any provision of this Agreement is deemed invalid under any applicable law, it shall be deemed modified or omitted to the extent necessary to comply with such law and the remainder of this Agreement shall remain in full force and effect.

GOVERNING LAW. This Agreement is governed by the laws of the State of Colorado, without giving effect to the choice of law provisions therein. By accepting this Agreement, you hereby consent to the exclusive jurisdiction of the state and federal courts sitting in the State of Colorado.

Chapter 1
Introduction

The StreamStor Software Development Kit

One of the most powerful features of StreamStor is that it is an open platform device allowing other PCI devices complete access to record or read data from the disk storage. Conduant makes it easy for system designers to use StreamStor by providing the StreamStor Software Development Kit (SDK). This manual is applicable to SDKs with a major number of 9 (i.e., SDK 9.0, SDK 9.1, etc.).

The SDK includes an Application Programming Interface (API) library. This library provides the control software for StreamStor in the form of DLLs (Dynamic Link Libraries) for Windows and an archive library for Linux that can be accessed by user application software. Application software can be developed in any environment capable of utilizing these library functions. This includes the various Windows programming languages such as Visual C++ and Visual Basic as well as graphical programming environments such as LabVIEW.

Installing the Software

Your StreamStor system was shipped with the Software Development Kit on CD-ROM. Please power up your computer. On Windows systems, when ready, run the `setup.exe` program on the CD-ROM to start the installation process. On Linux systems, refer to the file `linux/docs/install.txt` on the CD-ROM.

Plug and play operating systems such as Windows will detect the installation of the StreamStor board and attempt to configure the boards using the hardware plug and play wizard program. The required installation information file for plug and play installation is included on the CD-ROM. Make sure the plug and play wizard includes the CD-ROM drive in its search so that the StreamStor drivers will be properly installed. You should not cancel the plug and play wizard since this can create hardware conflicts in the system when using the StreamStor controller. Note that the `setup.exe` program must still be executed to install the StreamStor SDK onto your system.

The software installation procedure will install the device drivers, library files, example programs and all other components of the SDK onto your system.

The StreamStor SDK does not include software interfaces or drivers used for the control of data acquisition cards made by other manufacturers. However, it does include some sample programs to help in your software development efforts. Other drivers and examples may be available depending on your choice of data acquisition hardware. Contact Conduant support for more information.

Always review the `readme.html` file included with the SDK for the latest information not included in this manual.

Software Components

The SDK software components include operating system device drivers, support files, programming libraries and utility programs.

Device Driver

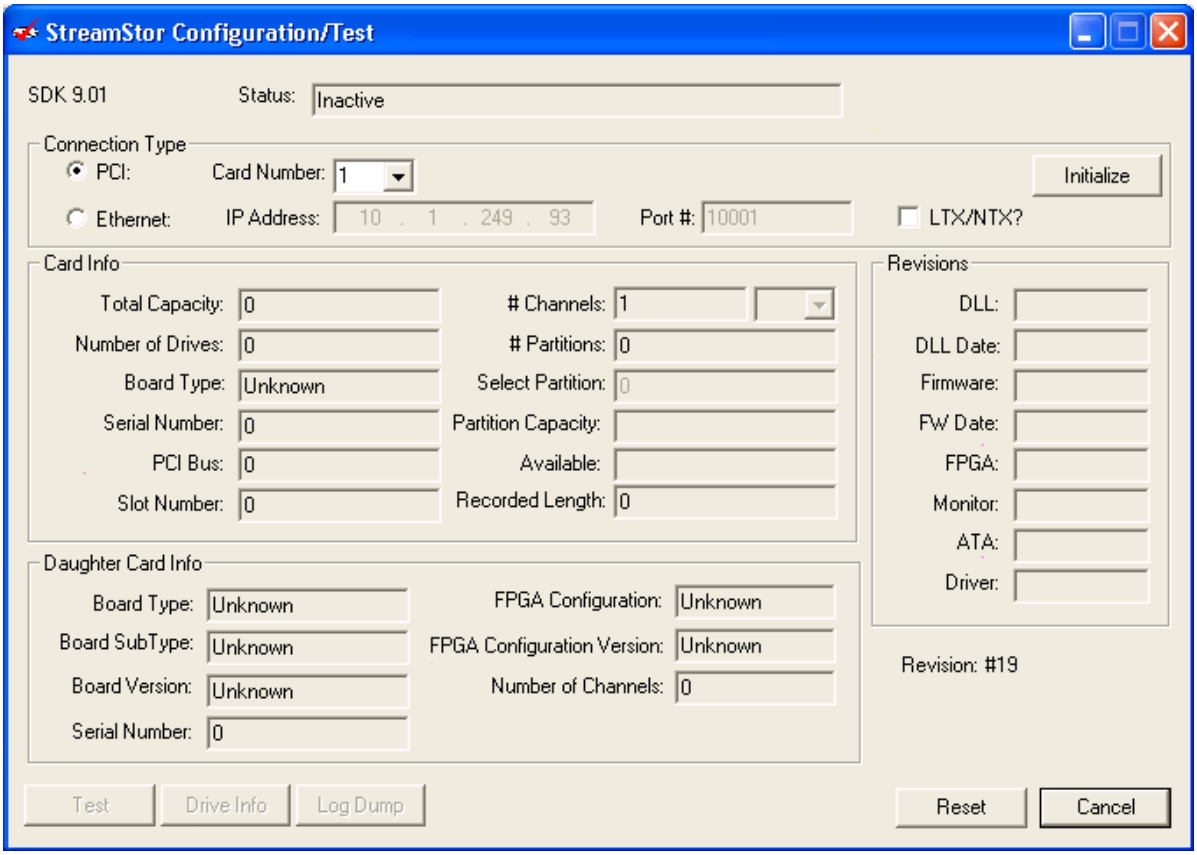
The StreamStor SDK provides device driver support for the Windows 2000, Windows XP, Windows 7 and Linux operating systems. The drivers are installed automatically by the supplied setup program. On Windows systems, the device driver is named `windrvr6.sys`. The Linux device driver is installed as a kernel module named `windrvr6`. On Linux systems, refer to the file `linux/docs/install.txt` on the CD-ROM for driver installation instructions.

Windows Uninstall

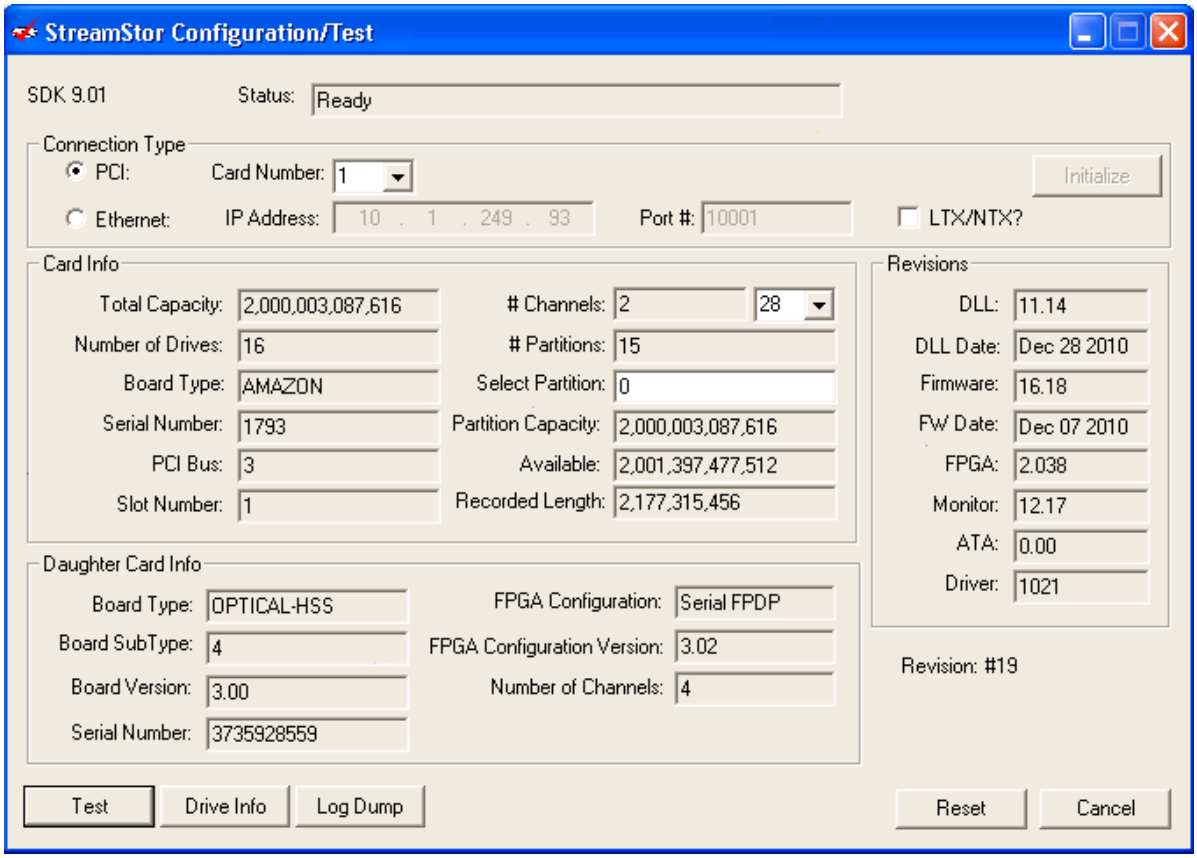
The StreamStor SDK can be easily uninstalled in Windows by using the "Add/Remove Software" wizard in the control panel. Simply select "StreamStor SDK" and all installed components will be automatically removed. You can also select "Remove StreamStor SDK" in the StreamStor menu.

Windows Configuration/Test Utility

The utility program `sscfg.exe` is included with the SDK for testing the StreamStor system for proper configuration and functionality. If you have just received your StreamStor system or you are experiencing problems, this program will open the StreamStor device and report configuration information. It also includes a basic confidence test to ensure that your system is working properly. NOTE: The confidence test will overwrite any data already present on the recorder. The DLL file `bisrun.dll` is a required component. It should have been installed automatically into the installation directory. If `sscfg.exe` is moved, you must also move `bisrun.dll` to the same directory or to a Windows system directory. The initial `sscfg` screen will look something like this:



If more than one StreamStor is installed in your system (PCI or PCI Express) there will be multiple choices in the card number pull down menu. After selecting the card number you must press the *Initialize* button to begin the process of finding, initializing and querying the StreamStor board for device information. If your board has been successfully configured, *Initialize* will enable the *Test* button and fill in the various device information fields. The `sscfg` screen should now appear similar to this:



If you encounter an error during initialization there may be damage to your system from shipping or the system has not been installed correctly. Please contact technical support for assistance.

The program can also be used to communicate with a remote StreamStor device such as an LTX2 or NTX-16 or a system running the StreamStor remote server. To open these devices you must enter the IP address and port number (default port is 10001).

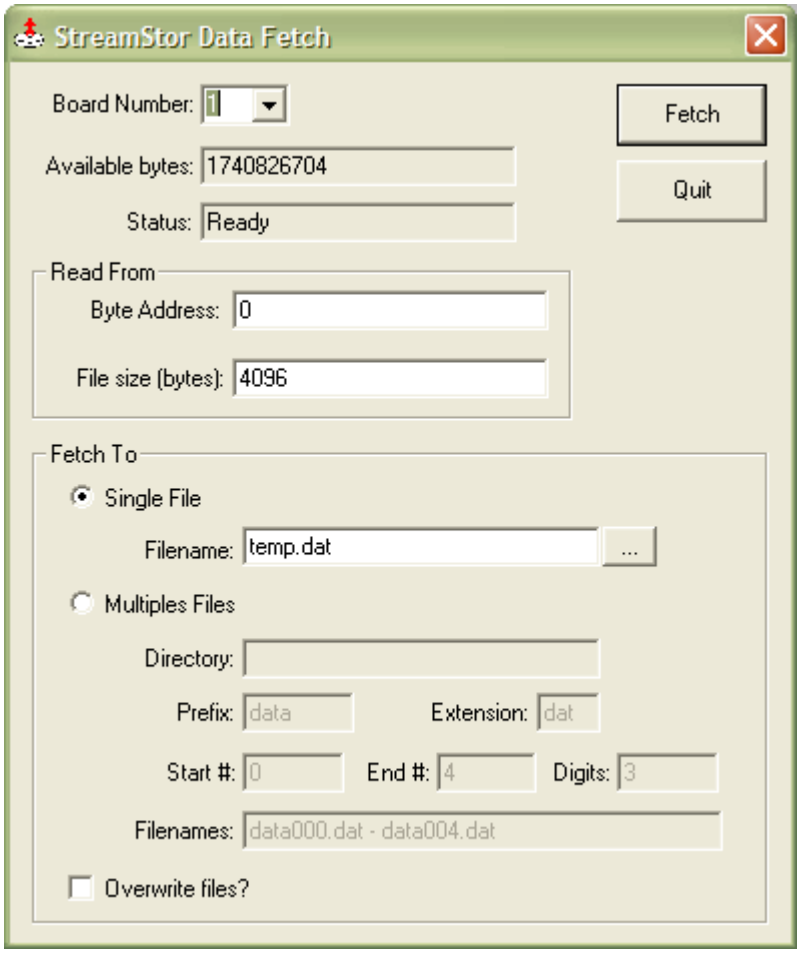
If the initialization has completed successfully you should check the information provided by `sscfg` to ensure your system has been correctly identified according to your purchased model and configuration. If you discover any problems please contact Conduant. At this point you can press the Test button to run a quick confidence test on the controller board and disk system.

CAUTION: ***Running the confidence test in sscfg WILL overwrite any recorded data on StreamStor storage.***

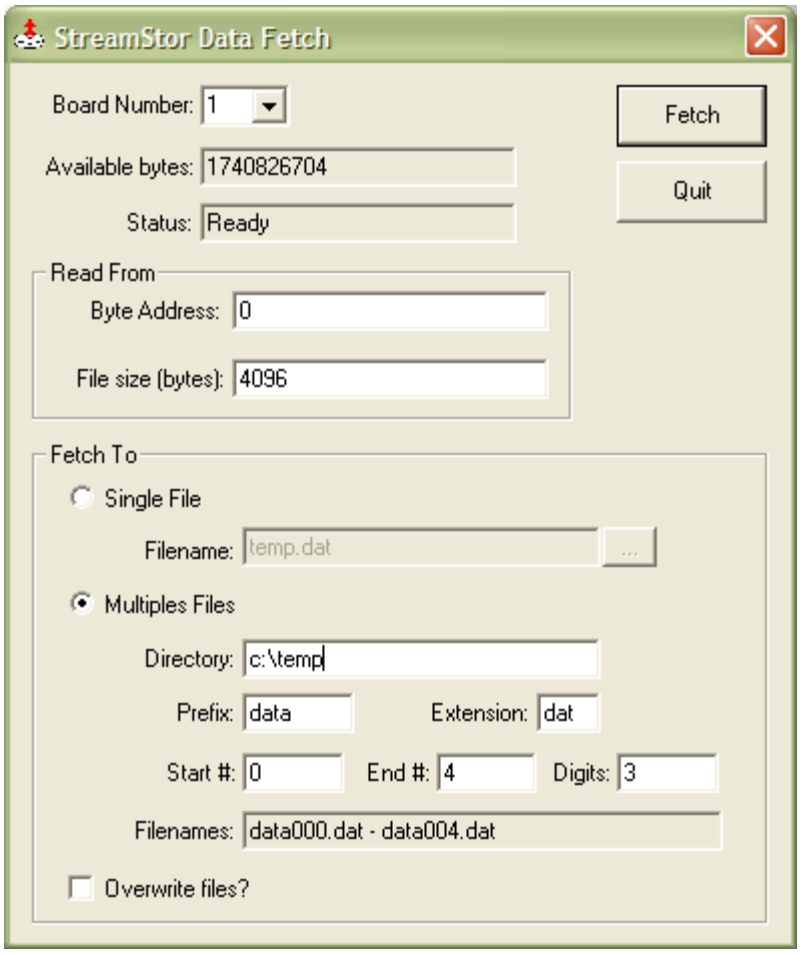
If you get any error messages running this test please follow the instructions in the Troubleshooting section. If this test completes successfully, your StreamStor system is functioning normally.

Windows Fetch Utility

The utility program `ssfetch.exe` has been included to provide a basic tool for retrieving data from the StreamStor storage system-to-system disk files. The interface to `ssfetch` looks like this:



There are two options when using `ssfetch` to retrieve data, the first option is to simply retrieve a block of data to a single system file. The “Single File” button enables this mode and the filename specified is used as the destination for data retrieved from StreamStor. The current status of the recorder is displayed in the “Status” field and the “Available bytes” field indicates the length of data currently recorded on the device. The “Read From” box provides the controls for specfying the location and amount of data to be retrieved. The amount (File size) and address must be an increment of 4 bytes.



The second option for retrieving data is to use the "Multiple Files" option to automatically create system files of sequential and equal size data blocks from StreamStor. The directory field allows you to choose an alternate system directory (current directory will be used by default). The prefix and extension fields are used to define the common text for the filenames. The "Start #", "End #" and "Digits" define a number used to form unique filenames. The "Start #" with the number of digits defined by "Digits" is appended to the prefix and the extension is appended after that (with a period) to form the filename. The "Filenames" area will show a preview of the file names to be used. The amount of data specified by "File size" is written to this file and the process is repeated with the number incrementing until "End #" is reached. The "Byte Address" for each retrieval is incremented by the file size amount so that sequential data is retrieved. This mode is useful for retrieving blocks of data into independent files when the size of the block is fixed such as when digital images have been recorded.

In both modes, the "Byte address" field is automatically incremented after each fetch by the amount of data transferred.

Windows Library

The software development kit includes a DLL library for integration of StreamStor into Windows based user applications. The required DLL file is `xlrapi.dll`. The driver library is also required and is of the form wdapiXXXX.dll where XXXX is the current driver version (1011 in SDK 9.0). The library file `xlrapi.lib` is also included for linking the DLL functions to a user program. The required include files are `xlrapi.h`, `xlrtypes.h` and `xlrdbcommon.h`. Only the `xlrapi.h` file needs to be included in a user program. Example programs are included in the SDK. All of the include files are installed automatically by the installation software in the "Include" directory. The library file for linking user programs is installed in the "Lib" directory and the DLL is installed in the StreamStor installation directory.

Linux Uninstall

The StreamStor SDK can be easily uninstalled in Linux by removing the installation directory and the WinDriver module. To do so, enter the following commands as root where **<InstallDir>** is the full path name where the StreamStor SDK is installed.

1. Verify that the WinDriver driver modules are not being used by another program:

   - View the list of modules and the programs using each of them:

     ```
     # /sbin/lsmod
     ```

   - Identify any applications and modules that are using the WinDriver driver modules. (By default, WinDriver module names begin with **windrvr6**).

   - Close any applications that are using the WinDriver driver module(s).

2. Uninstall WinDriver by running the following command to unload the WinDriver driver module(s):

   ```
   # /sbin/modprobe –r windrvr6
   ```

3. Remove the file **.windriver.rc** from **$HOME** and /etc:

   ```
   # rm –f $HOME/.windriver.rc

   # rm –f /etc/.windriver.rc
   ```

4. Remove the SDK installation directory as follows:

```
# rm –rf <InstallDir>
```

For example, to remove the entire SDK:

```
# rm –rf /usr/local/streamstor
```

5. Remove the WinDriver shared object file, if it exists. It is named:

   /usr/lib/libwdapi<version>.so.

   For example, remove **libwdapi1100.so** for WinDriver Version 11.0.0

Linux Configuration/Test Utilities

Several Linux utility programs are included with the SDK to test the StreamStor system for proper configuration and functionality. If you have just received your StreamStor system or if you are experiencing problems, running these programs will perform configuration and confidence tests to ensure that your system is working properly.

Linux programs that use the StreamStor SDK (such as the utilities below) require that the environment variable `STREAMSTOR_BIB_PATH` be set and exported to the SDK directory containing the StreamStor `*.bib` files. For example:

```
STREAMSTOR_BIB_PATH=/usr/local/streamstor/linux/bib

export STREAMSTOR_BIB_PATH
```

The program `ssopen` simply attempts to open the StreamStor and then closes it. To execute it:

1. `cd <InstallDir>/linux/util`

2. `./ssopen`

   If your system can communicate with the StreamStor board, you should see this output:

```
Attempting to open StreamStor...
StreamStor opened successfully!
Device Status:
   SystemReady-> 1
   MonitorReady-> 0
   DriveFail-> 0
   DriveFailNumber-> 0
   SysError-> 0
   SysErrorCode-> 0
   CtlrError-> 0
```

The program `sstest` is similar to the Windows configuration test, `sscfg.exe`. It will attempt to initialize and configure the StreamStor and perform a confidence test. The confidence test will write data to the StreamStor storage and then will read that data.

☠ **CAUTION:** ***Running the confidence test sstest WILL overwrite any recorded data on StreamStor storage.***

To execute `sstest`:

1. `cd <InstallDir>/linux/util`

2. `./sstest`

If the confidence test completes successfully, you should see output similar to the following:

```
*Getting Device Info
Board Type: AMAZON-EXP
Serial Number: 120030
Number of drives: 8
Total Capacity: 320083329024

*Getting Version Info
SDK Version: 9.02
API Version: 11.31
API Datecode: June 1 2012
Firmware Version: 16.31
Firmware Datecode: September 14 2011
Monitor Version: 12.17
XBAR Version: 2.028
ATA Version: 0.00
Ultra ATA Version: 0.00
Driver Version: 1100

Processing Test Script
Script processing complete.

< optional daughterboard information >

==== Starting Test ====

***Basic Confidence Test***
 ->Writing test pattern
 ->Write Completed

***Check Directory***
   Dir Length: 0x2000000
 ->Read/Compare Recorded Data
 ->Read/Compare Completed
```

```
***Basic Confidence Test Completed***
```

If you get any error messages running this test, please follow the instructions in the Troubleshooting section. If this test completes successfully your StreamStor system is functioning normally.

Linux Library

When the SDK is installed on a Linux system, a static function library is installed named `libssapi.a`. It contains all the StreamStor API functions. The required header files are `xlrapi.h`, `xlrtypes.h` and `xlrdbcommon.h`. Only the `xlrapi.h` file must be included by the user application. The library must be supplied to the linker to create a final executable program. An example C program that shows how to call the SDK library functions and a corresponding `gcc` makefile are in the directory `<InstallDir>/Linux/example`.

Data Structures

StreamStor API functions use the following structures. Refer to the end of the Function Reference section for details on each structure and its members.

S_BANKSTATUS	-	Bank status information
S_DBINFO	-	Daughterboard version information
S_DEVINFO	-	Device info parameters
S_DEVSTATUS	-	Device status flags
S_DIR	-	Recording directory information
S_DRIVEINFO	-	Drive information
S_EVENTS	-	Event information
S_PARTITIONINFO	-	Drive partitioning information
S_READDESC	-	Parameters defining read requests
S_RECDCHANNELINFO	-	Multi-channel recording information
S_SFPDPSTATUS	-	SFPDP port status information
S_SMARTTHRESHOLDS	-	SMART thresholds
S_SMARTVALUES	-	SMART values
S_XLRSWREV	-	Various device version strings

Chapter 2
Function Reference

XLRApiVersion

Syntax:

`void XLRApiVersion( char *versionstring )`

Description:

`XLRApiVersion` returns the API version as a string formatted as a *major.minor* version number.

Parameters:

`versionstring` is a pointer to a character string to hold the returned version. It must be of minimum length `XLR_VERSION_LENGTH.`

Return Value:

The API version is returned in *`versionstring`*.

Usage:

```
/* Read XLR API version into string */
char xlrstring[XLR_VERSION_LENGTH];

XLRApiVersion( xlrstring );
printf( “StreamStor API version is %s”, xlrstring );
```

See Also:

`XLRGetVersion and XLRSdkVersion.`

XLRAppend

Syntax:

`XLR_RETURN_CODE XLRAppend( SSHANDLE *xlrDevice* )`

Description:

`XLRAppend` is used to restart a recording after it has been stopped. Data is appended to the existing recording.

If the StreamStor is in bank mode, data will be appended to the selected bank. If the StreamStor is partitioned, this command will append data to the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE         xlrDevice;
S_READDESC       readDesc;
UINT32           myBuffer[40000];
XLR_RETURN_CODE  xlrReturnCode;

// Open the device
xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRRecord( xlrDevice, 0, 1 );
if( xlrReturnCode != XLR_SUCCESS )
   exit(1);

//
//  Data transfer . . .
//
// Stop the record operation
XLRStop( xlrDevice );

// Read some data back
readDesc.AddrHi = 0;
readDesc.AddrLo = 0x120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = &myBuffer;

xlrReturnCode = XLRRead( xlrDevice, &readDesc );
if( xlrReturnCode != XLR_SUCCESS )
   exit(1);
```

```
//
// Now start recording again without overwriting previous data
//
xlrReturnCode = XLRAppend( xlrDevice );
if( xlrReturnCode != XLR_SUCCESS )
   exit(1);
```

See Also:

XLRRecord, XLRSetBankMode and XLRSelectBank.

XLRArmChannelForSync

Syntax:

`XLR_RETURN_CODE XLRArmChannelForSync( SSHANDLE` *`xlrDevice,`* `UINT32` *`channel,`* `UINT32` *`syncOption`* `)`

Description:

`XLRArmChannelForSync` sets the start-on-sync option on the specified StreamStor FPDPII channel. Setting the option to `SS_OPT_ARM_START_ON_SYNC` arms the channel for a SYNC* pulse. When armed, it will not record or append data until a SYNC* pulse is received. Arming is in effect for a single record or append operation. Therefore, each time you want a record or append operation to wait for a SYNC* pulse, you must call `XLRArmChannelForSync`.

If no SYNC* pulse is received, no data will be recorded.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`channel` is the FPDPII channel number to arm.

syncOption is the option to set. The only option available is `SS_OPT_ARM_START_ON_SYNC`, which arms the FPDPII for start-on-sync.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;

if(XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   exit(1);
}
      …

// Arm FPDPII channel 30 for start-on-sync.
if( XLRArmChannelForSync( xlrDevice, 30,
     SS_OPT_ARM_START_ON_SYNC ) != XLR_SUCCESS )
{
  exit(1);
}

//
// Put the StreamStor in record mode by calling XLRRecord.
// Data will get recorded to disk when the SYNC* pulse is
// received.
//
```

```
if( XLRRecord( xlrDevice, 0, 1 ) != XLR_SUCCESS )
{
  exit(1);
}

   … SYNC* is received, so data will start recording …

//
// Done recording, so call XLRStop. XLRStop will
// clear the arming for start-on-sync.
//
XLRStop();

//
// Now you want to append data to the existing
// recording.  In this example, you want to wait for
// another SYNC* pulse, so you must call XLRArmChannelForSync
// again.
//
if( XLRArmChannelForSync( xlrDevice, 30,
     SS_OPT_ARM_START_ON_SYNC ) != XLR_SUCCESS )
{
  exit(1);
}

//
// Put the StreamStor in append mode by calling XLRAppend.
// Data will get appended to disk when the SYNC* pulse is
// received.
//
if( XLRAppend( xlrDevice) != XLR_SUCCESS )
{
   exit(1);
}

   … SYNC* is received, so data will start recording …

//
// Done appending, so call XLRStop.  XLRStop will clear the
// arming for start-on-sync.
//
XLRStop();
```

See Also:

XLRRecord and XLRAppend.

XLRArmFPDP

Syntax:

`XLR_RETURN_CODE XLRArmFPDP( SSHANDLE *xlrDevice* )`

Description:

`XLRArmFPDP` moves the StreamStor PCI816-XF2 from a ready to record state, to recording when an FPDP SYNC* pulse is received. StreamStor must already be in record mode, and `SS_OPT_FPDPSYNCARM` must be set. If no SYNC* pulse is received, no data will be recorded.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;

if(XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   exit(1)
}
      …

if( XLRAppend( xlrDevice ) != XLR_SUCCESS )
{
  exit(1);
}

if( XLRArmFPDP( xlrDevice ) != XLR_SUCCESS )
{
  exit(1);
}

// Waiting for SYNC pulse – data will be recorded to disk as soon
// as SYNC is received.
```

See Also:

`XLRSetDBMode`, `XLRRecord` and `XLRAppend`.

XLRBindInputChannel

Syntax:

`XLR_RETURN_CODE XLRBindInputChannel( SSHANDLE` *`xlrDevice`*`, UINT32` *`channel`* `)`

Description:

`XLRBindInputChannel` binds a channel for input INTO StreamStor. In other words, "input" is relative to StreamStor. To record on a particular channel, that channel must be bound to StreamStor via this command. `XLRClearChannels` must be called to unbind the channel(s) before calling `XLRBindInputChannel`.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`channel` is the channel number to bind – this is card specific.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Note: **CHANGING MODES CLEARS ALL INPUT AND OUTPUT CHANNELS. CHANNELS MUST BE BOUND AFTER THE MODE IS SELECTED.**

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );

// Set StreamStor mode to Single Channel.
xlrStatus = XLRSetMode( xlrDevice, SS_MODE_SINGLE_CHANNEL );
xlrStatus = XLRClearChannels( xlrDevice );

// For input over the PCI bus, select, then bind to channel zero.
xlrStatus = XLRSelectChannel( xlrDevice, 0);
xlrStatus = XLRBindInputChannel( xlrDevice, 0 );

// Now ready to record over the PCI bus.
xlrStatus = XLRRecord( xlrDevice, 0, 1 );
if( xlrStatus != XLR_SUCCESS )
{
  return(1);
}
```

See Also:

`XLRClearChannels`, `XLRBindOutputChannel`, and `XLRSelectChannel`.

XLRBindOutputChannel

Syntax:

`XLR_RETURN_CODE XLRBindOutputChannel( SSHANDLE` *`xlrDevice`*`, UINT32` *`channel`* `)`

Description:

`XLRBindOutputChannel` binds a channel for output FROM StreamStor. In other words, "output" is relative to StreamStor. To playback over a particular channel, that channel must be bound to StreamStor via this command. `XLRClearChannels` must be called to unbind the channel(s) before calling `XLRBindOutputChannel.`

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`channel` is the channel number to bind – this is card specific.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Note: **CHANGING MODES CLEARS ALL INPUT AND OUTPUT CHANNELS. CHANNELS MUST BE BOUND AFTER THE MODE IS SELECTED.**

Usage:

```
SSHANDLE          xlrDevice;
S_READDESC        readDesc;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );

// Set StreamStor mode to Single Channel.
xlrStatus = XLRSetMode( xlrDevice, SS_MODE_SINGLE_CHANNEL );
xlrStatus = XLRClearChannels( xlrDevice );

// For output over the PCI bus, select, then bind to channel zero.
xlrStatus = XLRSelectChannel( xlrDevice, 0);
xlrStatus = XLRBindOutputChannel( xlrDevice, 0 );

// Now ready to read over the PCI bus.
xlrStatus = XLRRead( xlrDevice, &readDesc );
```

See Also:

`XLRClearChannels, XLRBindInputChannel,` and `XLRSelectChannel.`

XLRCardReset

Syntax:

`XLR_RETURN_CODE XLRCardReset( UINT32 index )`

Description:

`XLRCardReset` will attempt to reset a StreamStor device and re-initialize the hardware and firmware. This function should be used only as a last resort.

Parameters:

`index` is the card index number.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
xlrReturnCode = XLRCardReset( 1 );
```

See Also:

`XLROpen` and `XLRReset`.

XLRClearChannels

Syntax:

`XLR_RETURN_CODE XLRClearChannels( SSHANDLE` *`xlrDevice`* `)`

Description:

`XLRClearChannels` unbinds all input and output channels from StreamStor. The system cannot be reading or writing, and new input and output channels must be bound before any recording or playback operation is started. `XLRClearChannels` must be called before calling `XLRBindInputChannel` or `XLRBindOutputChannel` to unbind the channels.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );
      …
xlrStatus = XLRClearChannels( xlrDevice );
      …
// Close device before exiting.
XLRClose( xlrDevice );
```

See Also:

`XLRBindInputChannel`, `XLRBindOutputChannel`, and `XLRSelectChannel`.

XLRClearOption

Syntax:

```
XLR_RETURN_CODE XLRClearOption( SSHANDLE xlrDevice,
   UINT32 options_to_clear )
```

Description:

`XLRClearOption` clears an option previously set by `XLRSetOption`, or clears all options. When an option is cleared, it is set to its default value. See `XLRSetOption` for the list of available options and default values. To clear an option, the drives must be idle (i.e., not in record or playback mode).

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen.`

`options_to_clear` is a vector of options to clear.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE              xlrDevice;
XLR_RETURN_CODE       xlrStatus;

//
// This example shows how to set options to their default values
// and how to set and clear a specific option.
//

xlrStatus = XLROpen( 1, &xlrDevice );

// Set all options to their default values.
xlrStatus = XLRClearOption( xlrDevice, SS_ALL_OPTIONS );

// Set the desired option.
xlrStatus = XLRSetOption( xlrDevice, SS_OPT_PLAYARM );
    . . .

// Clear the option.
xlrStatus = XLRClearOption( xlrDevice, SS_OPT_PLAYARM );
```

See Also:

`XLRSetOption` and `XLRGetOption.`

XLRClearWriteProtect

Syntax:

`XLR_RETURN_CODE XLRClearWriteProtect( SSHANDLE *xlrDevice* )`

Description:

`XLRClearWriteProtect` removes write protection from a previously write protected StreamStor recorder. By default, drives are not write protected. The drives must be idle (i.e., not in record mode or playback mode) to clear the write protection.

If the StreamStor is in bank mode, this command will clear write protection only on the currently selected bank.

If the StreamStor is partitioned, this command will clear write protection only on the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );
      …
xlrStatus = XLRClearWriteProtect( xlrDevice );
      …
// Close device before exiting.
XLRClose( xlrDevice );
```

See Also:

`XLRSetWriteProtect, XLRSetBankMode, XLRSelectBank, XLRGetDirectory and XLRPartitionCreate.`

XLRClose

Syntax:

`void XLRClose( SSHANDLE *xlrDevice* )`

Description:

`XLRClose` closes the StreamStor device. This should be called before exiting an application that has opened a StreamStor device with `XLROpen`. No other application can open the StreamStor device until this function has been called.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

None.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );
.
.
.
// Close device before exiting.
XLRClose( xlrDevice );
```

See Also:

`XLROpen`.

XLRDeleteAppend

Syntax:

`XLR_RETURN_CODE XLRDeleteAppend( SSHANDLE xlrDevice, UINT32 AddrHigh, UINT32 AddrLow )`

Description:

`XLRDeleteAppend` deletes the last appended data set on the StreamStor device. An appended data set is defined as the data recorded to StreamStor with the `XLRAppend` function. An optional address can be provided to define the new last append start point. Zero should be used for the address in most circumstances.

The new last append address must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will delete appended data from the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`AddrHigh` is the upper 32 bits of the 64-bit address to use for the new last append start point. In most cases, this should be zero.

`AddrLow` is the upper 32 bits of the 64-bit address to use for the new last append start point. In most cases, this should be zero.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );

// Append data.
xlrStatus = XLRAppend( xlrDevice );
.
.
.
// Stop recording.
XLRStop( xlrDevice );

// Delete just the data recorded above.
xlrStatus = XLRDeleteAppend( xlrDevice, 0, 0 );

// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

`XLRTruncate`, `XLRSetBankMode` and `XLRSelectBank`.

XLRDeviceFind

Syntax:

`UINT32 XLRDeviceFind( )`

Description:

`XLRDeviceFind` searches the PCI bus(es) and returns the number of StreamStor cards present in the system.

Parameters:

None.

Return Value:

This function returns the number of StreamStor cards in the system. If the driver has not been installed properly, this function returns zero.

Usage:

```
UINT32      NumCards;

if( NumCards = XLRDeviceFind() )
{
   // There are StreamStor cards on this system.
   printf("StreamStor cards found: %d\n", NumCards );
}
else
{
   // No StreamStor cards on the system.
   printf("No StreamStor cards detected!\n");
}
```

See Also:

`XLROpen.`

XLRDismountBank

Syntax:

`XLR_RETURN_CODE XLRDismountBank( SSHANDLE` *`xlrDevice,`* `UINT32` *`bankID`* `)`

Description:

`XLRDismountBank` will power down the selected bank. A bank can also be dismounted by a key on-off transition.

A dismounted bank can be re-powered by a key off-on transition or by calling `XLRMountBank`.

If you attempt to dismount a bank that has already been dismounted, no action is taken and XLR_SUCCESS is returned.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`bankID` is a constant indicating the bank to be dismounted (`BANK_A` or `BANK_B`).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
S_BANKSTATUS      AbankStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );
xlrStatus = XLRSetBankMode ( xlrDevice, SS_BANKMODE_NORMAL );
xlrStatus = XLRGetBankStatus ( xlrDevice, BANK_A, &AbankStatus );
if ( AbankStatus.MediaStatus == MEDIASTATUS_FULL )
{
      printf ( “BANK A is full.\n” );
      xlrStatus = XLRDismountBank ( xlrDevice, BANK_A );
}
```

See Also:

`XLRMountBank, XLRGetBankStatus, XLRSetBankMode` and `XLRSelectBank`.

XLREdit

Syntax:

XLR_RETURN_CODE XLREdit(SSHANDLE *xlrDevice*, PS_READDESC *pReadDesc*)

Description:

XLREdit edits data from the StreamStor device by overwriting existing data (specified by the AddrHi, AddrLo, and XferLength structure members of pReadDesc) with new data contained in the buffer pointed to by the BufferAddr structure member of pReadDesc.

The edit address of the requested data must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will edit the data on the currently selected bank. If the StreamStor is partitioned, this command will edit the data on the currently selected partition.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pReadDesc is a pointer to an S_READDESC structure that holds the edit address, length and buffer address containing the new data to overwrite the existing data.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE        xlrDevice;
S_READDESC      readDesc;
UINT32          myBuffer[40000];
XLR_RETURN_CODE xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
/* Fill buffer with new data here */

//AddrHi and AddrLo must represent an appropriately aligned address.
readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLREdit( xlrDevice, &readDesc );
```

See Also:

`XLREditData, XLRSetBankMode` and `XLRSelectBank`.

XLREditData

Syntax:

```
XLR_RETURN_CODE XLREditData( SSHANDLE xlrDevice,
PUINT32 BufferAddr, UINT32 AddrHigh, UINT32 AddrLow,
UINT32 XferLength )
```

Description:

`XLREditData` edits data from the StreamStor device by overwriting existing data (specified by the `AddrHi`, `AddrLo`, and `XferLength` parameters) with new data contained in the buffer pointed to by the `BufferAddr.`

This function is identical to `XLREdit` without the structure to pass the edit parameters.

The edit address of the requested data must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will edit the data on the currently selected bank. If the StreamStor is partitioned, this command will edit the data on the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`Buffer` is the address of the user memory buffer to hold the requested data.

`AddrHigh` is the upper 32 bits of a 64-bit byte address of the requested data.

`AddrLow` is the lower 32 bits of a 64-bit byte address of the requested data.

`XferLength` is the number of bytes requested.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;
UINT32            myBuffer[40000];

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …

/* Fill buffer with new data here */

xlrReturnCode = XLREditData( xlrDevice, myBuffer, 0, 0xFE120000,
sizeof(myBuffer) );
```

See Also:

`XLREdit`, `XLRSetBankMode` and `XLRSelectBank`.

XLRErase

Syntax:

```
XLR_RETURN_CODE XLRErase( SSHANDLE xlrDevice, SS_OWMODE mode )
```

Description:

`XLRErase` erases data on the drives.

If the StreamStor is in bank mode, this command will erase only the selected bank.

If the StreamStor is partitioned, the command will erase only the data within the currently selected partition. Other partitions will be unaffected. An exception to this is `XLRErase` called with the `SS_OVERWRITE_PARTITION` option. When called with this option, all partitions will be deleted from the device (or from the currently selected bank, if in bank mode).

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`mode` is the erase mode.

There are five erase modes:

- `SS_OVERWRITE_NONE` sets the directories to zero, such that the drives are reported as having no data. However, all data is still on the drives. `XLRErase` will return when this command is complete.
- `SS_OVERWRITE_RANDOM_PATTERN` overwrites all data on the drives with a random pattern so that the data is permanently deleted. `XLRErase` returns immediately, but the erasure can take several hours – use `XLRGetDeviceStatus` (see below) to find out when erasure is complete.
- `SS_OVERWRITE_RW_PATTERN` is similar to `SS_OVERWRITE_RANDOM_PATTERN` except that data is read first and then overwritten with a random pattern. This mode can be used to verify that all sectors can be read and written. Note that this mode will take on average twice as long as the `SS_OVERWRITE_RANDOM_PATTERN` mode to complete. `XLRErase` returns immediately, but the erasure can take several hours – use `XLRGetDeviceStatus` (see below) to find out when erasure is complete.
- `SS_OVERWRITE_DIRECTORY` destroys the directory locator block (for the currently selected partition, if the system is partitioned). This option will erase all data including the user directory and labels. Other partitions (if partitioned) are unaffected. `XLRErase` will return when this command is complete.
- `SS_OVERWRITE_PARTITION` destroys everything: all partitions, data, user directories, and labels. `XLRErase` will return when this command is complete.

As with other API functions that record data, `XLRErase` will immediately return control to the calling program. If an erase is in progress, `XLRGetDeviceStatus` will indicate that the device is in `Recording` mode.

If the SS_OVERWRITE_RANDOM_PATTERN mode is specified, and an overwrite operation is in progress, a call to XLRGetLength will return the number of bytes remaining to overwrite for the slowest bus. (Each bus is erased in parallel, thus it returns the number of bytes remaining for the slowest bus. In other words, when the slowest bus completes, the entire operation will be complete.)

If the SS_OVERWRITE_RW_PATTERN mode is specified, the erase is done in two passes. The first pass will read all blocks on the device and the second pass will write all blocks on the device. During the read pass, a call to XLRGetLength will return the number of bytes remaining to be read. During the write pass, a call to XLRGetLength will return the number of bytes remaining to be overwritten.

Stopping StreamStor part way through an overwrite erase will immediately set the directory length to zero. Restarting the overwrite erase will start from the beginning – not where you previously stopped.

The following table summarizes what, in addition to the data, is erased.

Erase Option Specified	Label Erased?	User Directory Erased?	Partitions Removed?
SS_OVERWRITE_NONE	No	No	No
SS_OVERWRITE_RANDOM_PATTERN	No	No	No
SS_OVERWRITE_RW_PATTERN	No	No	No
SS_OVERWRITE_DIRECTORY	Yes	Yes	No
SS_OVERWRITE_PARTITION	Yes	Yes	Yes

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturn;
UINT64            xlrLength;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn != XLR_SUCCESS )
      return(1);
xlrReturn = XLRErase( xlrDevice, SS_OVERWRITE_RANDOM_PATTERN );
if( xlrReturn != XLR_SUCCESS )
      return(1);

//
//Overwrite Erase Examples:
//Example 1: 2 20GB drives per bus - master / slave configuration
//
```

```
xlrLength = XLRGetLength( xlrDevice );
//xlrLength equals approximately 40GB – if called
//at the beginning of the erase.

//Example 2: 1 100GB drive per bus - master only configuration.
xlrLength = XLRGetLength( xlrDevice );
//xlrLength equals approximately 100GB – if called
//at the beginning of the erase.
```

See Also:

XLRSetLabel, XLRSetUserDir, XLRGetBankStatus, XLRGetLength, XLRSetWriteProtect, XLRClearWriteProtect, XLRSetBankMode, XLRSelectBank, and XLRPartitionSelect.

XLRGetBankStatus

Syntax:

XLR_RETURN_CODE XLRGetBankStatus(SSHANDLE *xlrDevice,* UINT32 *bankID*, PS_BANKSTATUS *pBankStatus*)

Description:

XLRGetBankStatus retrieves information from the StreamStor about the specified *bankID.* The StreamStor must be in bank mode to get the status of a bank. XLRGetBankStatus can be called when the StreamStor is not idle.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

bankID is a constant indicating the bank to report on (BANK_A or BANK_B).

pBankStatus is a pointer to an S_BANKSTATUS structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_BANKSTATUS      AbankStatus;
S_BANKSTATUS      BbankStatus
XLR_RETURN_CODE   xlrStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRSetBankMode ( xlrDevice, SS_BANKMODE_NORMAL );
xlrStatus = XLRGetBankStatus ( xlrDevice, BANK_A, &AbankStatus );
xlrStatus = XLRGetBankStatus ( xlrDevice, BANK_B, &BbankStatus );
if ( AbankStatus.MediaStatus == MEDIASTATUS_FULL )
{
      printf ( “BANK A is full.\n” );
}
if ( BbankStatus.MediaStatus == MEDIASTATUS_FULL )
{
      printf ( “BANK B is full.\n” );
}
```

See Also:

XLRGetDeviceStatus, XLRSetBankMode and XLRSelectBank.

XLRGetBaseAddr

Syntax:

`UINT32 XLRGetBaseAddr( SSHANDLE xlrDevice )`

Description:

`XLRGetBaseAddr` returns the physical address of the recording data window. This address can be used to program PCI hardware devices for direct card-to-card data transfer. The address returned from this function is NOT a valid user address.

If using multi-channel PCI Express and virtual channels, call `XLRSelectChannel` then bind the channels you want to use. Once all channels have been bound, to get the physical address for a specific virtual channel, call `XLRSelectChannel` to select the channel, and then call `XLRGetBaseAddr` to get the physical address.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

This function returns the physical PCI address as a 32 bit unsigned integer.

Usage:

```
UINT32            xlrAddress;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );
if( xlrStatus != XLR_SUCCESS )
{
   // Error opening StreamStor.
}
else
{
   xlrAddress = XLRGetBaseAddr( xlrDevice );
}
```

See Also:

`XLRGetBaseRange`, `XLRGetWindowAddr` and `XLRSetMode`.

XLRGetBaseRange

Syntax:

`UINT32 XLRGetBaseRange( SSHANDLE xlrDevice )`

Description:

`XLRGetBaseRange` returns the size of the StreamStor device data window in bytes. This range of addresses is intended to be used by hardware transferring data that cannot be programmed to write with a non-incrementing address. Note that the address used to write to StreamStor does not effect the storage location of the data; StreamStor always stores data sequentially in the order it is written regardless of the address.

If using multi-channel PCI Express and virtual channels, the range is divided between the channels. For example if `XLRGetBaseRange` returns 16 megabytes, if you configure two virtual input channels, each channel will have a range of 8 megabytes.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

This function returns the window size in bytes.

Usage:

```
UINT32            xlrAddress, xlrRange;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );
if( xlrStatus != XLR_SUCCESS )
{
   // Error opening StreamStor.
}
else
{
   xlrAddress = XLRGetBaseAddr( xlrDevice );
   xlrRange = XLRGetBaseRange( xlrDevice );
}
// DMA Hardware may now be programmed to write to any address from
// xlrAddress to (xlrAddress + xlrRange).
```

See Also:

`XLRGetBaseAddr`, `XLRGetWindowAddr` and `XLRSetMode`.

XLRGetDBInfo

Syntax:

`XLR_RETURN_CODE XLRDBInfo(SSHANDLE *xlrDevice,* PS_DBInfo *pdbInfo*)`

Description:

`XLRGetDBInfo` retrieves information from the StreamStor daughterboard. The drives must be idle (i.e., not in record or playback mode) to get daughterboard information.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pdbInfo` is a pointer to an `S_DBInfo` structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_DBINFO          dbInfo;
XLR_RETURN_CODE   xlrStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRGetDBInfo( xlrDevice,&dbInfo );
printf( “Daughterboard type is %s\n”, dbInfo.PCBType );
```

See Also:

`XLRGetDeviceInfo.`

XLRGetChassisType

Syntax:

`UINT32 XLRGetChassisType( SSHANDLE *xlrDevice* )`

Description:

`XLRGetChassisType` retrieves an integer value representing the chassis type. The defined values for chassis types are:

- TK200 – a chassis that can hold two drive modules and supports bank switching.
- UNKNOWN_CHASSIS_TYPE – any chassis that is not a TK200 and therefore does not support bank switching.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

The chassis type, as described above.

Usage:

```
SSHANDLE          xlrDevice;
UINT32            chassisType
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen(1, &xlrDevice);
chassisType = XLRGetChassisType(xlrDevice);
if (chassisType == TK200)
{
      printf (“This system supports bank switching.\n”);
      xlrStatus = XLRSetBankMode(xlrDevice, SS_BANKMODE_NORMAL);
      xlrStatus = XLRSelectBank(xlrDevice, BANK_B);
      …
}
else
{
      printf (“This system does not support bank switching.\n”);
}
      …
// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

`XLRSetBankMode` and `XLRSelectBank`.

XLRGetDeviceInfo

Syntax:

`XLR_RETURN_CODE XLRGetDeviceInfo( SSHANDLE` *`xlrDevice`*`, PS_DEVINFO` *`pDevInfo`* `)`

Description:

`XLRGetDeviceInfo` retrieves information from the StreamStor device about its physical configuration. The drives must be idle (i.e., not in record or playback mode) when this function is called.

If the StreamStor is in bank mode, this command will report on the currently selected bank. For example, the number of drives reported will be the number of drives on the selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pDevInfo` is a pointer to an `S_DEVINFO` structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_DEVINFO         devInfo;
XLR_RETURN_CODE   xlrReturn;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn != XLR_SUCCESS )
      return(1);
xlrReturn = XLRGetDeviceInfo( xlrDevice, &devInfo );
if( xlrReturn != XLR_SUCCESS )
      return(1);
printf("StreamStor serial number is: %d", devInfo.SerialNum );
```

See Also:

`XLRGetDBInfo`, `XLRSetBankMode` and `XLRSelectBank`.

XLRGetDeviceStatus

Syntax:

XLR_RETURN_CODE XLRGetDeviceStatus(SSHANDLE *xlrDevice*, PS_DEVSTATUS *pDevStatus*)

Description:

XLRGetDeviceStatus retrieves status of the StreamStor device.

If the StreamStor is in bank mode, this command will report the device status of the currently selected bank.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pDevStatus is a pointer to an S_DEVSTATUS structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_DEVSTATUS       devStatus;
XLR_RETURN_CODE   xlrReturn;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn != XLR_SUCCESS )
      return(1);
xlrReturn = XLRGetDeviceStatus( xlrDevice, &devStatus );
if( xlrReturn != XLR_SUCCESS )
      return(1);
if( devStatus.Recording )
      printf("StreamStor is recording.");
else
      printf("StreamStor is idle");
```

See Also:

XLRGetBankStatus, XLRSetBankMode and XLRSelectBank.

XLRGetDirectory

Syntax:

`XLR_RETURN_CODE XLRGetDirectory( SSHANDLE` *`xlrDevice`*`, PS_DIR` *`pDir`*
`)`

Description:

`XLRGetDirectory` gets the directory information of the current recording on a StreamStor device. The drives must be idle (i.e., not in record or playback mode) when this function is called.

If the StreamStor is in bank mode, this command will report directory information on the currently selected bank.

If data on the StreamStor was recorded in multi-channel mode, this command will report directory information for data recorded on the currently selected channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pDir` is a pointer to an `S_DIR` structure to be filled by this function call.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
S_DIR             xlrDir;

xlrStatus = XLROpen( 1, &xlrDevice );

 … Record some data …

XLRStop(xlrDevice);
xlrStatus = XLRGetDirectory( xlrDevice, &xlrDir );
if( xlrStatus != XLR_SUCCESS )
{
      return(1);
}
```

See Also:

`XLRGetLength`, `XLRSetMode`, `XLRSelectChannel`, `XLRSetBankMode` and `XLRSelectBank`.

XLRGetDriveInfo

Syntax:

```
XLR_RETURN_CODE XLRGetDriveInfo( SSHANDLE xlrDevice, UINT32 Bus,
UINT32 MasterSlave, PS_DRIVEINFO pDriveInfo )
```

Description:

`XLRGetDriveInfo` retrieves info from the StreamStor drive about its physical configuration. The drives must be idle (i.e., not in record or playback mode) when this function is called.

If the StreamStor is in bank mode, this command will get drive information for the drives in the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`Bus` is the ATA bus number of the drive.

`MasterSlave` is `XLR_MASTER_DRIVE` (0) or `XLR_SLAVE_DRIVE` (1) to select the master or slave drive on the ATA bus.

`pDriveInfo` is a pointer to an `S_DRIVEINFO` structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_DRIVEINFO       drvInfo;
XLR_RETURN_CODE   xlrReturn;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn != XLR_SUCCESS )
      return(1);
xlrReturn = XLRGetDriveInfo( xlrDevice, 0, XLR_MASTER_DRIVE, &drvInfo
);
if( xlrReturn != XLR_SUCCESS )
      return(1);
printf( “Drive serial number is: %s\n”, drvInfo.Serial );
printf( “Drive model number is: %s\n”, drvInfo.Model );
printf( “Drive revision: %s\n”, drvInfo.Revision );
printf( “Drive capacity (sectors): %d\n”, drvInfo.Capacity );
```

See Also:

`XLRSetBankMode` and `XLRSelectBank`.

XLRGetDriveTemp

Syntax:

```
XLR_RETURN_CODE XLRGetDriveTemp( SSHANDLE xlrDevice, UCHAR Bus,
UCHAR MasterSlave, PUINT32 temp )
```

Description:

XLRGetDriveTemp retrieves the current temperature (in C) reading from the disk drive. Drive temperature information is retrieved from the drive's SMART ("Self-Monitoring Analysis and Reporting Technology") data. Some drive models are not SMART-capable, so temperature information cannot be retrieved by XLRGetDriveTemp.

XLRGetDriveTemp is supported on a limited number of drive models. Each disk drive vendor has its own technique for storing drive temperature data. Therefore, you should independently verify that the drive temperatures reported by XLRGetDriveTemp are accurate for your drives.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

Bus is the ATA bus number of the drive.

MasterSlave is XLR_MASTER_DRIVE (0) or XLR_SLAVE_DRIVE (1) to select the master or slave drive on the ATA bus.

temp is a pointer to an unsigned integer to be filled by this function call.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
PUINT32           temp;
XLR_RETURN_CODE   xlrReturn;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn != XLR_SUCCESS )
      return(1);
xlrReturn = XLRGetDriveTemp( xlrDevice, 0, XLR_MASTER_DRIVE, &temp );
if( xlrReturn != XLR_SUCCESS )
      return(1);
printf( "Drive temperature on Bus 0 Master is: %d degrees C\n", temp );
```

XLRGetErrorMessage

Syntax:

`XLR_RETURN_CODE XLRGetErrorMessage(char *string, XLR_ERROR_CODE err)`

Description:

`XLRGetErrorMessage` returns the error message of the most recent API failure.

Parameters:

`string` is a pointer to a string to accept the error message of at least `XLR_ERROR_LENGTH` size.

`err` is an error code returned from XLRGetLastError.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrHandle;
S_DIR             xlrDir;
XLR_RETURN_CODE   xlrReturn;
XLR_ERROR_CODE    xlrError;
char              temp[XLR_ERROR_LENGTH];

xlrStatus = XLROpen( 1, &xlrDevice );
      …
xlrReturn = XLRGetDirectory( xlrHandle, &xlrDir );
if( xlrReturn != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( temp, xlrError );
   printf( “Error message: %s\n”, temp );
   exit(1);
}
```

See Also:

`XLRGetLastError.`

XLRGetEvents

Syntax:

```
XLR_RETURN_CODE XLRGetEvents ( IN SSHANDLE xlrDevice,
   IN UINT32 bufSize, OUT PS_EVENTS events );
```

Description:

`XLRGetEvents` retrieves events that were captured as a result of setting one or more event options when calling the `XLRSetDBMode` function. (I.e., calling `XLRSetDBMode` with *`option`* set to `SS_OPT_EVENTLOG` bit-wise inclusively or'ed with an `SS_OPT_EVENT_*` option.)

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`bufsize` is the size, in bytes, of *`events`*.

`events` is a pointer to the `S_EVENTS` structure that is to receive the events.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrHandle;
XLR_RETURN_CODE   xlrStatus;
UINT32            eventCount=0;
UINT32            i=0;
PS_EVENTS         eventBufPtr = NULL;

xlrStatus = XLROpen( 1, &xlrHandle );

      // Select the desired channels….

//
// Set the FPDP mode and select the type of event to
// capture.  This shows how to capture events on
// rising edge of SYNC* signal on a PCI-816XF2.  Note that
// to use any of the SS_OPT_EVENT_* options, you must “or”
// the option (or options) with SS_OPT_EVENTLOG.
//
xlrStatus = XLRSetDBMode( xlrHandle,
   SS_FPDP_RECVMASTER, SS_OPT_FPDPEVENTLOG | SS_OPT_EVENT_SYNC_RISE );

      // Record some data …

xlrStatus = XLRStop( xlrHandle );

//
// Get the number of events that were captured so you know
// how much space to allocate to hold them.
//
eventCount = XLRGetEventsLength( xlrHandle );
eventBufPtr = (PS_EVENTS)malloc( eventCount * sizeof(S_EVENTS) );

//
// Retrieve the events into the array.
//
xlrStatus = XLRGetEvents( xlrHandle,
   eventCount * sizeof(S_EVENTS), eventBufPtr );

//
// Examine the events.
//
for ( i = 0; i < eventCount; i++ )
{
   printf(“Event[%u].Source = 0x%X - ", i, eventBufPtr[i].Source );
   printf( "Address: 0x%X%X\n",
      eventBufPtr[i].AddressHigh,
      eventBufPtr[i].AddressLow );
}
```

See Also:

XLRSetDBMode, XLRGetEventsLength and XLRRetrieveEvents.

XLRGetEventsLength

Syntax:

```
UINT32 XLRGetEventsLength( SSHANDLE xlrDevice )
```

Description:

XLRGetEventsLength returns the number of events that have been captured as a result of setting one or more event options when calling the XLRSetDBMode function. (I.e., calling XLRSetDBMode with *option* set to SS_OPT_EVENTLOG bit-wise inclusively or'ed with an SS_OPT_EVENT_* option.)

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

Return Value:

The number of events that have been captured.

Usage:

```
SSHANDLE          xlrHandle;
UINT32            numberOfEvents=0;
XLR_RETURN_CODE   xlrReturn;

xlrStatus = XLROpen( 1, &xlrDevice );

      // Capture events …

numberOfEvents = XLRGetEventsLength( xlrHandle );
```

See Also:

XLRSetDBMode, XLRRetrieveEvents and XLRGetEvents.

XLRGetFIFOLength

Syntax:

`UINT64 XLRGetFIFOLength( SSHANDLE xlrDevice )`

Description:

`XLRGetFIFOLength` returns the amount of data currently in the FIFO. This function is only valid when StreamStor is in a forking or pass thru mode (`SS_MODE_FORK`, `SS_MODE_PASSTHRU`). If StreamStor is not in one of the modes listed above, or is not currently moving data, `XLRGetFIFOLength` will return 0.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

Usage:

```
SSHANDLE          xlrDevice;
UINT64            length = 0;

...

// Setup StreamStor in a valid fork or pass-thru mode

...
// Get the length of data in the fifo.
length = XLRGetFIFOLength( xlrDevice );
```

See Also:

`XLRSetMode`, `XLRReadFIFO` and `XLRGetLength`.

XLRGetLabel

Syntax:

`XLR_RETURN_CODE XLRGetLabel( SSHANDLE *xlrDevice*, char **label* )`

Description:

`XLRGetLabel` returns the label on the StreamStor recorder where the label was previously set with the `XLRSetLabel` command. If no label has been previously set, a default label will be returned.

If the StreamStor is in bank mode, this command will return the label of the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`label` is a pointer to a string to accept a label of at least `XLR_LABEL_LENGTH` in size.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
char              label[XLR_LABEL_LENGTH];

xlrStatus = XLROpen( 1, &xlrDevice );
xlrStatus = XLRSetLabel( xlrDevice, “Label 2” );
xlrStatus = XLRGetLabel( xlrDevice, label );
printf ( “This disk set is labeled %s\n”, label );
      …
// Null out the label.
label[0] =’\0’;
XlrStatus = XLRSetLabel( xlrDevice, label );
      …
// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

`XLRSetLabel`, `XLRErase`, `XLRSetBankMode` and `XLRSelectBank`.

XLRGetLastError

Syntax:

```
XLR_ERROR_CODE XLRGetLastError( void )
```

Description:

`XLRGetLastError` returns the error code of the most recent API failure. This function should always be called immediately after any StreamStor API function call that returns failure.

It is not meaningful to call `XLRGetLastError` if the last StreamStor API function call was successful. In this case, the returned error code will be error code 3 (`XLR_ERR_NOINFO`).

Parameters:

None.

Return Value:

This function returns the error code. Error codes are listed in Appendix A.

Usage:

```
SSHANDLE          xlrDevice;
XLR_ERROR_CODE    xlrError;
char              errString[XLR_ERROR_LENGTH];

xlrStatus = XLROpen( 1, &xlrDevice );
      …
xlrReturn = XLRGetDirectory( xlrDevice, &xlrDir );
if( xlrReturn != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( errString, xlrError );
   printf( “Error message: %s\n”, errString );
   exit(1);
}
```

See Also:

`XLRGetErrorMessage.`

XLRGetLength

Syntax:

`UINT64 XLRGetLength( SSHANDLE xlrDevice )`

Description:

`XLRGetLength` returns the length (in bytes) of the current recording as a 64-bit integer. This function can be used during an active recording or FIFO operation. Note that during active record and during FIFO operations, the returned value may not be exact since data is still moving between devices.

If the StreamStor is in bank mode, this command will return the length of data on the currently selected bank.

If data on the StreamStor was recorded in multi-channel mode, this command will return the length of data recorded on the currently selected channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

Current recording length in bytes.

Usage:

```
SSHANDLE          xlrHandle;
UINT64            recordingLength;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrHandle );
recordingLength = XLRGetLength( xlrHandle );
```

See Also:

`XLRGetDirectory`, `XLRGetBankStatus`, `XLRSelectBank`, `XLRSetMode` and `XLRSelectChannel`.

XLRGetLengthLowHigh

Syntax:

```
void XLRGetLengthLowHigh( SSHANDLE xlrDevice, PUINT32 low,
PUINT32 high )
```

Description:

`XLRGetLengthLowHigh` returns the current recording length (in bytes) in two 32-bit variables. This function is provided for programming environments unable to handle 64 bit integers.

If the StreamStor is in bank mode, the values returned will be for the recording on the currently selected bank.

If data on the StreamStor was recorded in multi-channel mode, the values returned will be for the data recorded on the currently selected channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`low` is a pointer to a `UINT32` (unsigned int) that will be written with the lower 32 bits of the recording size in bytes.

`high` is a pointer to a `UINT32` (unsigned int) that will be written with the upper 32 bits of the recording size in bytes.

Return Value:

None

See Also:

`XLRGetDirectory`, `XLRGetBankStatus`, `XLRSelectBank`, `XLRSetMode` and `XLRSelectChannel`.

XLRGetLengthPages

Syntax:

`UINT32 XLRGetLengthPages( SSHANDLE xlrDevice )`

Description:

`XLRGetLengthPages` returns the current recording length in units of system pages. This function is provided for programming environments unable to handle 64 bit integers. Windows environments typically utilize a page size of 4096 bytes but this should be checked using a query to the operating system.

If the StreamStor is in bank mode, the value returned will be for the recording on the currently selected bank.

If data on the StreamStor was recorded in multi-channel mode, the values returned will be for the data recorded on the currently selected channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

Recording length in system pages.

See Also:

`XLRGetDirectory`, `XLRGetBankStatus`, `XLRSelectBank`, `XLRSetMode` and `XLRSelectChannel`.

XLRGetMode

Syntax:

`XLR_RETURN_CODE XLRGetMode( SSHANDLE` *`xlrDevice`*`, SSMODE` *`pMode`* `)`

Description:

`XLRGetMode` returns the input/output path (or "port mode") on the StreamStor recorder where the mode was previously set with the `XLRSetMode` command.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pmode` is a pointer to an `SSMODE` variable that will receive the mode.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrHandle;
XLR_RETURN_CODE   xlrStatus;
SSMODE            portMode;

xlrStatus = XLROpen( 1, &xlrDevice );
xlrStatus = XLRGetMode(xlrDevice, &portMode);

if( portMode == SS_MODE_SINGLE_CHANNEL )
{
   printf( "In single channel mode.\n" );
}
…
```

See Also:

`XLRSetMode.`

XLRGetOption

Syntax:

```
XLR_RETURN_CODE XLRGetOption( SSHANDLE xlrDevice, UINT32
options_to_get, PBOOLEAN options_on )
```

Description

XLRGetOption is used to determine if one or more options are set.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

options_to_get is a vector of options to query.

options_on is the returned BOOLEAN indicating if all the options in *options_to_get* are set. If TRUE, all of the options in *options_to_get* are set. If set to FALSE, one or more of the options in *options_to_get* are not set.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE              xlrDevice;
XLR_RETURN_CODE       xlrStatus;
BOOLEAN               options_on;

xlrStatus = XLROpen( 1, &xlrDevice );

// See if an option is set.
xlrStatus = XLRGetOption( xlrDevice, SS_OPT_PLAYARM, &options_on );
if ( options_on == TRUE )
{
   printf ( "PlayArm option is set.\n" );
}
else
{
   printf ( "PlayArm option is not set.\n" );
}
```

See Also:

XLRSetOption and XLRClearOption.

XLRGetPartitionInfo

Syntax:

XLR_RETURN_CODE XLRGetPartitionInfo(SSHANDLE *xlrDevice,*
PS_PARTITIONINFO *pPartitionInfo*)

Description:

XLRGetPartitionInfo retrieves information from the StreamStor about the currently selected partition.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pPartitionInfo is a pointer to an S_PARTITIONINFO structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_PARTITIONINFO   partitionInfo;

// Open the device.
if( XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   printf( “ERROR: Open failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

if( XLRGetPartitionInfo( xlrDevice, &partitionInfo ) != XLR_SUCCESS )
{
   printf( “ERROR: GetPartitionInfo failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

printf (“Number of partitions = %u\n”, partitionInfo.NumPartitions);
```

See Also:

XLRPartitionCreate, XLRPartitionDelete and XLRPartitionSelect.

XLRGetPlayBufferStatus

Syntax:

```
XLR_RETURN_CODE XLRGetPlayBufferStatus( SSHANDLE xlrDevice,
PUINT32 status )
```

Description

`XLRGetPlayBufferStatus` retrieves the status of the playback buffer. The playback buffer is used when the playback trigger is armed. See the `SS_OPT_PLAYARM` option of the `XLRSetOption` command.

Parameters:

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen.`
- *`status`* is a pointer to a `UINT32` that will receive the playback buffer status, where status is one of:
  - `SS_PBS_IDLE` – The playback buffer is not in use.
  - `SS_PBS_FULL` – The playback buffer is full and a playback is not in progress.
  - `SS_PBS_FILLING` – Data is streaming into the playback buffer.
  - `SS_PBS_PLAYING` – Data from the playback buffer is playing.
  - `SS_PBS_UNKNOWN` – The status of the playback buffer cannot be determined.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
// Example showing how to use the playback trigger and get
// the play buffer status.

SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
UINT32            playBuffStatus;

XLROpen( 1, &xlrDevice );

// Setup channel for playback …

// Prepare for playback.
xlrStatus = XLRSetOption( xlrDevice, SS_OPT_PLAYARM );

//
// Since the playback buffer is not in use yet, the status returned
// here should be SS_PBS_IDLE.
//
xlrStatus = XlrGetPlayBufferStatus( xlrDevice, &playBuffStatus );

//
// Since SS_OPT_PLAYARM has been set, this call to XLRPlayback
// will not cause the data to flow yet.
//
xlrStatus = XLRPlayback( xlrDevice, 0, 0 );

//
// XLRPlayback has been called, but XLRPlayTrigger has not.  The status
// of the playback buffer should be SS_PBS_FILLING now.
//
xlrStatus = XlrGetPlayBufferStatus( xlrDevice, &playBuffStatus );

//
// Call XlrGetPlayBufferStatus periodically until the status of
// the playback buffer is SS_PBS_FULL.
//
            . . .
//
// Buffer is now full, so call XLRPlayTrigger to start the data
// flowing out of the buffer.
//
xlrStatus = XLRPlayTrigger( xlrDevice );

 ... playback the desired length of time . . .

// Stop the playback.
XLRStop( xlrDevice );
```

See Also:

XLRPlayTrigger, XLRSetOption, and XLRPlayBack.

XLRGetPlayLength

Syntax:

`UINT64 XLRGetPlayLength( SSHANDLE` *`xlrDevice`* `)`

Description:

`XLRGetPlayLength` returns the number of bytes that have been played back between calling `XLRPlayback` and `XLRStop`.

You can call `XLRGetPlayLength` while the StreamStor is playing back. In this case, however, the number of bytes played back is only an estimate (because the number of bytes played back is updated internally approximately every three seconds).

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

Number of bytes played back. Zero is returned if an error occurred.

Usage:

```
SSHANDLE         xlrDevice;
UINT32           addrHi, addrLow;
UINT64           bytesPlayed;
char             errMessage[XLR_ERROR_LENGTH];
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
addrHi = 0;
addrLow = 0xFE120000;

xlrReturnCode = XLRPlayback( xlrDevice, addrLow, addrHi );
      …
XLRStop(xlrDevice);

// Get the number of bytes that were played back.
bytesPlayed = XLRGetPlayLength(xlrDevice);
if ( bytesPlayed == 0 ) {
   printf (“Nothing got played back.\n”);
   XLRGetErrorMessage( errMessage, XLRGetLastError() );
   printf( "XLRGetPlayLength error: %s\n", errMessage );
}
```

See Also:

`XLRPlayback` and `XLRSetPlaybackLength`.

XLRGetRecordedChannelInfo

Syntax:

`XLR_RETURN_CODE XLRGetRecordedChannelInfo(SSHANDLE` *`xlrDevice,`*
`PS_RECDCHANNELINFO` *`pRecdChannelInfo`*`)`

Description:

`XLRGetRecordedChannelInfo` retrieves information about the number of channels recorded and which channel numbers are recorded on this StreamStor device.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pRecdChannelInfo` is a pointer to a `S_RECDCHANNELINFO` structure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_RECDCHANNELINFO channelInfo;
XLR_RETURN_CODE   xlrStatus;
int               i;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRGetRecordedChannelInfo( xlrDevice, &channelInfo );
printf( “Number of Channels recorded %u\n”,
         channelInfo.NumChannelsRecorded );

for( i=0; i < channelInfo.NumChannelsRecorded; i++ )
{
      printf( “Channel number recorded: %u\n”,
         channelInfo.RecordedChannelNumber[i]);

}
```

XLRGetSample

Syntax:

`XLR_RETURN_CODE XLRGetSample( SSHANDLE` *`xlrDevice,`* `UINT32` *`bufSize,`*
`PUINT32` *`pBuffer`* `)`

Description:

`XLRGetSample` retrieves sample data from StreamStor during a recording. Prior to calling `XLRGetSample`, you must call `XLRSetSampleMode` to place the StreamStor in sampling mode. In order to retrieve samples, data must be actively transferring to the StreamStor.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`bufSize` is the size of the buffer pointed to by *`pBuffer`*. This size *must* match the size specified in the call to `XLRSetSampleMode`.

`pBuffer` is the pointer to the buffer to hold the sample data.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
#define SAMPLE_SIZE 0x800000
SSHANDLE          xlrDevice;
PUINT32           pBuffer = NULL;
XLR_RETURN_CODE   xlrStatus;


xlrStatus = XLROpen( 1, &xlrDevice );
      …
pBuffer = (PUINT32)malloc(SAMPLE_SIZE);

// Turn on sampling.
xlrStatus = XLRSetSampleMode( xlrDevice,
     SAMPLE_SIZE, SS_SAMPLEMODE_NORMAL );

// Start recording.
xlrStatus = XLRRecord( xlrDevice, 0, 1 );

// Wait a few seconds to get data flowing…

// Request first sample.
xlrStatus = XLRGetSample( xlrDevice, SAMPLE_SIZE, pBuffer );
      …
```

See Also:

`XLRSetSampleMode`.

XLRGetSFPDPInterfaceStatus

Syntax:

```
XLR_RETURN_CODE XLRGetSFPDPInterfaceStatus(SSHANDLE xlrDevice,
UINT32 portNumber, PS_SFPDPStatus pSFPDPStatus)
```

Description:

XLRGetSFPDPInterfaceStatus retrieves information from the StreamStor SFPDP daughterboard indicating the current status of one of the four SFPDP ports. Note: This is only supported when a SFPDP Daughter Board is present. If your SFPDP Daughter Board has only two lasers (two SFPDP ports), then the SFDPDP status for SFPDP Ports 3 and 4 is not valid.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

portNumber is the SFPDP port number used to select which SFPDP port data is to be returned in the S_SFPDPStatus structure. Valid port numbers are 1, 2, 3, and 4.

pSFPDPStatus is a pointer to an S_SFPDPStatus structure that the SFPDP Status data is returned in.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_SFPDPStatus     sfpdpStatus;
XLR_RETURN_CODE   xlrStatus;

// Open the device.
xlrStatus = XLROpen( 1, &xlrDevice );

// Retrieve the SFPDP Status for SFPDP Port number 2
xlrStatus = XLRGetSFPDPInterfaceStatus( xlrDevice, 2, &sfpdpStatus );
if( sfpdpStatus.PortOpticalEngPrsnt )
      printf("SFPDP Port 2 Optical Energy Present");
```

See Also:

XLRGetRecordedChannelInfo

XLRGetSystemAddr

Syntax:

`UINT32 XLRGetSystemAddr( SSHANDLE` *`xlrDevice`* `)`

Description:

`XLRGetSystemAddr` returns the kernel address of the recording data window. This address can be used from device drivers or other kernel level software. The address returned from this function is NOT a valid user address.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

This function returns the physical PCI address as a 32 bit unsigned integer.

Usage:

```
UINT32            xlrAddress;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );
if( xlrStatus != XLR_SUCCESS )
{
   // Error opening StreamStor.
}
else
{
   xlrAddress = XLRGetSystemAddr( xlrDevice );
}
```

XLRGetUserDir

Syntax:

```
XLR_RETURN_CODE XLRGetUserDir( SSHANDLE xlrDevice, UINT32
xferLength,
    UINT32 offset, PVOID udirPtr)
```

Description:

`XLRGetUserDir` returns the user directory on the StreamStor recorder where the user directory was previously set with the `XLRSetUserDir` command.

If the StreamStor is in bank mode, this command will return the user directory of the currently selected bank.

If the StreamStor is partitioned, this command will return the user directory of the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`xferLength` is the length of the user directory. The maximum size of a user directory is `XLR_MAX_UDIR_LENGTH`.

`offset` is the beginning offset into the user directory.

`udirPtr` is a pointer to a buffer large enough to hold the expected user directory.

Note: This command can be very slow over the remote interface.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
char              userDirBuff[1024];
UINT32            cirLength;

xlrStatus = XLROpen( 1, &xlrDevice );
dirLength = XLRGetUserDirLength( xlrDevice );
if ( dirLength == 0 ) {
   printf (“This system does not have a user directory.\n”);
}
else {
   xlrStatus = XLRGetUserDir ( xlrDevice, dirLength, 0, userDirBuff );
}
      …
// Close device before exiting
```

```
XLRClose( xlrDevice );
```

See Also:

XLRSetUserDirectory, XLRGetUserDirLength, XLRSetBankMode and XLRSelectBank.

XLRGetUserDirLength

Syntax:

`UINT32 XLRGetUserDirLength( SSHANDLE xlrDevice )`

Description:

`XLRGetUserDirLength` returns the length (in bytes) of the user directory on the StreamStor recorder where the user directory was previously set with the `XLRSetUserDir` command.

If the StreamStor is in bank mode, this command will return the length of the user directory on the currently selected bank.

If the StreamStor is partitioned, this command will return the length of the user directory on the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

Length of the user directory.

Usage:

```
UINT32                  xlrAddress;
SSHANDLE                xlrDevice;
XLR_RETURN_CODE         xlrStatus;
char                    userDirBuff[1024];
UINT32                  dirLength;
UINT32                  offset;

xlrStatus = XLROpen( 1, &xlrDevice );
dirLength = XLRGetUserDirLength( xlrDevice );
if ( dirLength == 0 )
{
   printf ( “This system does not have a user directory.\n” );
}
else
{
   offset = 0;
   xlrStatus = XLRGetUserDir( xlrDevice, dirLength, offset, userDirBuff
);
}
      …
// Close device before exiting.
XLRClose( xlrDevice );
```

See Also:

`XLRSetUserDir`, `XLRGetUserDir`, `XLRSetBankMode` and `XLRSelectBank`.

XLRGetVersion

Syntax:

XLR_RETURN_CODE XLRGetVersion(SSHANDLE *xlrDevice*, PS_XLRSWREV *pVersion*)

Description:

XLRGetVersion gets the API and firmware version information from a StreamStor device.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pVersion is a pointer to an S_XLRSWREV structure to hold the version strings returned.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE     xlrDevice;
S_XLRSWVER   swVersion;
char         temp[XLR_ERROR_LENGTH];

xlrStatus = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRGetVersion( xlrDevice, &swVersion );
if( xlrReturnCode != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( temp, xlrError );
   printf( “%s\n”, temp );
   exit(1);
}
printf(“Firmware version: %s\n”, swVersion.FirmwareVersion );
```

See Also:

XLRApiVersion and XLRGetDBInfo.

XLRGetWrapLength

Syntax:

`UINT64 XLRGetWrapLength( SSHANDLE xlrDevice )`

Description:

`XLRGetWrapLength` returns the amount of data recorded plus the amount of data overwritten during a wrapped recording.

If the StreamStor is in bank mode, this command will return the wrap length of data on the currently selected bank.

If data on the StreamStor was recorded in multi-channel mode, this command will return the wrap length for the currently selected channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

The wrap length, in bytes, returned as a 64 bit integer.

Usage:

```
SSHANDLE          xlrHandle;
UINT64            wrapLength;
UINT64            recordedLength;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrHandle );

// Record data in wrap mode.
xlrReturnCode = XLRRecord(xlrHandle, 1,1);

 . . . record until the recording wraps . . .

xlrReturnCode = XLRStop(xlrHandle);

// Get the amount of data that is available for reading
recordedLength = XLRGetLength(xlrHandle);

// Get the total amount of data recorded, i.e.,
// recordedLength + the number of overwritten bytes.
wrapLength = XLRGetWrapLength(xlrHandle);
```

See Also:

`XLRGetDirectory`, `XLRGetLength`, `XLRGetBankStatus`, `XLRSelectBank`, `XLRSetMode` and `XLRSelectChannel`.

XLRGetWindowAddr

Syntax:

`PUINT32 XLRGetWindowAddr( SSHANDLE` *`xlrDevice`* `)`

Description:

`XLRGetWindowAddr` returns the user virtual address of the recording data window. This address can be used to directly write data to the StreamStor device from a user program.

If using multi-channel PCI Express and virtual channels, call `XLRSelectChannel` then bind the channels you want to use. Once all channels have been bound, to get the user virtual address for a specific virtual channel, call `XLRSelectChannel` to select the channel, and then call `XLRGetWindowAddr` to get the virtual address for the channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

This function returns a pointer to the data window mapped into the user virtual address space.

Usage:

```
PUINT32           xlrAddress;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturn;

xlrReturn = XLROpen( 1, &xlrDevice );
if( xlrReturn == XLR_SUCCESS )
{
   xlrAddress = XLRGetWindowAddr( xlrDevice );
   *xlrAddress = someData;

   // someData has been written to the StreamStor device.  Note that
   // xlrAddress does not need to be incremented for subsequent writes.
}
```

See Also:

`XLRGetBaseAddr`, `XLRGetBaseRange` and `XLRSetMode`.

XLRMountBank

Syntax:

XLR_RETURN_CODE XLRMountBank(SSHANDLE *xlrDevice,* UINT32 *bankId*)

Description:

XLRMountBank will power up the selected bank. A dismounted bank can also be re-powered by a key off-on transition.

A mounted bank can be powered off by a key on-off transition or by calling XLRDismountBank.

If you attempt to mount a bank that is already mounted, no action is taken and XLR_SUCCESS is returned.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

bankID is a constant indicating the bank to be mounted (BANK_A or BANK_B).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
S_BANKSTATUS      AbankStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRSetBankMode ( xlrDevice, SS_BANKMODE_NORMAL );
xlrStatus = XLRGetBankStatus ( xlrDevice, BANK_A, &AbankStatus );
if ( AbankStatus.MediaStatus == MEDIASTATUS_FULL )
{
      printf ( “BANK A is full.  Wait for bank to dismount\n” );
      printf ( “then insert new bank module into BANK A.\n” );
      xlrStatus = XLRDismountBank ( xlrDevice, BANK_A );
            … wait for new bank to be inserted …
      xlrStatus = XLRMountBank (xlrDevice, BANK_A);
}
```

See Also:

XLRDismountBank, XLRGetBankStatus, XLRSetBankMode and XLRSelectBank.

XLRNetOpen

Syntax:

```
XLR_RETURN_CODE XLRNetOpen( UINT32 devType, const char *address,
USHORT port, SSHANDLE *pXlrHandle )
```

Description:

`XLRNetOpen` opens a remote StreamStor device over an Ethernet link and initializes the hardware and firmware in preparation for recording on an external interface. The device is transitioned to system ready state if required. This function must be called before any other API function if using an Ethernet interface to StreamStor. After successful completion of this function, the handle pointed to by *`pXlrHandle`* can be used for all subsequent API calls.

NOTE: You should call `XLRClose` even if `XLRNetOpen` returns `XLR_FAIL`.

Parameters:

`devType` identifies the type of StreamStor to open, where *`devType`* is one of:

- `SS_NET_TYPE_LTX2` – indicates the device to be opened is an LTX2.
- `SS_NET_TYPE_REMOTE` – indicates the device to be opened is a remote StreamStor server.

`address` is a pointer to an array with a valid IPv4 address in dotted-quad notation (xxx.xxx.xxx.xxx), i.e. (“127.0.0.1”).

`port` indicates which network port the connection to StreamStor should be made on (default is 10001).

`pXlrHandle` is a pointer to a system handle for initialization. Successful completion loads this parameter with a valid handle to the hardware device to use in subsequent API calls. *`*pXlrHandle`* is assigned the value `INVALID_SSHANDLE` on failure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrHandle;
UINT32            xlrError;
char              errString[XLR_ERROR_LENGTH];

if( XLRNetOpen( SS_NET_TYPE_LTX2, “127.0.0.1”, 10001, &xlrHandle ) !=
XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( errString, xlrError );
   printf( “%s\n”, errString );
   XLRClose( xlrHandle );
   exit(1);
}
.
.
.
XLRClose( xlrHandle );
```

See Also:

XLRClose.

XLRNetCardReset

Syntax:

```
XLR_RETURN_CODE XLRNetCardReset( UINT32 devType, const char
*address, USHORT port )
```

Description:

`XLRNetCardReset` resets the remote StreamStor device over an Ethernet link and resets the device. See `XLRCardReset` for more information. The device is closed after returning from this function and must be opened normally with `XLRNetOpen`.

Parameters:

devType identifies the type of StreamStor to reset, where *devType* is one of:

- `SS_NET_TYPE_LTX2` – indicates the device to be opened is an LTX2.
- `SS_NET_TYPE_REMOTE` – indicates the device to be opened is a remote StreamStor server.

address is a pointer to an array with a valid IPv4 address in dotted-quad notation (xxx.xxx.xxx.xxx), i.e. (“127.0.0.1”). It identifies the StreamStor device to reset.

port indicates which network port the connection to StreamStor should be made on (default is 10001).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
UINT32            xlrError;
char              errString[XLR_ERROR_LENGTH];

if( XLRNetCardReset( SS_NET_TYPE_LTX2, “127.0.0.1”, 10001 ) !=
XLR_SUCCESS )
{
   xlrError = XLRGetLastError();
   XLRGetErrorMessage( errString, xlrError );
   printf( “Reset failed.  Error = %s\n”, errString );
   exit(1);
}
```

See Also:

XLRClose.

XLROpen

Syntax:

`XLR_RETURN_CODE XLROpen( UINT32 devIndex, SSHANDLE *pXlrHandle )`

Description:

`XLROpen` opens a StreamStor device and initializes the hardware and firmware in preparation for recording. The device is transitioned to system ready state if required. This function must be called before any other API function. After successful completion of this function, the handle pointed to by *`pXlrHandle`* can be used for all subsequent API calls.

NOTE: You should call `XLRClose` even if `XLROpen` returns `XLR_FAIL`.

Parameters:

`devIndex` identifies the desired StreamStor to open when multiple StreamStor devices are in use. Use 1 for single card systems. Use `XLRDeviceFind` to find the number of devices installed.

`pXlrHandle` is a pointer to a system handle for initialization. Successful completion loads this parameter with a valid handle to the hardware device to use in subsequent API calls. *`*pXlrHandle`* is assigned the value `INVALID_SSHANDLE` on failure.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrHandle;
XLR_RETURN_CODE   xlrReturnCode;
UINT32            xlrError;
char              errString[XLR_ERROR_LENGTH];

xlrReturnCode = XLROpen( 1, &xlrHandle );
if( xlrReturnCode != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( errString, xlrError );
   printf( "%s\n", errString );
   XLRClose( xlrHandle );
   exit(1);
}
.
.
.
XLRClose( xlrHandle );
```

See Also:

`XLRClose, XLRDeviceFind, XLRSetBankMode, and XLRSelectBank.`

XLRPartitionCreate

Syntax:

```
XLR_RETURN_CODE XLRPartitionCreate( SSHANDLE xlrDevice, UINT64
length )
```

Description:

`XLRPartitionCreate` will create a new partition following any previously created partitions. A maximum of `XLR_MAX_PARTITIONS` partitions are allowed.

Note that you cannot create a partition at a specific offset on the StreamStor device. Instead, the StreamStor will determine where to create the partition.

The size of the partition that is created will not be exactly the size requested (i.e., it will not equal *`length`*). Because of hardware limitations, etc. the size of the partition is rounded up to certain block boundaries, so it will be bigger than *`length`*. Also, note that each partition created has overhead associated with it. Therefore, the total capacity of a system is not available for partitioning, since a portion of the available space is reserved to manage the partition overhead. The estimated overhead can be calculated using the number of drives and the factor `XLR_PARTOVERHEAD_BYTES`. Please see the example program `XLRPartitionExample.c` in the SDK for sample calculations.

You must create the first partition on an empty StreamStor device. That is, if the StreamStor has data on it that was created when the system was not partitioned, the data must first be erased. Then you can create partitions. Once a device is partitioned, you can create new partitions on it at any time.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`length` is the size in bytes of the partition to create. On so-called Generation 5 boards such as the Amazon Express, you can specify a length of all F's (for example, `0xFFFFFFFFFFFFFFFFULL`), to indicate that you want to create the largest partition possible.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_PARTITIONINFO   pInfo;

if( XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   printf( “Error: Open failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

//
// By default, the StreamStor device is not partitioned.  To
// determine if the device is partitioned, call XLRGetPartitionInfo.
//
if ( XLRGetPartitionInfo(xlrDevice, &pInfo) != XLR_SUCCESS )
{
   printf ( "Error:  Cannot get partitioning information: %u\n",
     XLRGetLastError() );
   exit(1);
}
if( pInfo.Partitioned == TRUE )
{
   printf ("This StreamStor device is partitioned.\n");
   // Since the device is already partitioned, we can create
   // new partitions now – we don’t have to erase pre-existing
   // data.
}
else
{
   printf ("This StreamStor device is NOT partitioned.\n");
   // If the device has any data (or a user directory) on it,
   // it must first be erased before we can begin partitioning the
   // device.
   if( XLRErase(xlrDevice, SS_OVERWRITE_PARTITION) != XLR_SUCCESS )
   {
      printf ( “Error:  Cannot erase:  %u\n”, XLRGetLastError() );
      exit(1);
   }
}

// Create a partition.  If this is the first partition to be created
// on the device, it will be assigned a partition number of 0.
// If it is not the first partition, XLRPartitionCreate will
// assign it the next available partition number.
if ( XLRPartitionCreate( xlrDevice, 0x10000000 ) != XLR_SUCCESS )
{
   printf( “Error: PartitionCreate failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}
```

See Also:

XLRPartitionSelect, XLRPartitionDelete and XLRGetPartitionInfo.

XLRPartitionDelete

Syntax:

`XLR_RETURN_CODE XLRPartitionDelete( SSHANDLE` *`xlrDevice,`* `UINT32` *`partition`* `)`

Description:

`XLRPartitionDelete` will delete the last partition on the StreamStor device.

To delete the last partition, you must first call `XLRSelectPartition` to select a partition other than the last partition.

If there is only one partition on the device, select partition 0 (zero) and then call `XLRPartitionDelete`. The effect of deleting the only partition on the device is the same as if `XLRErase` was called with the `SS_OVERWRITE_PARTITION` option.

IMPORTANT: `XLRPartitionDelete` will ignore write protection.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`partition` is the number of the last partition on the device.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_DIR             dir;
S_PARTITIONINFO   reported;
UINT32            partitionToDelete;

if( XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   printf( “Error: Open failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

if( XLRGetPartitionInfo(xlrDevice, &reported) != XLR_SUCCESS)
{
    printf( “Error: GetPartitionInfo failed: %u\n”,
      XLRGetLastError() );
    XLRClose( xlrDevice );
    exit( -1 );
}
```

```
if(reported.Partitioned == FALSE) {
    printf( “This device is not partitioned.\n”);
    XLRClose( xlrDevice );
    exit( 0 );
}

// Partition numbering starts at zero.  You can only delete the
// last partition.
partitionToDelete = reported.NumPartitions – 1;

// XLRPartionDelete ignores write protection, so explicitly check for
// write protection on the partition before trying to delete it.
if(XLRPartitionSelect( xlrDevice, partitionToDelete ) != XLR_SUCCESS )
{
   printf( “Error: PartitionSelect failed: %u\n”, XLRGetLastError() );
   XLRClose( xlrDevice );
   exit( -1 );
}
if(XLRGetDirectory( xlrDevice, &dir) != XLR_SUCCESS )
{
   printf( “Error: XLRGetDirectory failed: %u\n”, XLRGetLastError() );
   XLRClose( xlrDevice );
   exit( -1 );
}
if(dir.WriteProtected == TRUE) {
   printf( “Error:  This partition is write protected.\n”);
   XLRClose( xlrDevice );
   exit( -1 );
}

// You cannot delete the currently selected partition (unless there is
// only one partition on the device), so to make sure that some
// other partition is selected, we select partition 0.
if(XLRPartitionSelect( xlrDevice, 0 ) != XLR_SUCCESS )
{
   printf( “Error: PartitionSelect failed: %u\n”, XLRGetLastError() );
   XLRClose( xlrDevice );
   exit( -1 );
}

// Note how we specify the partition we want to delete in the
// XLRPartitionDelete command.
if( XLRPartitionDelete( xlrDevice, partitionToDelete ) != XLR_SUCCESS )
{
   printf( “Error: PartitionDelete failed: %u\n”, XLRGetLastError() );
   XLRClose( xlrDevice );
   exit( -1 );
}
```

See Also:

XLRPartitionCreate, XLRPartitionResize, XLRGetPartitionInfo, XLRPartitionSelect.

XLRPartitionResize

Syntax:

`XLR_RETURN_CODE XLRPartitionResize( SSHANDLE xlrDevice, UINT64 newSize )`

Description:

`XLRPartitionResize` will resize the last partition on the StreamStor device.

If the requested *`newSize`* would result in the loss of data, the resize will not be done and `XLRPartitonResize` will return `XLR_FAIL`.

To resize the partition, you must first select it with the `XLRSelectPartition` command.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`newsize` is the desired new length for the last partition.

If the partition is not empty:

- If *`newsize`* is non-zero, the partition will be resized as close as possible to *`newsize`*. *`newsize`* must be greater than the amount of recorded data.
- If *`newsize`* is zero, then the partition will be resized as close as possible to the size of the recorded data.

It the partition is empty:

- *`newsize`* must be greater than zero. The partition will be resized as close as possible to *`newsize`*.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE    xlrDevice;
UINT64      newSize=1048576;

if( XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   printf( “Error: Open failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

// Assume the last partition created was partition 10.  You must
// select it before attempting to resize it.

if(XLRPartitionSelect( xlrDevice, 10 ) != XLR_SUCCESS )
{
   printf( “Error: PartitionSelect failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

if( XLRPartitionResize( xlrDevice, newSize ) != XLR_SUCCESS )
{
   printf( “Error: PartitionResize failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}
```

See Also:

XLRPartitionCreate, XLRPartitionDelete, XLRGetPartitionInfo, XLRPartitionSelect.

XLRPartitionSelect

Syntax:

`XLR_RETURN_CODE XLRPartitionSelect( SSHANDLE` *`xlrDevice,`* `UINT32` *`partition`* `)`

Description:

`XLRPartitionSelect` will select an already existing partition. Partitions are numbered starting at 0. Thus, a 5 partition system will contain partitions: 0, 1, 2, 3, and 4.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`partition` is the partition number to select.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE xlrDevice;

if( XLROpen( 1, &xlrDevice ) != XLR_SUCCESS )
{
   printf( “Error: Open failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}

if( XLRPartitionSelect( xlrDevice, 4 ) != XLR_SUCCESS )
{
   printf( “Error: PartitionSelect failed: %u\n”, XLRGetLastError() );
   exit( -1 );
}
```

See Also:

`XLRPartitionCreate, XLRPartitionDelete` and `XLRGetPartitionInfo.`

XLRPlayback

Syntax:

`XLR_RETURN_CODE XLRPlayback( SSHANDLE` *`xlrDevice`*`, UINT32` *`Addrhigh`*`, UINT32` *`Addrlow`* `)`

Description:

`XLRPlayback` puts StreamStor into playback mode where data is made available for transfer to an outside device. The supplied address will be used to set the starting point of the data to be made available for transfer.

Playback continues until:

- `XLRStop` is called to halt the playback or
- all data is played back or
- a play limit (see `XLRSetPlaybackLength`) is reached.

This function can be used for streaming data out the external (FPDP or SFPDP) port or it can be used in conjunction with `XLRSetReadLimit` to allow a PCI device to source data from StreamStor.

The playback address must be an eight-byte aligned value.

If the StreamStor is in bank mode, this command will play back data from the currently selected bank.

If data was recorded on multiple channels:

- Before playing back data, you must first select the channel upon which the data was recorded and then bind it as the input channel.
- If you are playing back a single channel, you can specify a non-zero starting point with `XLRPlayback`. However, if you are attempting to playback more than one channel simultaneously, you cannot specify a non-zero starting point. Instead, you must set *`AddrHigh`* and *`AddrLow`* to zero.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`AddrHigh` is the upper 32-bit value of the 64-bit address to begin reading.

`AddrLow` is the lower 32-bit value of the 64-bit address to begin reading.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
// This example shows how you can stream data out the external
// (FPDP) port.

#include <stdio.h>
#include <stdlib.h>
#include "xlrapi.h"

int main( int argc, char *argv[] )
{
   SSHANDLE          xlrDevice;
   S_DEVINFO         devInfo;
   UINT64            bytesPlayed;
   UINT64            offset;
   UINT64            recordingLength;
   UINT32            AddrHi;
   UINT32            AddrLo;
   XLR_RETURN_CODE   xlrStatus;
   char              errMessage[XLR_ERROR_LENGTH];

   xlrStatus = XLROpen( 1, &xlrDevice );
   if( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error opening device: %s\n", errMessage );
      exit(1);
   }

   xlrStatus = XLRGetDeviceInfo( xlrDevice, &devInfo );
   if( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error getting device information: %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
   }

   // Use the external port for playback.
   xlrStatus = XLRSetMode( xlrDevice, SS_MODE_SINGLE_CHANNEL );
   if( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error setting the mode:  %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
    }

   xlrStatus = XLRBindInputChannel( xlrDevice, 0 );
   if( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error setting the mode:  %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
   }
```

```
   xlrStatus = XLRBindOutputChannel( xlrDevice, 0 );
   if( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error setting the mode:  %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
   }

   recordingLength = XLRGetLength( xlrDevice);
   if ( recordingLength == 0 )
   {
      printf ( "No data to playback.\n" );
      XLRClose( xlrDevice );
      exit(0);
   }
   // Set the FPDP mode.  SS_FPDP_XMIT does not drive the clock.
   xlrStatus = XLRSetDBMode( xlrDevice,SS_FPDP_XMIT,0 );
   if ( xlrStatus != XLR_SUCCESS )
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error setting FPDP mode: %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
    }

   // An offset of zero starts playback at the beginning.
   offset= 0;

   AddrHi = offset >> 32;
   AddrLo = offset & 0xFFFFFFFF;

   // Start the playback.
   xlrStatus = XLRPlayback(xlrDevice, AddrHi, AddrLo);
   if (xlrStatus != XLR_SUCCESS)
   {
      XLRGetErrorMessage( errMessage, XLRGetLastError() );
      printf( "Error with playback: %s\n", errMessage );
      XLRClose( xlrDevice );
      exit(1);
    }

   ... Sleep, allowing playback to continue...

   // Stop playback.
   XLRStop (xlrDevice);

   bytesPlayed = XLRGetPlayLength(xlrDevice);
   printf ("Number of bytes played = %llu\n", bytesPlayed);

   XLRClose(xlrDevice);
}
```

See Also:

XLRStop, XLRRead, XLRSetPlaybackLength, XLRGetPlayLength, XLRSetMode, XLRSetDBMode, XLRSetBankMode and XLRSelectBank.

XLRPlaybackLoop

Syntax:

```
XLR_RETURN_CODE XLRPlaybackLoop( SSHANDLE xlrDevice, UINT32
startHigh, UINT32 startLow, UINT32 lengthHigh, UINT32 lengthLow )
```

Description

`XLRPlaybackLoop` starts data playback and loops back to the start position when the end of data is reached or when the requested playback length is reached.

`startHigh` and *`startLow`* designate the start position, i.e., the requested offset into the device at which to begin playback.

`lengthHigh` and l*`engthLow`* designate the playback length, i.e., the requested number of bytes to play. If the playback length is zero, the StreamStor will play back all of the data and then "loop", playing back data beginning at the start position. Playback will continue looping until `XLRStop` is called. If the playback length is not zero, then looping will occur when the number of bytes played equals the requested playback length.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`startHigh` is the upper 32 bits of the 64 bit value that identifies the start position for the playback.

`startLow` is the lower 32 bites of the 64 bit value that identifies the start position for the playback.

`lengthHigh` is the upper 32 bits of the 64 bit value that identifies the playback length.

`lengthLow` is the lower 32 bits of the 64 bit value that identifies the playback length.

The playback length must be an eight byte-aligned value.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE           xlrDevice;
XLR_RETURN_CODE    xlrStatus;

XLROpen( 1, &xlrDevice );

 … Set up channel for playback …

//
// Start playing data beginning at offset 0.  Then, after all data has
// been played, automatically loop back to offset 0 and continue
// playback.

xlrStatus = XLRPlaybackLoop( xlrDevice, 0, 0 );
```

See Also:

XLRPlayback and XLRSetPlaybackLength.

XLRPlayTrigger

Syntax:

```
XLR_RETURN_CODE XLRPlayTrigger( SSHANDLE xlrDevice )
```

Description

`XLRPlayTrigger` starts read data flowing.

When `XLRPlayback` is called, preparations are made internally on the StreamStor to begin playback. When preparations are complete, the data begins to flow. So, there is a delay between the call to `XLRPlayback` and the flow of data. To minimize the delay, you can call `XLRSetOption` to set the `SS_OPT_PLAYARM` option. Setting this option causes the preparations for playback to be made, but playback does not start until `XLRPlayTrigger` is called.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
// Example showing how to arm the StreamStor for playback.

SSHANDLE           xlrDevice;
XLR_RETURN_CODE    xlrStatus;

XLROpen( 1, &xlrDevice );

// Set up channel for playback …

// Arm the playback trigger.
xlrStatus = XLRSetOption( xlrDevice, SS_OPT_PLAYARM );

//
// Since SS_OPT_PLAYARM has been set, this call to XLRPlayback
// will not cause the data to flow yet.
//
xlrStatus = XLRPlayback( xlrDevice, 0, 0 );

//
// If we do not sleep here, then the playback will work as
// it does ordinarily.  If instead we sleep, the internal
// playback buffer will fill up.  This should take no
// longer than three seconds.
//
 ... sleep, allowing playback buffer to fill ...
```

```
//
// The playback buffer should now be full.  Start the data flowing
// out of the buffer.
//
xlrStatus = XLRPlayTrigger( xlrDevice );

 ... playback the desired length of time . . .

// Stop the playback.
XLRStop( xlrDevice );
```

See Also:

XLRSetOption, XLRGetPlayBufferStatus, and XLRPlayback.

XLRRead

Syntax:

`XLR_RETURN_CODE XLRRead(SSHANDLE` *`xlrDevice`*`, PS_READDESC` *`pReadDesc`*`)`

Description:

`XLRRead` reads data from the StreamStor device.

The address of the requested data must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will read data from the currently selected bank.

If data was recorded on multiple channels, you must first select the channel upon which the data was recorded and then bind that channel to the output channel, which in the case of `XLRRead` should always be the PCI bus (channel 0). The StreamStor SDK has multi-channel examples in the `example` directory which demonstrate channel binding.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pReadDesc` is a pointer to an `S_READDESC` structure that holds the read address, length and buffer address for the read data.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_READDESC        readDesc;
UINT32            myBuffer[40000];
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
//AddrHi and AddrLo must represent an appropriately aligned address.
readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLRRead( xlrDevice, &readDesc );
```

See Also:

XLRPlayback, XLRSetMode, XLRSetDBMode, XLRSetBankMode and XLRSelectBank.

XLRReadData

Syntax:

XLR_RETURN_CODE XLRReadData(SSHANDLE *xlrDevice*, PUINT32 *Buffer*, UINT32 *AddrHigh*, UINT32 *AddrLow*, UINT32 *XferLength*)

Description:

XLRReadData reads data from the StreamStor device. This function is identical to XLRRead without the structure to pass request parameters.

The address of the requested data must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will read data from the currently selected bank.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

Buffer is the address of the user memory buffer to hold the requested data.

AddrHigh is the upper 32 bits of a 64-bit byte address of the requested data.

AddrLow is the lower 32 bits of a 64-bit byte address of the requested data.

XferLength is the number of bytes requested.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;
UINT32            myBuffer[40000];

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
//AddrHigh and AddrLow must represent an appropriately aligned address.
xlrReturnCode = XLRReadData( xlrDevice, myBuffer, 0, 0xFE120000,
sizeof(myBuffer) );
```

See Also:

XLRRead, XLRSetMode, XLRSetDBMode, XLRSetBankMode and XLRSelectBank.

XLRReadFifo

Syntax:

```
XLR_RETURN_CODE XLRReadFifo( SSHANDLE xlrDevice, PUINT32 Buffer,
UINT32 Length, BOOLEAN Direct )
```

Description:

XLRReadFifo reads data from the StreamStor device during a FIFO operation. Data can continue to be read with this function until the FIFO is empty or XLRStop is called. Note that the device must be in record mode when XLRReadFifo is called. A second call to XLRStop is required to take the StreamStor out of record mode.

For general information on FIFOs, please refer to the Forking and Passthru chapter of this manual. XLRGetFIFOLengthExample.c (which is in the StreamStor SDK example directory) shows how to use various FIFO commands.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

Buffer is the address of the buffer to receive the read data.

Length is the length of data to transfer in bytes.

Direct is a flag that indicates if the supplied *Buffer* address is a physical address for direct transfer. For normal transfer to a user memory buffer this flag should be FALSE (0).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;
UINT32            myBuffer[40000];

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRReadFifo(xlrDevice, myBuffer, sizeof(myBuffer),
FALSE);
```

See Also:

XLRGetFifoLength, XLRRecord, XLRSetDBMode, XLRSetBankMode and XLRSelectBank.

XLRReadImmed

Syntax:

`XLR_RETURN_CODE XLRReadImmed( SSHANDLE` *`xlrDevice`*`, PS_READDESC` *`pReadDesc`* `)`

Description:

`XLRReadImmed` reads data from the StreamStor device without waiting for completion. You must receive `XLR_READ_COMPLETE` status from `XLRReadStatus` before any other commands can be issued. Note that only a single outstanding request is allowed per execution thread.

The address of the requested data must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will read data from the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`pReadDesc` is a pointer to an `S_READDESC` structure that holds the read address, length and buffer address for the read data.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE         xlrDevice;
S_READDESC       readDesc;
UINT32           myBuffer[40000];
XLR_READ_STATUS  readStatus;
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …

//AddrHi and AddrLo must represent an appropriately aligned address.
readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLRReadImmed( xlrDevice, &readDesc );

/* DO SOME NON-STREAMSTOR RELATED WORK HERE */

readStatus = XLRReadStatus( TRUE );
if( readStatus != XLR_READ_COMPLETE )
{
   /* PROCESS ERROR! */
}
```

See Also:

XLRReadStatus, XLRSetBankMode and XLRSelectBank.

XLRReadSmartThresholds

Syntax:

```
XLR_RETURN_CODE XLRReadSmartThresholds( SSHANDLE xlrDevice,
S_SMARTTHRESHOLDS values[], UINT32 Bus, UINT32 MasterSlave )
```

Description:

XLRReadSmartThresholds returns the SMART threshold values that are provided by the disk drive vendor's self monitoring facility. To interpret the values returned, please refer to the ATA specifications or to the disk drive vendor's documentation.

If the StreamStor is in bank mode, this command will return the values for the specified drives on the currently selected bank.

This function is only supported on disk drives that support self monitoring. You can call XLRGetDriveInfo then examine the returned SMARTCapable value to determine this.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

values is a pointer to an array of S_SMARTTHRESHOLDS structures. The array must be large enough to hold at least XLR_MAX_SMARTVALUES structures.

Bus is the ATA bus number of the drive.

MasterSlave is XLR_MASTER_DRIVE (0) or XLR_SLAVE_DRIVE (1) to select the master or slave drive on the ATA bus.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
S_SMARTTHRESHOLDS thresholds[XLR_MAX_SMARTVALUES];

xlrStatus = XLROpen( 1, &xlrDevice );
      …
if( XLRReadSmartThresholds( hTarget, thresholds, 3, XLR_MASTER_DRIVE !=
XLR_SUCCESS )
{
   printf("Threshold information not available for Bus 3 Master
drive.\n");
}
else
{
      … Interpret the thresholds …
}
```

See Also:

XLRReadSmartValues, XLRGetDriveInfo, XLRSetBankMode and XLRSelectBank.

XLRReadSmartValues

Syntax:

```
XLR_RETURN_CODE XLRReadSmartValues( SSHANDLE xlrDevice,
PUSHORT version, S_SMARTTHRESHOLDS values[], UINT32 Bus, UINT32
MasterSlave )
```

Description:

`XLRReadSmartValues` returns the SMART values that are provided by the disk drive vendor's self monitoring facility. To interpret the values returned, please refer to the ATA specifications or to the disk drive vendor's documentation.

If the StreamStor is in bank mode, this command will return the values for the specified drives on the currently selected bank.

This function is only supported on disk drives that support self monitoring. You can call `XLRGetDriveInfo` then examine the returned `SMARTCapable` value to determine this.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`version` is the SMART attributes data structure revision number, as reported by the disk drive vendor.

`values` is a pointer to an array of S_SMARTVALUES structures. The array must be large enough to hold XLR_MAX_SMARTVALUES structures.

`Bus` is the ATA bus number of the drive.

`MasterSlave` is `XLR_MASTER_DRIVE` (0) or `XLR_SLAVE_DRIVE` (1) to select the master or slave drive on the ATA bus.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
USHORT            version;
XLR_RETURN_CODE   xlrStatus;
S_SMARTVALUES     smartVals[XLR_MAX_SMARTVALUES];

xlrStatus = XLROpen( 1, &xlrDevice );
      …
if( XLRReadSmartValues( hTarget, &version, smartVals,
    1, XLR_MASTER_DRIVE !=   XLR_SUCCESS )
{
   printf("Smart values not available for Bus 1 Master drive.\n");
}
else
{
      … Interpret the SMART values …
}
```

See Also:

XLRReadSmartThresholds, XLRGetDriveInfo, XLRSetBankMode and XLRSelectBank.

XLRReadStatus

Syntax:

`XLR_RETURN_CODE XLRReadStatus( BOOLEAN *Wait* )`

Description:

`XLRReadStatus` checks status of a read request issued with `XLRReadImmed` data from the StreamStor device.

If the StreamStor is in bank mode, this command will check the status of the currently selected bank.

Parameters:

`Wait` is a flag to indicate whether to wait for completion of the read request. If `TRUE`, the function will not return until the read is complete or an error has occurred.

Return Value:

If the read request has completed: `XLR_READ_COMPLETE`
If the read request is waiting to execute: `XLR_READ_WAITING`
If the read request is currently executing: `XLR_READ_RUNNING`
If an error occurred during execution of the request: `XLR_READ_ERROR`

Usage:

```
SSHANDLE         xlrDevice;
S_READDESC       readDesc;
UINT32           myBuffer[40000];
XLR_READ_STATUS  readStatus;
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLRReadImmed( xlrDevice, &readDesc );

while( moreWork )
{
   /* DO OTHER WORK HERE */
   readReturnCode = XLRReadStatus( FALSE );
   if( readStatus == XLR_READ_ERROR )
   {
      /* PROCESS ERROR! */
   }
   else if( readStatus == XLR_READ_COMPLETE )
      break;
}
```

See Also:

`XLRReadImmed`, `XLRSetBankMode` and `XLRSelectBank`.

XLRReadToPhy

Syntax:

XLR_RETURN_CODE XLRReadToPhy(SSHANDLE *xlrDevice*, PS_READDESC *pReadDesc*)

Description:

XLRReadToPhy reads data from the StreamStor device and writes directly to a supplied physical hardware address. This function is intended only for moving data between StreamStor and another device on the bus. The buffer address supplied MUST be a physical address and the entire transfer size must be available. The supplied address and length will be used to directly program the StreamStor DMA to transfer the data. Specifying incorrect addresses to this function can cause system crashes and instability.

The address of the requested data must be an eight byte-aligned value.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pReadDesc is a pointer to an S_READDESC structure that holds the read address, length and physical address for the read data.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE         xlrDevice;
S_READDESC       readDesc;
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
//AddrHi and AddrLo must represent an appropriately aligned address.
readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = XLRGetBaseRange( xlrDevice );
readDesc.BufferAddr = myDeviceAddress;

xlrReturnCode = XLRReadToPhy( xlrDevice, &readDesc );
```

See Also:

XLRPlayback.

XLRRecord

Syntax:

```
XLR_RETURN_CODE XLRRecord( SSHANDLE xlrDevice, BOOLEAN
WrapEnable, SHORT ZoneRange )
```

Description:

`XLRRecord` starts the record mode of the StreamStor device. After a successful call of this function, the StreamStor device will record to disk any data written to its data window on PCI or to its external data port. Recording will continue until the device is full or until `XLRStop` is called (whichever occurs first.)

Note:

- If the StreamStor is in bank mode, this command will record on the currently selected bank.
- If the StreamStor is partitioned, this command will record on the currently selected partition.
- If the StreamStor is in multi-channel mode, this command will record on all channels that you have bound for input (with `XLRBindInputChannel`).

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`WrapEnable` should be set to 1 to allow StreamStor to operate as a circular buffer. The oldest data will be overwritten if more data is received than is available on the disk drives. To force StreamStor to stop accepting data at the disk storage limits, set this parameter to 0.

`ZoneRange` is not currently supported and should be set to 1.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE         xlrDevice;
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
// Start recording data but ensure that no captured data is
// overwritten.
xlrReturnCode = XLRRecord( xlrDevice, 0, 1 );

/* System is now recording . . . */

// End the recording.
XLRStop(xlrDevice);
```

See Also:

XLRAppend, XLRWrite, XLRStop, XLRSetBankMode, XLRSelectBank, XLRPartitonSelect, XLRSetMode and XLBBindInputChannel.

XLRRecoverData

Syntax:

```
XLR_RETURN_CODE XLRRecoverData( SSHANDLE xlrDevice, IN UINT32
Mode )
```

Description:

`XLRRecoverData` attempts to recover data.

If partitioning was used on the system, prior to calling `XLRRecoverData`, you should call `XLRPartitionSelect` to select the partition that was in use prior to the failure (or prior to the overwrite).

`XLRRecoverData` does not recover the user directory or label - it will only attempt to recover the main data area. The last partial block of data may be truncated upon recovery, so you may lose up to 64K bytes of data.

Note that in some cases, no recovery or only partial recovery of data is possible. It is the user's responsibility to verify the integrity of any recovered data and, if necessary, truncate any corrupted data from the recording.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`Mode` is the type of recovery that is to be performed.

- If a recording has ended without calling `XLRStop` (as might happen if the StreamStor's power fails), StreamStor's directory may be corrupted. To recover data in this case, set `Mode` to `SS_RECOVER_POWERFAIL`.
- If a recording has been partially overwritten, the data that has not been overwritten may be recoverable. To recover data in this case, set `Mode` to `SS_RECOVER_OVERWRITE`.
- If a recording has been accidentally erased, but not overwritten, the data may still be recoverable. To recover data in this case, set `Mode` to `SS_RECOVER_UNERASE`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE        xlrDevice;
XLR_RETURN_CODE xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …

// Attempt to repair StreamStor directory and recover data.
xlrReturnCode = XLRRecoverData( xlrDevice, SS_RECOVER_POWERFAIL );
```

See Also:

XLRPartitionSelect.

XLRReset

Syntax:

`XLR_RETURN_CODE XLRReset( SSHANDLE` *`xlrDevice`* `)`

Description:

`XLRReset` will attempt to reset a StreamStor device and re-initialize the hardware and firmware. This function should be used only as a last resort.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRReset( xlrDevice );
```

See Also:

`XLRCardReset.`

XLRRetrieveEvents

Syntax:

`XLR_RETURN_CODE XLRRetrieveEvents( SSHANDLE` *`xlrDevice,`* `UINT64` *`EventLog[]`* `)`

Description:

`XLRRetrieveEvents` downloads an array of 64 bit integers into the `EventLog` array. Each integer is an offset into the current recording. An "event" is recorded for each FPDP *SYNC pulse if the `SS_OPT_FPDPEVENTLOG` option is set in `XLRSetDBMode`. A maximum of `MAX_EVENTS` events can be captured. Positions in the `EventLog` array that do not contain an event are set to 0.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`EventLog[]` is an array of 64 bit integers.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE                xlrDevice;
XLR_RETURN_CODE         xlrReturnCode;
UINT64                  Events[MAX_EVENTS];
UINT32                  i=0;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRRetrieveEvents( xlrDevice, Events );
if( xlrReturnCode != XLR_SUCCESS )
{
  printf("ERROR\n");
  return 1;
}

for( i = 0; i < MAX_EVENTS; i++ )
{
  if( Events[i] == 0 )
  {
    // No more events recorded.
    break;
  }
  printf("Event %d: %ull\n", i, Events[i]);
}
```

See Also:

`XLRGetEvents, XLRGetEventsLength and XLRSetDBMode.`

XLRSdkVersion

Syntax:

```
void XLRSdkVersion( char *versionstring )
```

Description:

`XLRSdkVersion` returns the SDK version as a string formatted as a *major.minor* version number. Note that the SDK is a collection of the various components that make up the software, hardware and firmware of the StreamStor system. The SDK version may not reflect independent updates of these components.

Parameters:

`versionstring` is a pointer to a character string to hold the returned version. It must be of minimum length `XLR_VERSION_LENGTH.`

Return Value:

The SDK version is returned in *`versionstring`*.

Usage:

```
/* Read XLR API version into string */
char xlrstring[XLR_VERSION_LENGTH];

XLRSdkVersion( xlrstring );
printf( “StreamStor SDK version is %s”, xlrstring );
```

See Also:

`XLRGetVersion and XLRApiVersion.`

XLRSelectBank

Syntax:

```
XLR_RETURN_CODE XLRSelectBank( SSHANDLE xlrDevice, UINT32 bankID
)
```

Description:

XLRSelectBank will select the specified bank. Subsequent calls to bank aware commands will then perform their operations on the selected bank. For example, if BANK_B is the currently selected bank, then a subsequent call to XLRSetLabel will label BANK_B.

Once a bank is selected, that bank remains selected until the other bank is explicitly selected or the StreamStor mode is changed. For example, say BANK_B was selected, a record was performed, and then XLRClose was called. Upon reopening the StreamStor with XLROpen, unless XLRSetBankMode was called to take the system out of bank mode or the select bank was changed with XLRSelectBank, the system would still be in bank mode and BANK_B would be the selected bank.

The StreamStor must be in bank mode to select a bank. When calling XLRSelectBank, the drive module to be selected must be mounted in the selected bank and be in the "ready" state.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

bankID is the bank to be selected (BANK_A or BANK_B).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRSetBankMode ( xlrDevice, SS_BANKMODE_NORMAL );

// Clear the write protection on Bank A.
xlrStatus = XLRSelectBank ( xlrDevice, BANK_A );
xlrStatus = XLRClearWriteProtect( xlrDevice );
// Set the write protection on Bank B.
xlrStatus = XLRSelectBank ( xlrDevice, BANK_B );
xlrStatus = XLRSetWriteProtect( xlrDevice );
```

See Also:

XLRSetBankMode and XLRGetBankStatus.

XLRSelectChannel

Syntax:

`XLR_RETURN_CODE XLRSelectChannel(SSHANDLE *xlrDevice*, UINT32 *channel*)`

Description:

`XLRSelectChannel` selects the channel that future commands will operate on. A channel can be selected and operated on regardless of whether or not it's bound.

Please refer to the hardware manual for your StreamStor controller or daughterboard for the list of channels it supports.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`channel` is the number of the channel to select.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );
      …

xlrStatus = XLRSelectChannel( xlrDevice, 30 );
      …

xlrStatus = XLRSetDBMode( xlrDevice, SS_FPDP_RECVMASTER, 0 );
      …
```

See Also:

`XLRClearChannels, XLRBindInputChannel, and XLRSetDBMode.`

XLRSelfTest

Syntax:

```
XLR_RETURN_CODE XLRSelfTest( SSHANDLE xlrDevice,
   SS_SELFTEST test )
```

Description:

`XLRSelfTest` performs an internal self-test on the StreamStor device. **After self testing has completed, you should reset the StreamStor card.**

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`test` is a constant specifying what kind of test you wish to perform. Possible values are:

- `XLR_BIST_PCI` – test communications with PCI bus.
- `XLR_BIST_BUFFER` – write and then read all 512 MB of RAM – checks for bit errors.
- `XLR_BIST_DISK`*x* – *x* represents the bus number (0-7). This will test all disks present on the specified bus (i.e. master and slave if present).
- `XLR_BIST_ALL` – performs complete self-test (PCI, buffer, and all disks present on the system).

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL. Call `XLRGetLastError` and `XLRGetErrorMessage` (as demonstrated below) to find which component(s) failed the test.

Diagnostic Error Messages:

Action:	Error Message:
Test of drive that isn't present	Invalid command
Drive fails self test	Drive missing or failing
Test of non XF/XF2 StreamStor	Invalid request for system mode
Buffer test failed	Device command failed execution
PCI test failed	Device command failed execution

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;
Char              temp[XLR_ERROR_LENGTH];

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRSelfTest( xlrDevice, XLR_BIST_DISK5 );
if( xlrReturnCode != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( temp, xlrError );
   printf( “%s\n”, temp );
   exit(1);
}
// Reset the card.
XLRCardReset(1);
```

XLRSetBankMode

Syntax:

`XLR_RETURN_CODE XLRSetBankMode(SSHANDLE` *`xlrDevice`*`, S_BANKMODE` *`mode`*`)`

Description:

`XLRSetBankMode` sets the banking mode for the StreamStor. By default, the StreamStor device is not in bank mode, i.e., a call to `XLROpen` will set the bank mode to `SS_BANKMODE_DISABLED`. (Exceptions to this are the StreamStor PCI-816V100 and PCI-816VXF2 boards, which `XLROpen` automatically places in bank mode).

The StreamStor remains in the selected mode until a call to `XLRSetBankMode` is made to change the mode.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`mode` is the constant that defines the banking mode for the StreamStor.

- `SS_BANKMODE_NORMAL` – puts the StreamStor into normal bank mode. If `BANK_A` is ready, it is selected as the current bank. If `BANK_A` is not ready, then if `BANK_B` is ready, it is selected as the current bank.
- `SS_BANKMODE_DISABLED` – disables bank mode.
- `SS_BANKMODE_AUTO_ON_FULL` – automatically switches between banks when one fills up during a recording.

Autoswitch Notes:
Recording:
When using `SS_BANKMODE_AUTO_ON_FULL`, StreamStor will switch to the other bank when the current bank becomes full. This transition requires that the bank being switched to is ready, and has not been previously recorded on. For example, if recording begins on Bank A, fills Bank A, and switches to Bank B, StreamStor will not switch back to Bank A unless Bank A has been replaced. If Bank A has been replaced with a new bank, StreamStor will switch back to Bank A when Bank B is full. If Bank A has not been replaced, the recording will end because there is no free space left. Unless the write protect option is set, each call to `XLRRecord` will prepare each bank present in the system for recording. If the banks have been recorded on before, they will be overwritten. In other words, StreamStor losses its ability to track which banks have been "used" (i.e. written) between calls to `XLRRecord`. If data written to the bank should not be overwritten, the write protect option should be set on that bank.

Playback:
Auto bank switching is **not** available in playback mode. Any recordings made with auto bank switching will be considered separate recordings for playback. In other words, from a playback perspective (offset and length), each bank begins at an offset of 0. For example, if both banks have been recorded on (such that an autoswitch occurred to Bank B when Bank A filled up), each bank's recording will start at an offset of 0. To start playback at the beginning of Bank A, select Bank A with a call to `XLRSelectBank`, and then begin playback at an offset of 0. To start playback at the beginning of Bank B, select Bank B with a call to `XLRSelectBank`, and then begin playback at an offset of 0.

Note that if you are in bank mode and are using partitions, the bank mode `SS_BANK_AUTO_ON_FULL` is not allowed. The only way to begin operations on a different bank is to explicitly select the bank to be used.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

xlrStatus = XLRSetBankMode (xlrDevice, SS_BANKMODE_NORMAL);

//The default bank is Bank A and it is ready, so this call will clear
//write protection on BANK_A.
xlrStatus = XLRClearWriteProtect( xlrDevice );

//Turn bank mode off.
xlrStatus = XLRSetBankMode (xlrDevice, SS_BANKMODE_DISABLED);
```

See Also:

`XLRGetChassisType` and `XLRSelectBank`.

XLRSetDBMode

Syntax:

```
XLR_RETURN_CODE XLRSetDBMode( SSHANDLE xlrDevice, FPDPMODE Mode,
FPDPOP option )
```

Description:

`XLRSetDBMode` is used to set the operating mode of the external port on the Amazon daughterboard (if one exists) or on the FPDP interface for other StreamStor board types. For details on FPDP, please refer to the "External Port" chapter of this manual.

Please refer to the hardware manual for your StreamStor controller or daughterboard for the list of modes it supports.

IMPORTANT: When setting the operating mode for FPDP/FPDP-II connections on a given bus, do not configure more than one connector as a transmitter (FPDP/TM or FPDP/T) on that bus at a time. Otherwise, bus drivers may be permanently damaged.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`mode` is a constant that defines the mode of operation.

`option` is used to modify the operation of the FPDP port.

Please refer to the hardware manual for your StreamStor controller or daughterboard for the list of modes and options it supports.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

//
// This example shows how to set various modes and options.  Note that
// the manifest constants used differ based on the type of device.
// For example, FPDP/R is SS_FPDPMODE_RECVM for an Amazon daughter
// and is SS_FPDP_RECVMASTER for a PCI-816XF2.
//
xlrStatus = XLROpen( 1, &xlrDevice );
…
//
// Select channel 30.  Subsequent XLRSetDBMode calls will set
// the options and the modes for the selected channel.
//
xlrStatus = XLRSelectChannel( xlrDevice, 30 );

// Example 1:  Set the FPDP port mode to FPDP/R and use the default
// options on a PCI-816XF2.
xlrStatus = XLRSetDBMode( xlrDevice, SS_FPDP_RECV, 0 );

// Example 2: Enable the data strobe clock and "Not Ready"
// assert options on a PCI-816XF2 and use FPDP/RM.
xlrReturnCode = XLRSetDBMode( xlrDevice,
   SS_FPDP_RECVMASTER, SS_OPT_FPDPSTROB|SS_OPT_NRASSERT );

// Example 3: Enable data strobe clock and use FPDP/R on a
// PCI-816XF2.
XLRSetDBMode( xlrDevice, SS_FPDP_RECV, SS_OPT_FPDPSTROB );

// Example 4: Enable FPDP/RM mode on a PCI-816XF2, using the
// default options.
xlrStatus = XLRSetDBMode( xlrDevice, SS_FPDP_RECVMASTER, 0 );

// Example 5:  Enable data strobe clock on an Amazon daughterboard
// using FPDP/RM.
xlrStatus = ( xlrDevice, SS_FPDPMODE_RECVM, SS_DBOPT_FPDPSTROB );

// Example 6: Enable "Not Ready" assert options on an Amazon
// daughterboard and use FPDP/RM.
xlrReturnCode = XLRSetDBMode( xlrDevice,
   SS_FPDPMODE_RECVM, SS_DBOPT_FPDPNRASSERT );
```

See Also:

`XLRSetMode` and `XLRSelectChannel`.

XLRSetDriveStandbyMode

Syntax:

```
XLR_RETURN_CODE XLRSetDriveStandbyMode( SSHANDLE xlrDevice,
BOOLEAN StandbyEnable)
```

Description:

`XLRSetDriveStandbyMode` is used to put the drives into power standby mode and to take the drives out of power standby mode.

Standby mode is a power management feature that spins-down disk drives. The spindle motor is stopped, and most of the electronics are powered off. Placing drives in standby mode when they are not in use will reduce the power consumption of the drives. This mode also reduces the chance of head-to-disk contact, which greatly decreases the probability of disk damage.

When drives are placed in standby mode, the recovery time when exiting standby mode depends on the disk drive model and other factors.

This command only affects drives that support standby mode, such as 2.5" notebook drives.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`StandbyEnable` sets the mode. If *`StandbyEnable`* is `TRUE`, the drives are put into standby mode. Otherwise, the drives are taken out of standby mode. By default, the drives are not in standby mode.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );

//
// If the enable parameter is TRUE, standby mode will be enabled,
// which means drives should spin down.
//
xlrStatus = XLRSetDriveStandbyMode(xlrDevice, TRUE);

// Do whatever you need to do while the drives are spun down.
            . . .

//
```

```
// Now you want to access the drives (for a record or playback,
// for example).  Spin the drives up.
//
xlrStatus = XLRSetDriveStandbyMode(xlrDevice, FALSE);

// Sleep 5 seconds, giving the drives time to spin up.
Sleep(5);

xlrStatus = XLRRecord (xlrDevice, 0, 1);
            . . .

// Close device before exiting
XLRClose( xlrDevice );
```

XLRSetLabel

Syntax:

```
XLR_RETURN_CODE XLRSetLabel( SSHANDLE xlrDevice, char *label,
UINT32 labelSize )
```

Description:

`XLRSetLabel` sets the label on the StreamStor recorder. The drives must be idle (i.e., not in record or playback mode) to set a label.

Note that if you call `XLRErase` to erase a recorder, whether or not the label is erased depends on the erase option you specify. Also, note that the length returned by `XLRGetLength` and by `XLRGetDirectory` does not include the length of the label.

If the StreamStor is in bank mode, this command will set the label for the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`label` is a pointer to a null terminated string no more than `XLR_LABEL_LENGTH` in size (including the `NULL`).

`labelSize` is the length of label, not including the `NULL` terminator. If *`labelSize`* is 0 (zero), the label on the device will be null'ed out.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
char              label[XLR_LABEL_LENGTH];

xlrStatus = XLROpen( 1, &xlrDevice );

strcpy (label, “Volume 1, 19-April-2004”);
xlrStatus = XLRSetLabel( xlrDevice, “Volume 1, label, strlen(label));
      …
xlrStatus = XLRGetLabel( xlrDevice, label );
printf ( “This disk set is labeled %s\n”, label );
      …
// Null out the label.
xlrStatus = XLRSetLabel( xlrDevice, “”, 0 );
      …
// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

`XLRGetLabel`, `XLRErase`, `XLRSetBankMode` and `XLRSelectBank`.

XLRSetMode

Syntax:

`XLR_RETURN_CODE XLRSetMode( SSHANDLE xlrDevice, SSMODE Mode )`

Description:

`XLRSetMode` is used to set the input/output path and functionality of StreamStor.

Please refer to the hardware manual for your StreamStor controller or daughterboard for the list of modes it supports.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`Mode` is a constant that defines the mode of StreamStor operation. Possible mode values are:

- `SS_MODE_SINGLE_CHANNEL` - This is the default mode that receives and sends data over one channel – i.e. the PCI bus, or one of the external ports.
- `SS_MODE_FORK` - This mode allows data to be recorded and simultaneously output. For example, data can be coming in over one of the external ports, recorded, and sent out the other external port.
- `SS_MODE_PASSTHRU` - This mode is identical to `SS_MODE_FORK` except that the data is not recorded; data merely passes in one port and out the other.
- `SS_MODE_MULTI_CHANNEL` - This mode allows StreamStor to record data from multiple input channels simultaneously. For example, data can be coming in from the external port and the PCI bus at the same time.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Note: **CHANGING MODES CLEARS ALL INPUT AND OUTPUT CHANNELS. CHANNELS MUST BE BOUND AFTER THE MODE IS SELECTED.**

Usage:

```
SSHANDLE        xlrDevice;
XLR_RETURN_CODE xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
    …
// Set StreamStor to use the external port
xlrReturnCode = XLRSetMode( xlrDevice, SS_MODE_SINGLE_CHANNEL );
```

See Also:

`XLRSetDBMode`, `XLRBindInputChannel`, `XLRBindOutputChannel`

XLRSetOption

Syntax:

```
XLR_RETURN_CODE XLRSetOption( SSHANDLE xlrDevice,
   UINT32 options_to_set )
```

Description

`XLRSetOption` sets one or more options. To set an option, the drives must be idle (i.e., not in record or playback mode).

Please refer to the hardware manual for your StreamStor controller or daughterboard for the list of options it supports.

The options are:

- `SS_OPT_FSMAPPED` – When set, the StreamStor is enabled to read while recording. Note that you cannot read while recording if you are recording in wrap mode. By default, this option is not set.
- `SS_OPT_PLAYARM` - When set, the StreamStor is armed for a two-stage playback. You set this option and then call `XLRPlayback`. Data will be buffered up for playback, but no data will play until triggered. (See `XLRPlayTrigger.)` By default, this option is not set.

When `XLROpen` is called, all options are set to their default value.

Parameters:

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen.`
- *`options_to_set`* is a vector containing one or more of the above options`.`

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE             xlrDevice;
XLR_RETURN_CODE      xlrStatus;

xlrStatus = XLROpen( 1, &xlrDevice );

// Set the desired option.
xlrStatus = XLRSetOption( xlrDevice, SS_OPT_PLAYARM );
    . . . record some data . . .
XLRStop( xlrDevice);

// Clear the option.
xlrStatus = XLRClearOption( xlrDevice, SS_OPT_PLAYARM );
```

See Also:

`XLRClearOption`, `XLRGetOption`, and `XLRPlayTrigger`.

XLRSetPlaybackLength

Syntax:

`XLR_RETURN_CODE XLRSetPlaybackLength( SSHANDLE` *`xlrDevice,`* `UINT32` *`lenHigh`*`, UINT32` *`lenLow`* `)`

Description:

`XLRSetPlaybackLength` is used to stop playback after a specified number of bytes have been played. When a playback length is set, playback continues until the number of specified bytes has been played, `XLRStop` is called or all data has been played. The playback length remains in effect until the next `XLROpen` or until `XLRSetPlaybackLength` is called again.

The playback length is used by the `XLRPlayback`, `XLRRead` and `XLRReadToPhy` commands. Playback length is initially set to zero by `XLROpen`. A playback length of zero causes playback or reading to continue until all data is played or read. Note that each time you call `XLRPlayback`, `XLRRead` or `XLRReadToPhy`, the count of the number of bytes that have already played returns to zero.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`lenHigh` is the upper 32 bits of the 64 bit value that identifies the playback length.

`lenLow` is the lower 32 bits of the 64 bit value that identifies the playback length.

The playback length must be an eight byte-aligned value.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE        xlrDevice;
UINT64          bytesToPlay;
UINT32          lenHigh;
UINT32          lenLow;
S_DEVSTATUS     devStatus;
XLR_RETURN_CODE xlrStatus;

bytesToPlay = 1048576;

xlrStatus = XLROpen( 1, &xlrDevice );
      …

//lenHigh and lenLow must represent an 8 byte aligned address.
lenHigh = bytesToPlay >> 32;
lenLow = bytesToPlay & 0xFFFFFFFF;
xlrReturnCode = XLRSetPlaybackLength( xlrDevice, lenHigh, lenLow );

// Start playback (at the beginning of the recording in this example).
// Poll every 10 seconds to see if playback has stopped.
// Note: if you use polling to check status, the more often you poll,
// the greater the impact on performance.

XlrReturnCode = XLRPlayback( xlrDevice, 0,0 );

do {
      … sleep for 10 seconds, then poll device status …
      XlrReturnCode = XLRGetDeviceStatus ( xlrDevice, &devStatus );
} while (devStatus.Playing == TRUE);
```

See Also:

XLRPlayback, XLRGetPlayLength, XLRSetMode, XLRSetDBMode, XLRSetBankMode and XLRSelectBank.

XLRSetPortClock

Syntax:

`XLR_RETURN_CODE XLRSetPortClock( SSHANDLE` *`xlrDevice,`* `UINT32` *`clock`* `)`

Description:

`XLRSetPortClock` is used to set the operating frequency of the external port if applicable. Before calling `XLRSetPortClock`, you must call `XLRSelectChannel` to select the desired channel.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`clock` is a constant that defines the desired clock frequency. Possible values are defined in the header file `xlrdbcommon.h` as `SS_PORTCLOCK_xMHz` values. The FPDP clock is programmable from 6 MHz up to 50 MHz.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
// Set the external clock frequency
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …

xlrReturnCode = XLRSelectChannel( xlrDevice, 30 );
xlrReturnCode = XLRSetPortClock( xlrDevice, SS_PORTCLOCK_40MHZ );
```

XLRSetReadLimit

Syntax:

`XLR_RETURN_CODE XLRSetReadLimit( SSHANDLE` *`xlrDevice,`* `UINT32` *`Limit`*
`)`

Description:

`XLRSetReadLimit` sets the size of the address range an outside device will be using when reading data from StreamStor during playback (`XLRPlayback`). This is required to prevent StreamStor hardware from discarding cached read data when an external DMA engine recycles to a new starting read address on the PCI bus.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`Limit` is the address range size that the outside device will use when reading from StreamStor during playback operations.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
UINT32            DMA_size = 0x2000;
PUINT32           pBuffer;
PUINT32           pSSAddr;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
     …
// Put StreamStor into Playback mode at beginning of recording
xlrReturnCode = XLRSetReadLimit( xlrDevice, DMA_size );
xlrReturnCode = XLRPlayback( xlrDevice, 0, 0 );

// Outside device can now DMA data from StreamStor within an
// address range size defined by DMA_size.
// The following simulates this by reading from StreamStor to memory.
pBuffer = (PUINT32)malloc(DMA_size);
pSSAddr = XLRGetWindowAddr( xlrDevice );

for( j = 0; j < loops; j++ )
{
   for( i = 0; i < DMA_size; i += 4 )
   {
      *pBuffer++ = *pSSAddr++;
   }
}
```

XLRSetSampleMode

Syntax:

```
XLR_RETURN_CODE XLRSetSampleMode( SSHANDLE xlrDevice, UINT32
bufSize, S_SAMPLEMODE mode )
```

Description:

`XLRSetSampleMode` turns sampling mode on/off and configures the buffer size of sampling.

To start sampling, call `XLRSetSampleMode` with *`bufSize`* set to the size of the sample and *`mode`* set to `SS_SAMPLEMODE_NORMAL` or `SS_SAMPLEMODE_PASSTHRU.`

To end sampling, call `XLRSetSampleMode` with a *`bufSize`* set to zero and *`mode`* set to `SS_SAMPLEMODE_DISABLED`.

Parameters:

`xlrDevice` is the device handle returned from a previous call to XLROpen.

`bufSize` is the size of the sample (in bytes) to grab from StreamStor. *`bufSize`* must be an eight byte-aligned value. The maximum sample size is 8 megabytes.

`mode` is the constant that defines the sampling mode for the StreamStor.

- `SS_SAMPLEMODE_NORMAL` – puts the StreamStor into normal sample mode. In this mode, you can retrieve samples while data is recorded to the StreamStor drives.
- `SS_SAMPLEMODE_PASSTHRU` – puts the StreamStor into passthru sample mode. In this mode, data is not recorded to the StreamStor drives. Instead, the StreamStor performs simultaneous input and real-time output of the data. This mode is used in situations where the data must be sampled in real-time and recording of the data is not necessary.
- `SS_SAMPLEMODE_DISABLED` – disables sampling mode.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
#define SAMPLE_SIZE  0x100000
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
// Configure sampling mode
xlrReturnCode = XLRSetSampleMode(
   xlrDevice, SAMPLE_SIZE, SS_SAMPLEMODE_NORMAL );
```

See Also:

XLRGetSample.

XLRSetUserDir

Syntax:

```
XLR_RETURN_CODE XLRSetUserDir( SSHANDLE xlrDevice, PVOID udirPtr,
   UINT32 udirSize )
```

Description:

`XLRSetUserDir` sets the user directory on the StreamStor recorder. The drives must be idle (i.e., not in record or playback mode) to set the user directory.

If the StreamStor is in bank mode, this command will set the user directory only on the selected bank. If the StreamStor is partitioned, this command will create a user directory on the selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`udirPtr` is a pointer to the buffer containing the user directory. The buffer can contain any type of data (string, binary, etc.). The maximum size of the user directory is XLR_MAX_UDIR_LENGTH. On Generation 5 boards such as the Amazon Express, udirPtr must be a multiple of eight (or sixteen on a 64-bit system). Therefore, you should dynamically allocate the memory (for instance, use malloc() which returns suitably aligned addresses).

`udirSize` is the size (in bytes) of the buffer pointed to by *`udirPtr`*. If *`udirSize`* is zero, then `udirPtr` is ignored and any existing user directory will have its length set to zero. The user directory must be 8 byte aligned (i.e., its `udirSize` must be a multiple of 8).

Note: This command can be very slow over the remote interface.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
#include <stdlib.h>

SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
char *            udirBuffPtr = NULL;
UINT32            dirLength = 0;

// Open the device
xlrStatus = XLROpen(1, &xlrDevice);

// Dynamically allocate the address space for the user directory.
udirBuffPtr = (char *)malloc(XLR_MAX_UDIR_LENGTH);

strcpy (udirBuffPtr, “1234567”);

//Add 1 to length for the NULL terminating character.
dirLength = strlen(udirBuffPtr) + 1;

xlrStatus = XLRSetUserDir(xlrDevice, udirBuffPtr, dirLength);
```

See Also:

XLRGetUserDir, XLRGetUserDirLength, XLRSetBankMode, XLRCreatePartition and XLRSelectBank.

XLRSetWriteProtect

Syntax:

```
XLR_RETURN_CODE XLRSetWriteProtect( SSHANDLE xlrDevice )
```

Description:

`XLRSetWriteProtect` marks a StreamStor recorder as write protected. After write protection is set, subsequent attempts to alter the recorded data (i.e., calls to `XLRRecord`, `XLRAppend` or `XLRErase`) will return an error. The drives must be idle (i.e., not in record or playback mode) to set the write protection.

Physical removal and reinsertion of the drives will not change the write protection. The only way to remove the write protection is to call `XLRClearWriteProtect`.

By default, drives are not write protected. The drives must be idle (i.e., not in record mode or playback mode) to set the write protection.

If the StreamStor is in bank mode, this command will set write protection only on the selected bank.

If the StreamStor is partitioned, this command will set write protection only on the selected partition.

IMPORTANT: the `XLRPartitionDelete` command ignores write protection.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

//
//We know we want to reuse these disks and that they were previously
//write protected.  Clear the protection so we can erase the drives
//and start a fresh recording.
xlrStatus = XLRClearWriteProtect( xlrDevice );
xlrStatus = XLRErase( xlrDevice, SS_OVERWRITE_NONE );

//Start recording.
xlrStatus = XLRRecord ( xlrDevice, 0,1 );
      …
xlrStatus = XLRStop( xlrDevice );

//Write protect this recording.
xlrStatus = XLRSetWriteProtect( xlrDevice );
      …
// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

XLRClearWriteProtect, XLRGetBankStatus, XLRSetBankMode, XLRSelectBank, XLRGetDirectory, XLRPartitionCreate, and XLRPartitionDelete.

XLRStop

Syntax:

```
XLR_RETURN_CODE XLRStop( SSHANDLE xlrDevice )
```

Description:

XLRStop will halt a recording operation and make sure all data is flushed to disk. This function should always be used to end a recording.

XLRStop can also be used to halt a playback initiated by XLRPlayback.

If the StreamStor is in multi-channel mode, calling XLRStop stops all recording (or playback) on all channels.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE         xlrDevice;
XLR_RETURN_CODE  xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRStop( xlrDevice );
```

See Also:

XLRRecord, XLRAppend, XLRPlayback and XLRSetMode.

XLRTruncate

Syntax:

`XLR_RETURN_CODE XLRTruncate( SSHANDLE` *`xlrDevice`*`, UINT32` *`AddrHigh`*`, UINT32` *`AddrLow`* `)`

Description:

`XLRTruncate` will truncate an existing recording at the address provided. The address must fall within the bounds of the currently recorded data set.

The truncation address must be an eight byte-aligned value.

If the StreamStor is in bank mode, this command will truncate data from the currently selected bank. If the StreamStor is partitioned, this command will truncate data from the currently selected partition.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`AddrHigh` is the upper 32 bits of the 64-bit truncation address.

`AddrLow` is the lower 32 bits of the 64-bit truncation address.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;
UINT32            AddrHi;
UINT32            AddrLo;

// Open the device
xlrStatus = XLROpen( 1, &xlrDevice );

// Append data
xlrStatus = XLRAppend( xlrDevice );
.
.
.
// Stop recording
XLRStop( xlrDevice );

//AddrHi and AddrLo must represent an appropriately aligned address.
AddrHi = 0;
AddrLo = 0xFE120000;

// Truncate the recording.
```

```
xlrStatus = XLRTruncate( xlrDevice, AddrHi, AddrLo );

// Close device before exiting
XLRClose( xlrDevice );
```

See Also:

XLRDeleteAppend, XLRSetBankMode and XLRSelectBank.

XLRWrite

Syntax:

XLR_RETURN_CODE XLRWrite(SSHANDLE *xlrDevice,* PS_READDESC p*WriteDesc*)

Description:

XLRWrite writes data from a user memory buffer to StreamStor. The StreamStor must be in record mode (XLRRecord or XLRAppend) before calling this function.

Parameters:

xlrDevice is the device handle returned from a previous call to XLROpen.

pWriteDesc is a pointer to an S_READDESC structure that holds the length and buffer address of the write data. Note that the AddrHigh and AddrLow parameters are ignored.

If the StreamStor is in bank mode, this command will write data to the currently selected bank.

If the StreamStor is partitioned, this command will write to the currently selected partition.

XLRWrite is not supported in multi-channel mode (since multi-channel mode is not supported over PCI.)

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE     xlrDevice;
S_READDESC   writeDesc;
UINT32       myBuffer[40000];

writeDesc.XferLength = sizeof( myBuffer );
writeDesc.BufferAddr = myBuffer;

// Open StreamStor.
if( XLROpen( &xlrDevice, 1 ) != XLR_SUCCESS )
   return(1);


//Put StreamStor into record mode.
if( XLRRecord( xlrDevice, 0, 1 ) != XLR_SUCCESS )
   return(1);

/* Fill the memory here . . . */

// Write the buffer to StreamStor.
if( XLRWrite( xlrDevice, &writeDesc ) != XLR_SUCCESS )
```

```
   return(1);
```

See Also:

XLRRecord, XLRAppend, XLRWriteData, XLRSetMode, XLRSetBankMode and XLRSelectBank.

XLRWriteData

Syntax:

```
XLR_RETURN_CODE XLRWriteData( SSHANDLE xlrDevice, PVOID BufAddr,
UINT32 TransferSize )
```

Description:

`XLRWriteData` is identical to `XLRWrite` except that the parameters are not passed in a structure.

If the StreamStor is in bank mode, this command will write data from the currently selected bank.

Parameters:

`xlrDevice` is the device handle returned from a previous call to `XLROpen`.

`BufAddr` is a pointer to the buffer to be written to StreamStor.

`TransferSize` is the number of bytes to write.

Return Value:

On success, this function returns XLR_SUCCESS.
On failure, this function returns XLR_FAIL.

Usage:

```
SSHANDLE          xlrDevice;
UINT32            myBuffer[40000];
XLR_RETURN_CODE   xlrReturnCode;

xlrReturnCode = XLROpen( 1, &xlrDevice );
      …
xlrReturnCode = XLRWriteData( xlrDevice, myBuffer, sizeof(myBuffer) );
```

See Also:

`XLRRecord`, `XLRAppend`, `XLRWrite`, `XLRSetMode`, `XLRSetBankMode` and `XLRSelectBank`.

Structure S_BANKSTATUS

```
typedef struct _BANKSTATUS
{
   char           Label[XLR_LABEL_LENGTH];
   UINT32         State;
   UINT32         Selected;
   UINT32         PowerRequested;
   UINT32         PowerEnabled;
   UINT32         MediaStatus;
   UINT32         WriteProtected;
   UINT32         ErrorCode;
   UINT32         ErrorData;
   UINT64         Length;
   UINT64         TotalCapacity;
   UINT64         TotalCapacityBytes;
} S_BANKSTATUS, *PS_BANKSTATUS;
```

Purpose

This structure is used by the `XLRGetBankStatus` function to return information about the StreamStor bank status of the specified bank.

Members

`Label` - String holding the bank label.

`Length` - Length of the bank's recording (in bytes).

`State` – A bank can be in any of 3 states: `STATE_READY`, `STATE_NOT_READY`, and `STATE_TRANSITION`. If the state is `STATE_READY`, the bank is ready for use. Otherwise, it is not ready, or it is transitioning to or from a ready or not ready state.

`Selected` – If `TRUE`, the specified bank is the currently selected bank.

`PowerRequested` - If `TRUE`, a power up request has been received for the bank. Otherwise, a request has not been received.

`PowerEnabled` – If `TRUE`, the bank has power. Otherwise, it does not.

`MediaStatus` – There are four possible values:

- `MEDIASTATUS_EMPTY` – indicates that the bank has no data recorded on it.
- `MEDIASTATUS_NOT_EMPTY` – indicates that the bank has some data recorded on it but it is not full.
- `MEDIASTATUS_FULL` – indicates that the bank is full.
- `MEDIASTATUS_FAULTED` – indicates that there is data on the bank, but for some reason the directory structure is corrupted. The bank must be erased before it can be used.

`WriteProtected` – If `TRUE`, the bank is write protected. Otherwise, the bank is not write protected.

ErrorCode – If 0 (zero), an error has not been detected on the bank. Otherwise, it is set to the error code.

ErrorData – If ErrorCode is non-zero, ErrorData holds any additional data about the error.

TotalCapacity – The bank's capacity in system pages.

TotalCapacityBytes – Same as *TotalCapacity*, expressed in bytes instead of pages.

Structure S_DBINFO

```
typedef struct _DBINFO
{
   UINT32 SerialNum;
   char   PCBVersion[XLR_VERSION_LENGTH];
   char   PCBType[XLR_MAX_NAME];
   char   PCBSubType[XLR_MAX_NAME];
   char   FPGAConfig[XLR_MAX_NAME];
   char   FPGAConfigVersion[XLR_VERSION_LENGTH];
   UINT32 NumChannels;
   UINT32 Param[8];
} S_DBINFO, *PS_DBINFO;
```

Purpose

This structure is used by the `XLRGetDBInfo` function to return information about the daughterboard.

Members

`SerialNum` – the daughterboard serial number.

`PCBVersion` – the daughterboard version.

`PCBType` – the daughterboard type.

`PCBSubType` – the daughterboard subtype.

`FPGAConfig` – the function type of the code loaded in the FPGA on the daughterboard.

`FPGAConfigVersion` – the version of the code loaded in the FPGA on the daughterboard.

`NumChannels` – the number of channels on the daughterboard.

`Param` - Reserved for future use.

Structure S_DEVINFO

```
typedef struct _DEVINFO
{
   char     BoardType[XLR_MAX_NAME];
   UINT32   SerialNum;
   UINT32   NumDrives;
   UINT32   NumBuses;
   UINT32   MaxBandwidth;
   UINT32   PciBus;
   UINT32   PciSlot;
   UINT32   NumExtPorts;
   UINT64   TotalCapacity;
   UINT64   TotalCapacityBytes;
}S_DEVINFO, *PS_DEVINFO;
```

Purpose

This structure is used by the `XLRGetDeviceInfo` function to return information about the StreamStor system configuration.

Members

`BoardType` - the board type (model name).

`SerialNum` - the serial number of the StreamStor board.

`NumDrives` - the number of drives currently connected and configured on the StreamStor controller.

`NumBuses` – the number of ATA buses in use.

`MaxBandwidth` – Reserved.

`PciBus` - the PCI bus number to which the StreamStor is connected.

`PciSlot` – the PCI slot number to which the StreamStor is connected.

`NumExtPorts` – the number of external ports.

`TotalCapacity` - the total recording capacity of the StreamStor system in system pages (a page is 4096 bytes typically on Intel based Windows systems). Note that if the StreamStor is partitioned, the reported *`TotalCapacity`* is the capacity of the currently selected partition.

`TotalCapacityBytes` – Same as *`TotalCapacity`*, expressed in bytes instead of pages.

Structure S_DEVSTATUS

```
typedef struct _DEVSTATUS
{
   BOOLEAN SystemReady;
   BOOLEAN BootmonReady;
   BOOLEAN Recording;
   BOOLEAN Playing;
   BOOLEAN Reserved1;
   BOOLEAN Reserved2;
   BOOLEAN Reserved3;
   BOOLEAN Reserved4;
   BOOLEAN RecordActive[XLR_MAX_VRS];
   BOOLEAN ReadActive[XLR_MAX_VRS];
   BOOLEAN FifoActive;
   BOOLEAN DriveFail;
   UINT32  DriveFailNumber;
   BOOLEAN SysError;
   UINT32  SysErrorCode;
   BOOLEAN Booting;
   BOOLEAN FifoFull;
   BOOLEAN Overflow[XLR_MAX_VRS];
   BOOLEAN PrefetchComplete;
}S_DEVSTATUS, *PS_DEVSTATUS;
```

Purpose

This structure holds various system status flags as returned by the `XLRGetDeviceStatus` function.

Note: The array index value is always 0 for RecordActive, ReadActive, VRActive, and Overflow

Members

`SystemReady` – System ready flag, indicates the system firmware and hardware have been initialized successfully.

`BootmonReady` – Power on boot flag, indicates that the system boot succeeded and the system is ready for initialization (`XLROpen`).

`Recording` – Indicates that the system is currently in a record mode.

`Playing` – Indicates that the system is currently in a playback mode.

`Reserved1, Reserved2, Reserved3 and Reserved4` – not used.

`RecordActive` – If not in bank mode, element 0 indicates that the system is currently recording. If in bank mode, element 0 indicates that BANK A is currently recording and element 1 indicates that BANK B is currently recording.

`ReadActive` – If not in bank mode, element 0 indicates that the system is currently reading. If in bank mode, element 0 indicates that BANK A is currently reading and element 1 indicates that BANK B is currently reading.

FifoActive – Indicates that the system is currently in FIFO mode.

DriveFail – Indicates that a drive has failed.

DriveFailNumber – Indicates the drive that has failed. Valid when DriveFail is TRUE.

SysError – Indicates that system initialization failed.

SysErrorCode – Holds initialization error code if SysError is TRUE.

Booting – For Conduant internal use only.

FifoFull – Indicates the system is at capacity while in FIFO mode.

Overflow – Indicates the disk drives reached capacity during a record operation. When in mode SS_MODE_FORK or SS_MODE_PASSTHRU (see XLRSetMode), Overflow gets set when the external port data has overflowed the available FIFO space.

PrefetchComplete - Indicates that initial prefetch of data from a read command has completed.

Structure S_DIR

```
typedef struct _DIR
{
   UINT64   Length;
   UINT64   AppendLength;
   BOOLEAN  Full;
   BOOLEAN  WriteProtected;
}S_DIR, *PS_DIR;
```

Purpose

This structure holds the directory information for the current recording. The structure is filled with a call to `XLRGetDirectory`. Use `XLRGetLengthPages` for environments that can't support 64 bit integers (`UINT64`).

Members

`Length` – The length of the current recording in bytes. Note that this parameter is a 64 bit number.

`AppendLength` - The length of the last set of data recorded using `XLRAppend`. Note that this parameter is a 64-bit number.

`Full` – This flag will be `TRUE` (non-zero) when the system has been filled to capacity.

`WriteProtected` – If not in bank mode, this flag will be `TRUE` (non-zero) if the system is write protected. If the system is in bank mode, this flag will be `TRUE` if the currently selected bank is write protected.

Structure S_DRIVEINFO

```
typedef struct _DRIVEINFO
{
   char     Model[XLR_MAX_DRIVENAME];
   char     Serial[XLR_MAX_DRIVESERIAL];
   char     Revision[XLR_MAX_DRIVEREV];
   BOOLEAN  SMARTCapable;
   BOOLEAN  SMARTState;
   UINT64   Capacity;
}S_DRIVEINFO, *PS_DRIVEINFO;
```

Purpose

This structure is used by the `XLRGetDriveInfo` function to return information about the disk drives on the StreamStor system.

Members

`Model` – Model name as reported by the disk drive identify command.

`Serial` – Drive serial number as reported by the disk drive identify command.

`Revision` – Drive revision level as reported by the disk drive identify command.

`SMARTCapable` – Indicates whether the drive has "SMART" capabilities. SMART is Self-Monitoring Analysis and Reporting Technology. You can query drives with this technology and determine if they are faulty. If *`SMARTCapable`* is `TRUE`, the drive has this feature. Otherwise, the drive does not have this feature.

`SMARTState` - On drives that are *`SMARTCapable`*, this structure member is used to indicate the drive's state. If *`SMARTState`* is `TRUE`, the drive is good. Otherwise, the drive is faulty. The value of this structure member is only valid if *`SMARTCapable`* is `TRUE`.

`Capacity` – Drive capacity as reported by identify command. Value is number of 512 byte sectors.

Structure S_EVENTS

```
typedef struct _EVENTS
{
   UINT32   Source;
   UINT32   AddressHigh;
   UINT32   AddressLow;
   UCHAR    Reserved[4];
} S_EVENTS, *PS_EVENTS;
```

Purpose

This structure is used by the `XLRGetEvents` function to return information about events that have been captured as a result of setting one or more event options when calling the `XLRSetDBMode` function.

Members

`Source` – Indicates the source of the event. It is bit significant, as follows:

- Bit 0 on = PIO1
- Bit 1 on = PIO2
- Bit 3 on = Sync

`AddressHigh` - Upper 32 bits of location in bit stream where event occurred.

- *`AddressLow`* - Lower 32 bits of location in bit stream where event occurred.

`Reserved` – Not used.

Structure S_PARTITIONINFO

```
typedef struct _PARTITIONINFO{
   BOOLEAN   Partitioned;
   UINT32    NumPartitions;
   UINT32    SelectedPartition;
   UINT64    SpaceAllocatedBytes;
   UINT64    SpaceAvailableBytes;
   UINT64    PartitionCapacityBytes;
}S_PARTITIONINFO, *PS_PARTITIONINFO;
```

Purpose

This structure is used by the `XLRGetPartitionInfo` function to return information about partitions.

Members

`Partitioned` – Indicates whether StreamStor is currently partitioned or not.

`NumPartitions` – The number of partitions currently on StreamStor.

`SelectedPartition` – Currently selected partition.

`SpaceAllocatedBytes` – The number of system bytes currently allocated in partitions (all partitions not just the currently selected partition).

`SpaceAvailableBytes` – The number of bytes of unpartitioned space available.

`PartitionCapacityBytes` – Size of the selected partition, in bytes.

Structure S_READDESC

```
typedef struct _READDESC{
   PUINT32   BufferAddr;
   UINT32    AddrHi;
   UINT32    AddrLo;
   UINT32    XferLength;
}S_READDESC, *PS_READDESC;
```

Purpose

This structure is used to define the parameters for a read from or a write to the StreamStor. (See, for example, `XLRRead` and `XLRWrite`).

Members

`BufferAddr` – Address of buffer to hold data read from StreamStor. Must be at least *`XferLength`* bytes.

`AddrHi` – High word (32 bit) of starting byte address.

`AddrLo` – Low word (32 bit) of starting byte address.

`XferLength` – Number of bytes to transfer from StreamStor.

Structure S_RECDCHANNELINFO

```
typedef struct _RECDCHANNELINFO
{
   UINT32    NumChannelsRecorded;
   UINT32    RecordedChannelNumber[MAX_NUM_CHANNELS];
}S_RECDCHANNELINFO, *PS_RECDCHANNELINFO;
```

Purpose

This structure is used by the `XLRGetRecordedChannelInfo` function to return information about the number of channels recorded and which channel numbers are recorded on this StreamStor device.

Members

`NumChannelsRecorded` – the number of channels recorded. This is also the number of valid RecordedChannelNumber[ ] array members.

`RecordedChannelNumber[]` – this array contains the list of channels that data was recorded on. The member *`NumChannelsRecorded`* indicates how many array members contain channel number data. As an example, if two channels of data were recorded on the StreamStor device, then NumChannelsRecorded would be set to 2 and only the first two array members in RecordedChannelNumber[] would have channel number data in them. The remaining array members would all be zero.

Structure S_SFPDPSTATUS

```
typedef struct _SFPDPSTATUS
{
   BOOLEAN PortOpticalEngPrsnt;
   BOOLEAN PortRXDataActive;
   BOOLEAN PortTXDataActive;
   BOOLEAN PortRcvProtocolOK;
   BOOLEAN PortNearEndFlowCtrlActive;
   BOOLEAN PortFarEndFlowCtrlActive;
   BOOLEAN PortCRCErrorDetected;
   BOOLEAN PortCRCErrPrevFrame;
}S_SFPDPSTATUS, *PS_SFPDPSTATUS;
```

Purpose

This structure holds various SFPDP (Serial FPDP) interface status flags as returned by the `XLRGetSFPDPInterfaceStatus` function for a specific port.

Members

`PortOpticalEngPrsnt` – A value of TRUE indicates the SFPDP daughter board has detected optical energy on this port.

`PortRXDataActive` – A value of TRUE indicates data is currently passing through the RX side of this port.

`PortTXDataActive` – A value of TRUE indicates data is currently passing through the TX side of this port.

`PortRcvProtocolOK` – A value of TRUE indicates Link up and receiving SFPDP protocol correctly on this port.

`PortNearEndFlowCtrlActive` – A value of TRUE indicates near end flow control active on this port.

`PortFarEndFlowCtrlActive` – A value of TRUE indicates Far end flow control active on this port.

`PortCRCErrorDetected` – A value of TRUE indicates a CRC Error has been detected on this SFPDP port at some time in the past.

`PortCRCErrPrevFrame` – A value of TRUE indicates a CRC Error was detected in the previous frame received on this SFPDP port.

Structure S_SMARTTHRESHOLDS

```
typedef struct _SMARTTHRESHOLDS{
   UCHAR     ID;
   UCHAR     Threshold;
   UCHAR     Reserved[10];
}S_SMARTTHRESHOLDS, *PS_SMARTTHRESHOLDS;
```

Purpose

This structure is used by the `XLRReadSmartThresholds` function to return SMART threshold values retrieved from SMART-capable disk drives.

Members

For information on interpreting the members of this structure, please refer to the ATA specifications or to the disk drive vendor's documentation.

Structure S_SMARTVALUES

```
typedef struct _SMARTVALUES{
   UCHAR     ID;
   USHORT    Status;
   UCHAR     Current;
   UCHAR     Worst;
   UCHAR     raw[6];
   UCHAR     Reserved;
}S_SMARTVALUES, *PS_SMARTVALUES;
```

Purpose

This structure is used by the `XLRReadSmartValues` function to return SMART values retrieved from SMART-capable disk drives.

Members

For information on interpreting the members of this structure, please refer to the ATA specifications or to the disk drive vendor's documentation.

Structure S_XLRSWREV

```
typedef struct _XLRSWREV
{
   char     ApiVersion[XLR_VERSION_LENGTH];
   char     ApiDateCode[XLR_DATECODE_LENGTH];
   char     FirmwareVersion[XLR_VERSION_LENGTH];
   char     FirmDateCode[XLR_DATECODE_LENGTH];
   char     MonitorVersion[XLR_VERSION_LENGTH];
   char     XbarVersion[XLR_VERSION_LENGTH];
   char     AtaVersion[XLR_VERSION_LENGTH];
   char     UAtaVersion[XLR_VERSION_LENGTH];
   char     DriverVersion[XLR_VERSION_LENGTH];
}S_XLRSWREV, *PS_XLRSWREV;
```

Purpose

This structure is used by `XLRGetVersion` to return software/hardware version strings.

Members

`ApiVersion` – Version of the StreamStor API library.

`ApiDateCode` – Build date of the StreamStor API library.

`FirmwareVersion` – StreamStor firmware version.

`FirmDateCode` – Build date of the firmware.

`MonitorVersion` – Boot monitor firmware version.

`XbarVersion` – Controller logic version.

`AtaVersion` – ATA controller version.

`UAtaVersion` – Ultra ATA controller version.

`DriverVersion` – Driver version.

Chapter 3
PCI Integration

PCI Integration

To allow maximum bandwidth for recording digital data over the PCI bus, StreamStor is designed for direct card-to-card data transfers. Since many data acquisition cards already perform DMA operations directly to system memory, the StreamStor controller uses this capability for the direct transfer of data. The software development kit provides the necessary control functions for integration of StreamStor into user applications.

Initialization and Setup

Initialization requires a call to the `XLROpen` function. This function will lock the device for exclusive access and initialize the recording system. The initialization routine includes locating the StreamStor controller on the PCI bus, downloading software and initializing required data structures, etc.

PCI Bus Interfacing

Although the PCI bus itself has been designed for card-to-card transactions, most operating systems have no provisions for this functionality. In addition, most operating systems do not have provisions for real-time event management, which is required when recording data at high bandwidths. For these reasons, there may be a requirement to modify existing device drivers for the PCI card that is to send data to the StreamStor system.

The StreamStor controller requests a memory mapped window during computer booting providing a memory space for writing data to be recorded. The default size of this window is 8MB although you should use the `XLRGetBaseRange` to verify this in your application. The StreamStor SDK provides two functions that return the physical and logical addresses of this window.

The address returned by `XLRGetBaseAddr` is the physical address that is assigned to the StreamStor data window during the boot process. The StreamStor PCI interface chip will respond to any memory writes on the PCI bus in this address range. Note, however, that the StreamStor system does not utilize the address to determine where to store the data. Any data writes are recorded to disk in the order they are received. This physical address can be used directly for programming DMA hardware on the PCI data source device. Various techniques can be used for programming the DMA hardware but generally you will need to set up a DMA block transfer that continuously recycles back to the original starting address. If the DMA hardware supports chaining (scatter/gather) then a looping transfer can be set up. Consult the documentation for your PCI data acquisition card for more information.

☠ CAUTION: ***The physical address returned by XLRGetBaseAddr cannot be used in place of a buffer memory address. Use XLRGetWindowAddr instead.***

The address returned by `XLRGetWindowAddr` is a logical address created by the operating system to "map" the physical address space of the StreamStor controller into the application memory space. This address can sometimes be used with software provided by PCI card vendors in place of the address of a memory buffer. Check with Conduant about your specific environment for more details. In addition, "writing" to this address space from an application is an effective method to save application specific directory or indexing information about the recording. It is the responsibility of the user application to manage this type of data.

Multi-Card Operation

Multiple StreamStor cards can be used in a single system either on the same bus or on "bridged" PCI buses. If multiple StreamStor cards are installed into the same bus there will be contention for ownership of the bus during data transfers and the effective bandwidth will be reduced. If multiple StreamStor cards are installed on opposite sides of a PCI-PCI bridge than there is no loss in bandwidth as long as the data capture card is co-located on the same bus as the StreamStor card it is streaming data to.

Software applications gain exclusive access to a StreamStor card after calling the `XLROpen` function. Until the application exits or calls `XLRClose`, no other application may connect to that StreamStor card. A single application can connect to and control multiple StreamStor cards but must manage the unique handles returned from multiple calls to the `XLROpen` function. The index number passed into `XLROpen` determines which card is to be controlled by the handle returned. If multiple applications (or multiple instances of the same application) are used to control StreamStor cards, they must each connect to a unique StreamStor card. The `XLRDeviceFind` function returns the number of StreamStor devices found in the system. The index number cannot be larger than this number. In most cases, the higher value index indicates a card that is on a bus or slot further from the main bus. The PCI bus number and slot number are available from the `XLRGetDeviceInfo` command. The command can be used to identify the appropriate card in a multi-card system.

Chapter 4
Operation

Operation

The operation of StreamStor for recording data is very similar to the familiar interface of a tape recorder. The `XLRRecord` function puts the recorder into record mode and the `XLRStop` function ends the recording. Data reading is more like a traditional computer storage device since the data can be retrieved randomly. The StreamStor recorder also has a special "wrap" mode to allow continuous recording past the capacity limits of the disks by overwriting the oldest data.

Data Recording

After getting the base address of the data window using `XLRGetBaseAddr`, it is used to setup the DMA hardware on the data acquisition card for direct slave writing to the StreamStor controller. Because the capacity available on StreamStor is much larger than the 32 bit PCI address scheme (4 GB) will allow, the system is designed to ignore PCI addressing and assume any data written within the PCI address range is data to be recorded sequentially. The actual size of the data window can be found with a call to `XLRGetBaseRange` (default: 8MB). The PCI data source card is required to maintain a destination address within this range. This can easily be accomplished with DMA chaining or other techniques. For example, the data acquisition card can be programmed to start at the base address, write 64kB, than start over again at the base address for the next 64kB, etc.

Recording Data

To start a recording the application must call the `XLRRecord` function. Once `XLR_SUCCESS` status has been returned from this function, StreamStor will record all data written to its data address range. This function should be called BEFORE starting the flow of data to prevent overflow on the data source device. The user application can periodically sample the device status using `XLRGetDeviceStatus` to check for errors that occurred during recording. Note that this function call generates PCI traffic and can impact data transfer bandwidth if used excessively.

Many data acquisition cards have operating modes that allow the capture of a specific number of data points. Unfortunately, the software does not usually allow specifying a number larger than a 32-bit integer (4,294,967,295). For this reason it may be necessary to use the data acquisition card in a "pre-trigger" mode where data is captured continuously until the trigger and then a specified number of data points are captured after the trigger. The data acquisition card will then continuously cycle through its "memory buffer" until receiving the trigger. StreamStor will continuously record all of the data, however, up to its full capacity. To use the recorder in this fashion, you should enable the "Wrap" feature in the `XLRRecord` function so that StreamStor will overwrite the oldest data if the disk system is full.

In order to capture the maximum amount of data without overwriting old data the StreamStor system is designed to "exit" record mode when the disk subsystem is filled to capacity (unless "Wrap" has been set). The user application can poll the device status using `XLRGetDeviceStatus` watching for `Recording` to go `FALSE`. A normal `XLRStop` command should then be used to end record mode. Note that the StreamStor controller is designed to accept data on the PCI bus even after the disk subsystem is full to prevent system errors and allow you to shut down the data source after completely filling the available disk space.

Data Wrap

In some recording applications, it is desirable to continue recording past the capacity of the recording system by overwriting the oldest recorded data. This is sometimes called "pretrigger" or "circular" recording. The StreamStor system supports this recording mode by setting the "Wrap" bit in the `XLRRecord` command. The recorder will continue to record after the disk capacity is exhausted by overwriting the oldest data on the disks. Once the recording is finally stopped, the `XLRGetLength` command can be used to determine how much data has been recorded. If your data is blocked in anything other than 4 byte blocks, you will need to index back from the end of the data to find an aligned start point of your data. Contact technical support for more information on using this feature.

Ending the Recording

If storage wrapping mode has not been enabled, StreamStor will continue to record data until all recording space has been exhausted or the `XLRStop` function has been called. If the `XLRStop` function is not used, any data written to the StreamStor data range after space is exhausted will be lost.

If data wrapping has been enabled, StreamStor will continue to record data indefinitely until the `XLRStop` function is called. When free storage space has been exhausted, the system will begin to overwrite the oldest data so that the newest data is kept.

☞ **NOTE:** ***A data acquisition system can stop recording by simply ceasing any writes to the StreamStor data address range. The `XLRStop` function should still be used to flush all data to the disk drives and to prepare for reading of the data.***

Data Read

Because operating systems cannot handle the massive file sizes resulting from a long recording, the SDK provides a read function for retrieving data from the recorder. The user application must supply a memory buffer sufficient to hold the data requested. Note that the StreamStor system will have optimum read performance when reading is performed sequentially from the device.

Read Setup

The StreamStor device must be previously opened with `XLROpen` before reading data or performing other operations.

If the recording was done with wrapping enabled (old data may be overwritten), use the `XLRGetLength` command to get an accurate count of the bytes recorded. This number can then be used for indexing into the data.

Read Positioning

A structure is used to set the read pointer with a byte-offset count. A high and low value is used to overcome the 32 bit limitations of some programming environments.

Reading Data

An `XLRRead` command is used to request a data transfer from StreamStor to system memory.

Chapter 5
External Port

External Port

Some models of StreamStor include additional connectors and electronics to provide an alternate method of transferring data into and out of StreamStor. These additional paths offer several advantages, including:

- freedom from interaction with other devices on an arbitrated bus such as PCI;
- the reduction or elimination of bus FIFOs that may otherwise be required to interface with an arbitrated bus;
- full isolation of data path from operating system and computer hardware facilitates predictable and repeatable behavior;
- better or additional control over timing and other parameters;
- higher bus utilization efficiency due to non-arbitrated nature;
- access to interface signals without risk of crashing host computer;
- higher data rates than the most common PCI buses support; and
- the potential for dual-port operation (simultaneous transfers on both PCI bus and external ports) while recording or playing back.

FPDP

Overview

The FPDP (Front Panel Data Port) external port feature is included on a variety of StreamStor controllers. The hardware manual for your StreamStor controller or daughterboard will indicate if FPDP is supported on it and will provide any other model-specific details.

FPDP is a 32-bit synchronous data bus that allows data to be transferred at high speeds between devices. Simple and low-cost in its implementation, FPDP supports the necessary flow controls to manage transfers between devices of different speeds. The sustained speed on the StreamStor interface varies, depending on the StreamStor controller model.

In reading the following sections on using this feature, it is important to be familiar with the American National Standard for Front Panel Data Port Specifications (ANSI/VITA 17-1998). This manual is intended to clarify StreamStor's operation as it relates to the standard, not to educate one on the standard itself. For additional information about the standard, other FPDP products and manufacturers, and other technical details regarding FPDP, please visit www.fpdp.com.

The StreamStor FPDP interface is designed to meet and exceed the basic capabilities of FPDP as defined in the FPDP ANSI standard. The following sections describe:

- any optional FPDP features StreamStor has implemented;
- any features that StreamStor has implemented as a superset to the standard;
- any known deviations form the ANSI standard;
- any clarifications that might otherwise be left open to interpretation; and
- the API functions necessary to configure an external port.

Interface Electronics

Interface electronics and termination values on StreamStor are those recommended by the ANSI standard, though some signals and terminations can be electronically connected or isolated with crossbar switching devices in order to support electronic reconfiguration.

Data Formats

The FPDP is a multi-drop bus intended to carry either framed or unframed data. StreamStor currently supports only the unframed data mode. The SYNC* (Sync Pulse) signal is driven to an inactive state while StreamStor is a data transmitter on the FPDP bus.

Contact Conduant for more information on using framed data.

PIO Signals

PIO signals are programmable lines for I/O for user-defined functions. These are ancillary signals and are not required for the FPDP function. StreamStor currently does not drive or act on received PIO signals. Contact Conduant for more information on using PIO signals.

Interface Functions

To ready StreamStor to transfer data using FPDP, the API routine `XLRBindxxxChannel` must be called. The FPDP port's channel number will depend on the board type. (For details on channel numbers, see the `XLRSelectChannel` function in the Function Reference section of this manual.) The bind function is called as follows (*xxx* stands for "Input" or "Output" depending on intended usage):

```
XLRBindxxxChannel ( device, 0 );
```

After StreamStor is in external port mode, an API call to `XLRSetDBMode` is used to configure the port. This command allows you to set the mode to one of:

- FPDP Transmit Master (FPDP/TM)
- FPDP Transmit (FPDP/T, StreamStor unique)
- FPDP Receive (FPDP/R)
- FPDP Receive Master (FPDP/RM).
- FPDP Receive Master Clock Master (FPDP/RMCM, StreamStor unique)

The hardware manual for your StreamStor controller or daughterboard describes which FPDP modes are available for your board type.

In FPDP/T mode, StreamStor drives the FPDP DATA, DVALID* (Data Valid), DIR* (direction), and SYNC* (Sync Pulse) signals but uses the FPDP clock that is driven to the FPDP bus by some other source. In this mode,

StreamStor does not provide any termination for signals other than DATA[1]. To use this mode properly, StreamStor should NOT be positioned at either end of the FPDP bus. Note also that the maximum useable frequency in this mode will decay more rapidly as the cumulative distance from the clock source to the data source to the data destination increases.

In FPDP/RMCM mode, StreamStor acts as a Receive Master, excepting that StreamStor also drives the FPDP clock signals on the FPDP bus. In addition, StreamStor terminates the clock signals (PSTROBE, PSTROBE*, and STROB) as would a traditional FPDP/TM while terminating the remaining signals as would a FPDP/RM. To use this mode StreamStor should be physically positioned at an end of the FPDP bus. Note also that the maximum useable frequency in this mode will decay more rapidly as the cumulative distance from the clock source to the data source to the data destination increases.

When configuring StreamStor as a recorder, it may be desirable to prevent a transmitter from sending data until the StreamStor recording function is fully enabled. `XLRSetDBMode` can be used to assert the FPDP NRDY* (Not Ready) signal when StreamStor is activated as a FPDP receiver. NRDY* will remain asserted until the StreamStor data recording process is ready to proceed. An example of this is (for a PCI-816XF2):

```
XLRSetDBMode(device,SS_FPDP_RECVMASTER,
SS_OPT_FPDPNRASSERT);
```

IMPORTANT: When connecting and configuring FPDP/FPDP-II connections on a given bus, do not configure more than one connector as a transmitter (FPDP/TM or FPDP/T) on that bus at a time. Otherwise, bus drivers may be permanently damaged.

[1] StreamStor always provides series termination on the DATA signals as described in Permission 6.4.1 of the ANSI specification.

PSTROBE/PSTROBE* and STROB Signals

When in FPDP/TM and FPDP/RMCM modes, StreamStor will drive and terminate both the differential clock pair of PSTROBE, PSTROBE* (± PECL Data Strobe) and the single-ended STROB (Data Strobe) TTL clock. When in any other mode, the user will select which of the two FPDP clock sources StreamStor should use from the FPDP bus. The clock can be selected by calling `XLRSetDBMode` with the desired clock option. For example, to enable the data strobe clock (TTL) on a PCI-816XF2:

```
XLRSetDBMode(device,SS_FPDP_RECV,  SS_OPT_FPDPSTROB
);
```

Refer to the FPDP ANSI standard for recommendations and observations about the use of these signals.

Chapter 6
Channel Description and Selection

Channel Description and Selection

StreamStor boards have one or more data paths or channels that can be used to input and output data to/from the StreamStor board. The number of channels available depends on the board type. For example, the PCI-816XF2 has three channels - the PCI Bus, the FPDP top connector, and the FPDP front connector. The hardware manual for your StreamStor controller or daughterboard describes what channels are available for the specific board type.

A single channel or multiple channels may be selected to record from. Only one channel at a time can be selected to playback with the FPDP board type. The SFPDP board type allows up to four channels to be played back at a time. This section describes the commands that should be used to set up the StreamStor channels for recording and playback.

For StreamStor users who have software that was written prior to SDK 7.0 and currently use the PCI Bus to transfer data to/from the StreamStor card, your software does not need to be modified to support the new channel options. That is, calls to the following API functions do not need to be added to your software since the defaults are set to single channel mode using the PCI Bus channel 0.

- `XLRSetMode`
- `XLRBindInputChannel`
- `XLRBindOutputChannel`
- `XLRClearChannels`
- `XLRSelectChannel`

Calls to these functions only need to be added if:

- you want to use the multi-channel options or
- you want to use a channel other than the default channel (for example, if you want to use the FPDP front connector on a PCI-816XF2 instead of the FPDP top connector).

Channel Description

The hardware manual for your StreamStor controller or daughterboard describes the available channels on your board type.

Selecting an Operating Mode

The StreamStor operating mode is set by calling `XLRSetMode`. The hardware manual for your StreamStor controller or daughterboard describes the operating modes that are available for your board type. The default operating mode is `SS_MODE_SINGLE_CHANNEL` (single channel mode).

Clearing, Selecting, and Binding Channels

The `XLRSelectChannel` function is used to select a channel that future functions will act on. One example is `XLRSelectChannel` needs to be called to select the FPDP channel before a call to `XLRSetDBMode` is made. `XLRSelectChannel` should always be called to select a channel before calling `XLRBindInputChannel` or `XLRBindOutputChannel`. `XLRClearChannels` should be called once before setting up the channels to clear the default channels or to clear the previous setting of channels. Since `XLRClearChannels` clears the previous settings of channels, when setting up multiple channels `XLRClearChannels` should be called just once before the setting up of the channels.

If you do not want to use the default channels, then you must identify which channels you want to use. The process of identifying a channel is called *binding* a channel. Binding a channel is analogous to choosing the data path. The function `XLRBindInputChannel` is used to bind a channel for input into StreamStor and the function `XLRBindOutputChannel` is used to bind a channel for output from StreamStor. These functions should be called before data is transferred to or from the StreamStor board. The default channel for record and playback is the PCI Bus channel 0.

To record and playback a single channel, call `XLRSelectChannel` and then `XLRBindInputChannel` passing it the channel to record on. Then to playback, call `XLRSelectChannel` and then `XLRBindOutputChannel`, passing it a parameter of the channel to playback through.

The SFPDP daughter board supports a maximum of four channels that can be recorded simultaneously by the StreamStor board and four channels can be played simultaneously. To record multiple channels simultaneously, the function `XLRBindInputChannel` must be called once for each channel to be recorded upon. Note that all the channels you are going to use for recording must be bound and configured (with `XLRSetDBMode`) before you call `XLRRecord`.

Channels that are recorded in multi-channel mode retain the channel number they were recorded on, so when a channel needs to be played back, this channel number must be selected using `XLRSelectChannel.` On playback, both the channel and data output path need to be selected. The channel of data to be played back is selected using `XLRSelectChannel` and the data output path is

selected using `XLRBindOutputChannel`. This method allows the data to be output on a different channel than it was recorded on.

This is a typical calling sequence to playback data recorded using multiple channels:

1. Call `XLRClearChannels` once.
2. For each channel, call `XLRSelectChannel` to select the channel number to playback.
3. Call `XLRBindOutputChannel` to select the output path.
4. Call `XLRSetDBMode` to setup the daughterboard mode.

The order of calling these functions is very important. See the coding examples at the end of this chapter.

Since a recorded channel retains the channel number it is recorded on, only one channel number can recorded over each physical channel. This means only one channel number can be recorded over the PCI bus (channel 0).

Forking and passthru are not supported when in multi-channel mode.

SFPDP Multi-channel Commands

The StreamStor Amazon Real-time Disk Controller Installation and User Manual lists the available commands for the Amazon controller. Of the list of API commands in the Amazon User Manual, these commands are not supported if using the SFPDP daughterboard in multi-channel mode:

- `XLRAppend`
- `XLRDeleteAppend`
- `XLREdit/XLREditData`
- `XLRGetFIFOLength`
- `XLRGetSample`
- `XLRPlaybackLoop`
- `XLRPlayTrigger`
- `XLRReadFifo`
- `XLRRecoverData`
- `XLRSetPlaybackLength`
- `XLRSetReadLimit`
- `XLRSetSampleMode`
- `XLRTruncate`

The API command `XLRSetPortClock` is not available on the SFPDP daughter board, regardless of the mode.

Some API commands are “channel specific.” This means that the command operates on the currently selected (or selected and bound) channel. Examples of channel specific commands are:

- `XLRGetDirectory`
- `XLRGetLength`
- `XLRGetLengthPages`

In the case of `XLRGetLength`, if data on the StreamStor was recorded in multi-channel mode, it will return the length of data recorded on the currently selected channel.

Programming examples of multi-channel recording and playback can be found in the Examples directory of the SDK distribution.

Example 1

```
/* The following C code shows how to set up the PCI Bus channel 0 as
 * an input channel to record, and then read the data back
 * through the PCI Bus channel 0.  For simplicity, error handling
 * is not shown.
 */

/** include files **/
#include <stdio.h>
#include "xlrapi.h"
#define START_ADDRESS         0x100000

void main()
{
   SSHANDLE             xlrDevice;
   S_READDESC           sRead;
   PUINT32              pBuf = NULL;
   UINT64               dwAddress = 0;
   XLR_RETURN_CODE      xlrStatus;

   xlrStatus = XLROpen( 1, &xlrDevice );
      …
   // Set StreamStor mode to Single Channel
   xlrStatus = XLRSetMode( xlrDevice, SS_MODE_SINGLE_CHANNEL );

   // Channels must be cleared prior to binding.  XLRClearChannels
   // clears the input and the output channels.
   xlrStatus = XLRClearChannels( xlrDevice );

   // Select channel zero to begin recording on.
   xlrStatus = XLRSelectChannel( xlrDevice, 0 );

   // Input will be done over the PCI Bus, which is channel zero.
   xlrStatus = XLRBindInputChannel( xlrDevice, 0 );
      -
   // Record for a while on channel zero.
   xlrStatus = XLRRecord( xlrDevice, 0, 1 );

   printf( "Recording..." );
      … record for a while …
   printf( "Recording done!\n" );

   // Stop recording.
   XLRStop( hTarget );

   // Select Channel to read – channel 0
   xlrStatus = XLRSelectChannel( xlrDevice , 0 );

   // Bind PCI Bus channel 0 as output channel.
   xlrStatus = XLRBindOutputChannel( xlrDevice, 0 );

   pBuf = (PUINT32)malloc( BUFFER_SIZE );
```

```
   // Build the read descriptor.
   dwAddress = (UINT64)(START_ADDRESS);
   sRead.AddrHi = (UINT32)( dwAddress << 32 );
   sRead.AddrLo = (UINT32)( dwAddress );
   sRead.BufferAddr = pBuf;
   sRead.XferLength = BUFFER_SIZE;

   xlrStatus = XLRRead( xlrDevice, &sRead );

   printf( "Read Complete.\n" );
   XLRClose( xlrDevice ) ;

   if( pBuf )
   {
      free( pBuf );
      pBuf = NULL;
   }
}
```

Example 2

```
/*
 * The following C code shows how to set up the top FPDP connector as
 * an input channel to record, then read the data back through the
 * PCI Bus Channel 0.
 */

/** include files **/
#include <stdio.h>
#include <string.h>
#include "xlrapi.h"

#ifndef WIN32
#include <stdlib.h>   // for malloc
#endif

#define BUFFER_SIZE     131072

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

#define START_ADDRESS         0x100000

void PrintXLRError();
int main(int argc, char *argv[])
{
   SSHANDLE             hTarget;
   S_READDESC           sRead;
   S_DEVINFO            devInfo;
   UINT64               dwStartAddress=0;
   PUINT32              pBuf = NULL;
   XLR_RETURN_CODE      xlrStatus;

   xlrStatus = XLROpen(1, &hTarget);
   xlrStatus = XLRSetMode(hTarget, SS_MODE_SINGLE_CHANNEL);
   xlrStatus = XLRClearChannels(hTarget);

   // Bind the top port (channel 30) as the input channel.
   xlrStatus = XLRSelectChannel(hTarget, 30);
   xlrStatus = XLRBindInputChannel(hTarget, 30);

   xlrStatus = XLRGetDeviceInfo(hTarget, &devInfo);
   if (strncmp(devInfo.BoardType, "AMAZON", 6) == 0) {
      xlrStatus = XLRSetDBMode(hTarget, SS_FPDPMODE_RECVM, 0);
   }
   else {
      xlrStatus = XLRSetDBMode(hTarget, SS_FPDP_RECVMASTER, 0);
   }

   // Start recording.
```

```
   xlrStatus = XLRRecord(hTarget, FALSE, TRUE);

   printf("Recording...\n");

   //  ... Record some data  ...

   printf("Recording done!\n");
   XLRStop(hTarget);

   // Set up to read the data we just recorded.
   xlrStatus = XLRSetMode(hTarget, SS_MODE_SINGLE_CHANNEL);
   xlrStatus = XLRClearChannels(hTarget);

   // Select Channel to read - channel 0.
   xlrStatus = XLRSelectChannel(hTarget, 0);

   // Bind PCI Bus channel 0 as output channel.
   xlrStatus = XLRBindOutputChannel(hTarget, 0);

   pBuf = (PUINT32)malloc(BUFFER_SIZE);

   // Build the read descriptor to read some data.
   dwStartAddress = 4096;
   sRead.AddrHi = (UINT32)(dwStartAddress >> 32);
   sRead.AddrLo = (UINT32)(dwStartAddress & 0xFFFFFFFF);
   sRead.BufferAddr = pBuf;
   sRead.XferLength = BUFFER_SIZE;

   // Read a buffer.
   xlrStatus = XLRRead(hTarget, &sRead);

   printf("Read Complete.\n");

   XLRClose(hTarget);

   free(pBuf);
   pBuf = NULL;
   exit(0);
}
```

Using Multiple PCI Express Sources

Overview

The StreamStor system can record data from multiple sources simultaneously and aggregate the data onto the media for maximum performance. This means that the data is interleaved on the media but is tracked to allow separation by channel when played back or retrieved to system memory. Recording from multiple channels using one of the StreamStor mezzanine boards provides an obvious distinction for the StreamStor system to identify the source channel in order to properly track data from each independent channel. Unfortunately, when recording from multiple sources on the PCI Express fabric there is no independent physical interface to provide this channel distinction. In order to provide this functionality the StreamStor system implements multiple channels on the PCI Express fabric by partitioning the address space assigned for data recording. These "virtual" address spaces provide a means to identify the data source if the source restricts its data writes to stay within its assigned address space.

The StreamStor multi-channel recording capability on PCI Express should not be confused with multi-channel data sources such as A/D cards. A StreamStor channel provides the capability to distinguish between PCI Express data sources (cards) but does not distinguish individual channels from one source. For example, a high speed digitizer card might support the connection of 2 signals for digitizing and streaming to system memory. The digitizer and/or its software determine how this data is combined into the single data stream that is written to system memory. This data stream represents a single "channel" to the StreamStor. The system currently supports 16 virtual channels, as defined by `MAX_VIRTUAL_CHANNELS` in `xlrtypes.h`.

Address Allocation

The StreamStor address space is allocated at system boot time and the requested size is pre-configured. In general, the StreamStor PCI Express products are allocated a total memory address space of 16MB (0x1000000) for data recording. For single source recording applications, the entire memory address space is usable for recording. When operating in multi-channel mode, the StreamStor device will partition this space according to how many PCI Express input channels are bound for recording. The space allocation is done on binary multiples (1, 2, 4, 8, and 16, etc.).

If, for example, there are 5 PCI Express input channels bound, the memory will be partitioned into 8 equal size pieces. If 4 input channels are configured then the memory space will be split into 4 equal size pieces. See the example in Figure 1. The example assumes the StreamStor default total memory allocation size of 16MB.

Figure 1 - PCI Express Multi-Channel Recording Address Space Example

Number of Channels Configured	Address Space	Range (each space)
2	2	0x800000 (8 MB)
3 or 4	4	0x400000 (4 MB)
5, 6, 7, or 8	8	0x200000 (2 MB)
9 through 16	16	0x100000 (1 MB)

The functions `XLRGetBaseAddr`, `XLRGetBaseRange` and `XLRGetWindowAddr` can be used to query the configured addresses for each recording channel. Note that `XLRSelectChannel` must be used to select the channel before calling these functions since the values returned are channel specific.

Configuration

For recording multiple sources on the PCI Express interface, the StreamStor system must be configured for multi-channel mode using the `XLRSetMode` function. The functions `XLRSelectChannel` and `XLRBindInputChannel` functions must then be called to configure a channel for each unique PCI Express source device.

Virtual channel numbers start at 0. The channel numbers you use must start at 0 and must be contiguous.

Examples of valid virtual channel combinations:

0

0, 1, 2, 3

0, 1, 2, 3, 4

Examples of invalid virtual channel combinations:

```
1, 2, 3      # does not start at 0

0, 1, 4      # numbers are not contiguous

13           # does not start at 0
```

It is left to the user to assign channels and associated address ranges to each PCI Express data source being utilized.

As in any recording operation, the StreamStor system must be placed into record mode using `XLRRecord` or `XLRAppend`. To record data from a PCI Express source, the device software must be programmed to stream data to the StreamStor physical address assigned to the channel for that device. The programmer must also ensure that the data transfers stay within the address range assigned for the channel. In many cases, the device software is designed to allow streaming of acquired data directly to system memory buffers. The address returned from the `XLRGetWindowAddr` function can be used in place of an allocated system memory address. The address returned is a logical address in the user address space that will be remapped to the StreamStor physical address by the operating system and device driver when the source device is programmed for the transfer. If the programmer has direct access to the device hardware, it is often desirable to utilize direct programming of the device hardware to minimize OS overhead by using the physical address returned by the `XLRGetBaseAddr` function.

For detailed information on the StreamStor API functions described above please refer to the StreamStor User Guide.

Chapter 7
Bank Switching

Bank Switching

The Big River TK200 (“TK200”) is a rack mounted StreamStor storage system. It features two hot-swappable 8-drive modules that can be used to record continuously. The bank switching feature is available only on the Big River TK200.

Bank switching is used to control drive modules in separate banks as if they were contiguous units in a recording. A *bank* is a rack containing a drive module. The TK200 has two banks, referred to as Bank A and Bank B. Each bank can hold a drive module, and each drive module can hold up to four pairs of master/slave drives for a total of eight drives. Therefore, a TK200 can support a maximum of 16 drives. You can play or record data from the drive module in one bank while the other bank is idle or dismounted.

Setting Bank Mode

The TK200 can operate in bank mode or non-bank mode. When in non-bank mode, the TK200 operates the same as a non-TK200 system; namely, the drives in the drive modules are read and written as if they were a single set.

Only StreamStor systems with a TK200 chassis support bank mode. The chassis type can be determined by calling the API function `XLRGetChassisType`. Currently, the chassis type returned is either TK200 or `UNKNOWN_CHASSIS_TYPE`.

When in bank mode, the drives in each drive module are recorded independently. That is, a recording made on the drives on Bank A is independent of the recording made on the drives on Bank B. In this mode, you can load a single bank with a drive module or both banks with drive modules and begin recording or playback.

The API function `XLRSetBankMode` is used to enable or disable bank mode on the TK200. By default, bank mode is disabled - you must call `XLRSetBankMode` to enable it. (StreamStor PCI-816V100 and PCI-816VXF2 boards are an exception. By default, they are bank mode enabled.) To enable bank mode, the StreamStor recorder must be idle (not recording or playing data).

When a StreamStor is in bank mode, it stays in bank mode until `XLRSetBankMode` is called to take it out of bank mode or the StreamStor card is reset.

Assume both banks are loaded with drive modules. When the drive module in Bank A is full, you could switch recording to Bank B. While Bank B is

recording, you could replace the full drive module in Bank A with a new drive module. In this way, you could continue recording as long as you want, switching out full drives with empty drives indefinitely.

If bank mode is disabled, or if the chassis or board type does not support bank mode, all API functions operate on the drives as if the drives were a single device rather than two independent banks.

Selecting a Bank

Some API functions are "bank aware", which means that when the TK200 is in bank mode, the API function will be performed on the selected bank. The selected bank is identified by bank aware API functions as follows:

- If the system is not in bank mode and then `XLRSetBankMode` is called to enable it, Bank A, if it is available, is by default the selected bank. If Bank A is not available (i.e., there is no drive module in Bank A or Bank A's drive module is faulty), then Bank B, if it is available, becomes the selected bank.

- If the system is in bank mode and then `XLRSelectBank` is called, the bank specified in the `XLRSelectBank` call becomes the selected bank. It remains the selected bank until `XLRSelectBank` is called to select a different bank.

- If the system is in bank mode and the StreamStor is closed by calling `XLRClose`, then the next time `XLROpen` is called, it will still be in bank mode. The selected bank will be the bank that was selected at the time of the last call to `XLRClose`.

For example, assume that the StreamStor is in bank mode and `XLRSelectBank` has been called to select Bank B. Then `XLRRecord` is called. In this case, recording will begin on Bank B. If `XLRGetDirectory` is then called, the length returned will be the length of the recording on Bank B.

To determine which bank is selected, call `XLRGetBankStatus` and examine the selected member of the `S_BANKSTATUS` structure.

Recording a Drive Module

When the TK200 is in bank mode, API routines such as `XLRRecord` and `XLRWrite` can be used to put the recorder into record mode. If both banks contain drive modules, data will be written on the drive module in the selected bank until the drive module is full. When full, recording ceases on the selected bank. Your application may then explicitly select the other bank to resume recording.

Recording continues on the newly selected bank until its drive module, too, is full. As long as full drive modules are replaced by modules that are not full, recording can continue in this fashion, with recording alternating between the two banks, until `XLRStop` is called.

Playing back from a Drive Module

In bank mode, the standard `XLRPlayback`, `XLRRead` or `XLRReadData` functions can be used to playback or read data into memory. If both banks contain drives with data, data will be played back from the drive module in the selected bank until all data from the selected bank has been played (or a set play length is reached). Then, playback ceases on the selected bank. Your application may then select the other bank to continue playback. Playback continues on that bank until all data has been played. Playback can continue in this fashion, with data retrieval alternating between the two banks, until all data is played, a play length is reached, or `XLRStop` is called.

Labeling Drive Modules

By default, drives modules are not labeled. You can use the `XLRSetLabel` function to label idle drive modules with a null terminated string. When in non-bank mode, the label applies to the entire collection of drives that are mounted. When in bank mode, the label applies only to the drive module in the selected bank. For example, to label the drives in both banks you would:

1. Call `XLRSelectBank` with *`bankID`* set to `BANK_A`.
2. Call `XLRSetLabel` with the desired label for Bank A.
3. Call `XLRSelectBank` with *`bankID`* set to `BANK_B`.
4. Call `XLRSetLabel` with the desired label for Bank B.

The label can be up to `XLR_DRIVEMODULE_LABEL_LENGTH` bytes long.

Because the label is a NULL terminated string, you can "remove" a label from a drive module by calling `XLRSetLabel` with the desired label set to a null string.

You can call `XLRSetLabel` on an idle StreamStor to add or modify a label any time after a drive module has been selected. Labels need not be unique, i.e., the drive module in Bank A and the drive module in Bank B can have the same label.

To retrieve a label from a drive module, select the desired bank and then call `XLRGetLabel`. The buffer you read the label into must be large enough to hold the label and the NULL terminator.

Writing a User Directory

A user directory is a reserved area on a StreamStor recording that can only be written to by calling the function `XLRSetUserDir`. The user directory can be any type of data, including binary data. The user directory can be up to `XLR_MAX_UDIR_LENGTH` bytes long. When in non-bank mode, there is only one user directory. When in bank mode, each drive module can have a user directory.

Because of firmware limitations, the size of the user directory must be a multiple of eight bytes.

Writing a user directory on a bank is similar to writing a label on a bank. You first select the bank by calling `XLRSelectBank` and then call `XLRSetUserDir` to write the directory to it.

Since user directories are variable length and may or may not be NULL terminated, you must first get the user directory length before retrieving the user directory. For example, to get the user directory on Bank B, you would:

1. Call `XLRSelectBank` with *`bankID`* set to `BANK_B`.
2. Call `XLRGetUserDirLength` to get the length of the user directory on Bank B.
3. Call `XLRGetUserDir` to retrieve the user directory on Bank B, passing it the length that was returned by the call to `XLRGetUserDirLength`.

The Length of Drive Modules

The amount of data recorded on a drive module can be obtained by using any of the following API functions:

- `XLRGetLength` - the length is returned as a function value.
- `XLRGetDirectory` - the length is returned in the Length member of the `S_DIR` structure.
- `XLRGetBankStatus` - the length is returned in the `Length` member of the `S_BANKSTATUS` structure.

All three of the above functions are bank aware, which means that the length returned is the length on the selected bank.

Note that regardless of the bank mode, the length returned by the functions does not include the size of the label (if any) or the size of the user directory (if any).

Write Protecting Drive Modules

By default, drive modules are not write protected. You can use the `XLRSetWriteProtect` function to write protect idle drive modules. When in non-bank mode, the write protection applies to the entire collection of drives that are mounted. When in bank mode, the write protection applies only to the drive module in the selected bank.

The write protection remains from recording session to recording session, even if the drive module has been removed from the system and then reinserted. Write protection can be removed by calling the function `XLRClearWriteProtect`. When in non-bank mode, the write protection will be cleared from the entire collection of drives that are mounted. When in bank mode, the write protection is cleared only from the drive module in the selected bank.

Note that the write protection is only recognized by StreamStor recorders - it is not recognized by other systems.

Erasing Drive Modules

The command `XLRErase` is used to erase data on the StreamStor recorder. When in non-banking mode, the entire collection of drives is erased. When in bank mode, the erasure applies only to the drive module in the selected bank. The drives must be idle in order to perform the erase.

There are several options to `XLRErase`. When in bank mode, note that:

- If the drive module in the selected bank is write protected, `XLRErase` will have no effect on it.

- If the `XLRErase` option `SS_OVERWRITE_DIRECTORY` is used, the user directory on the selected drive module will be deleted. The label on the selected drive module will be replaced with the default label.

Getting Bank Status

Use `XLRGetBankStatus` to get the status of selected bank. This function will return the selected bank's status in a structure of type `S_BANKSTATUS`. For details on this structure, see the structure definition at the end of the Function Reference section of this manual.

Replacing a Drive Module

As drive modules fill up during a recording, you may want to replace the full modules with empty (or otherwise writable) drive modules. This becomes necessary, for instance, when Bank A becomes full, recording is in progress

on Bank B and a switch will be done back to Bank A. In this case, follow these steps to replace the full drive module with a new module:

1. Power off the bank containing the full drive module. When successfully powered down, all lights on the bank will go off.

2. Once the lights are off, remove the full drive module from the bank.

3. Put the new (write-enabled) drive module in the empty bank.

4. Power up the bank containing the new drive module. On power up, StreamStor will initialize the drive module. When the initialization has completed the READY light on the bank will light.

The same procedure can be applied to playback. If a recording spans more than two drive modules, when playback switches to the second drive module, once all data on the first drive module has been played, you can replace it with the third module in the set, etc.

You can also use the functions `XLRMountBank` and `XLRDismountBank` to mount and dismount banks.

Chapter 8

Drive Partitioning

Drive Partitioning

Partitioning allows you to logically divide the StreamStor drives into isolated sections. The partitioning feature is included on a variety of StreamStor controllers. The hardware manual for your StreamStor controller or daughterboard will indicate if partitioning is supported on it and will provide any other model-specific details.

The StreamStor can operate with or without partitions. By default, the drives are not partitioned. When the drives are not partitioned, they are operated upon as if they were a single unit. If the drives are partitioned, StreamStor operations are performed on the currently selected partition.

Creating a Partition

A StreamStor device is classified as not partitioned or partitioned. A partitioned system has one or more partitions and an undefined area that has not yet been partitioned. The API function `XLRPartitionCreate` is used to create a partition. You pass the function the length of the partition you want to create. The length must be a multiple of a page size, where a page is 4096 bytes. The function will attempt to create a partition of approximately the requested size. The actual size of the partition that is created is determined by the state of the disks and other internal boundary restrictions. The API function `XLRGetPartitionInfo` will return the actual size of the partition that was created in the `PartitionCapacity` member of the `S_PARTITIONINFO` structure.

When a partition is created, it is assigned a partition number. Partition numbers start at 0 (zero). The maximum number of allowed partitions may be hardware specific and is defined by the constant `XLR_MAX_PARTITIONS`.

If the StreamStor has data recorded on it that was written in a non-partitioned mode, that data must first be erased before you can create any partitions.

Note that you cannot create a partition at a specific offset on the StreamStor device. Instead, the StreamStor will determine where to create the partition.

Once a device has been partitioned, in order to perform operations on the partitions, you must first select the partition, using the partition number that was assigned to it when it was created.

Selecting a Partition

The API function `XLRPartitionSelect` is used to select the partition that is to be used for subsequent StreamStor partition-specific operations.

To select a partition, call `XLRPartitionSelect`, specifying the partition number that was assigned to it when it was created.

Some examples of partition-specific operations are:

- `XLRRecord` records only on the selected partition.
- `XLRSetWriteProtect` applies write protection only to the selected partition.
- `XLRErase` with any option (other than the option to destroy all partitions), erases the data only in the selected partition.
- `XLRGetDirectory` returns information that pertains only to the selected partition. For instance, the Full structure member of `S_DIR` will be set to `TRUE` if the selected partition is full.
- `XLRSetLabel` applies the requested label only to the selected partition.

If a partition has not been selected by calling `XLRPartitionSelect`, then partition 0 (zero) will be selected by default.

See the section "Bank Mode and Partitioning" in this chapter for details on how the selected bank works in conjunction with the selected partition.

Getting Partition Information

The API function `XLRGetPartitionInfo` is used to retrieve information from the StreamStor about the currently selected partition. To obtain this information, you pass `XLRGetPartitionInfo` a pointer to a structure of type `S_PARTITIONINFO`. The structure is returned, populated with the total number of partitions on the StreamStor, the partition number of the selected partition, the capacity of the selected partition, etc. Full details of the `S_PARTITIONINFO` structure can be found at the end of the Function Reference chapter.

To determine the amount of data that has been recorded in a partition, you first select the partition of interest. Next, call `XLRGetDirectory` (if the device is idle) or `XLRGetLength` (if the device is not idle).

Deleting a Partition

Partition deletion is supported only on the Amazon board types. For details, see the `XLRPartitionDelete` description in the Function Reference chapter of this manual.

Bank Mode and Partitioning

Partitioning can be used in conjunction with bank mode. When in bank mode, you can optionally partition one or both modules. In practice, when you are using bank mode, you will probably always want to partition both modules.

If in bank mode, the selected bank and the selected partition are used to determine where subsequent StreamStor operations are performed. For example, if you call `XLRSelectBank` to select Bank B, then call `XLRPartitionSelect` to select partition six, if you should then call `XLRSetLabel`, that label will be applied only to partition six on Bank B.

Note that if you are in bank mode and are using partitions, the bank mode `SS_BANK_AUTO_ON_FULL` is not allowed. The only way to begin operations on a different bank is to explicitly select the bank to be used.

Recording using Partitions

If a StreamStor has any partitions on it, subsequent operations on the StreamStor will be partition-specific. The following example illustrates this.

Assume you had previously created several partitions and that partition three had been created as one megabyte long. If you then select partition three with `XLRPartitionSelect`, and then begin a recording with wrap mode disabled, that data will be recorded only in partition three. When one megabyte has been recorded, that partition is "full" and recording will cease. To begin recording on partition number four, you must call `XLRPartitionSelect` to select it.

Using the same partitioning described above, if recording was started with wrap mode enabled, then once partition three had been recorded to its capacity, the recording would "wrap around" and start recording over the previously written data in partition three.

In a similar fashion, to play back recorded data you must first select the partition to be played.

Wrap Mode

You can record to partitions using non-wrap mode or using wrap mode. You can also append data to partitions using `XLRAppend`.

If a partition was recorded in wrap mode, you can append to it if it has not yet wrapped around. If the partition was recorded in wrap mode and the data has wrapped, you cannot append to it.

Removing Partitioning

Once a system has been partitioned, it remains partitioned until the system is erased using the `XLRErase` function with the `SS_OVERWRITE_PARTITION` option set. `XLRErase` will erase all data, partitions, user directories and labels. If in bank mode, the erasure will be applied to the currently selected bank.

Reusing Partitions

You can delete the data within the partition. You can then reuse the partition. To accomplish this, first call `XLRPartitionSelect` to select the partition that contains the data to be deleted. Then call `XLRErase` with the `SS_OVERWRITE_NONE` option.

Resizing Partitions

Partition resizing is supported only on the Amazon board types.

User Directories and Partitions

User directories can be created as follows:

- If the StreamStor device is not partitioned and is not in bank mode, then you can only create one user directory.
- If the StreamStor device is not partitioned and is in bank mode, then you can create a user directory for each bank.
- If the StreamStor device is partitioned, then you can create a user directory for each partition.

These user directory functions are partition-specific:

- `XLRSetUserDirectory`
- `XLRGetUserDirLength`
- `XLRGetUserDir`

So, you must first select the partition to be operated upon and then call the user directory functions.

Examples

The StreamStor SDK's example directory has several examples that demonstrate how to create partitions.

Chapter 9
Forking and Passthru

Forking and Passthru

StreamStor cards have the capability of real time “passing” and “forking” of data streams.

Overview

Data “forking” is the simultaneous recording and output of data. This is a real time operation which allows for the manipulation of data as well as recording of that same data stream simultaneously. Forking requires input from one source (PCI bus, top or front FPDP connector) and a *different* output channel. For example, data could be received over the front FPDP port, recorded to disk, and sent out the top FPDP port.

“Passthru” is the input of data over 1 channel (PCI bus, top or front FPDP port) and the simultaneous output (*over a different channel)* of that data *without* being recorded to disk. When StreamStor is operating in passthru mode, no disk drives are required; any drives that are connected are ignored by StreamStor.

☞ **NOTE:** **Both passthru and fork modes are REALTIME ONLY. Thus, the data must go out at the same speed as it is coming in. If not, an overflow condition will be signaled (see Overflow section below) and the data order OF THE OUTPUT STREAM can no longer be guaranteed. However, in forking mode - even an overflow condition - WILL NOT JEPORDIZE THE DISK RECORDING in any way.**

Forking

Forking is used in situations where the data must be used in real-time and recorded to disk simultaneously. Forking mode is set by a call to `XLRSetMode` using the `SS_MODE_FORK` parameter for the mode. Input and output streams are set by calls to `XLRBindInputChannel` and `XLRBindOutputChannel`.

☞ ***NOTE:*** ***The bound input and output channels must be different.***

A call to either `XLRRecord` or `XLRAppend` will start the data flow. Make sure that the FPDP port(s) are configured before record/append is called.

Passthru

Data "passthru" is the simultaneous input and real-time output of data. Passthru is used in situations where the data must be used in real-time and recording that data is not necessary or desired. Passthru is configured the same way forking is except that `XLRSetMode` is passed the `SS_MODE_PASSTHRU` parameter. `XLRRecord` is called to start data flowing even though no disk recording takes place.

Output over the PCI bus

Using the PCI bus as an output channel differs from single channel reads in that calls to `XLRReadFIFO` are required. `XLRReadFIFO` retrieves data from StreamStor to the user provided buffer (similar in operation to `XLRReadData`). `XLRReadFIFO` first ensures that the amount of data available in StreamStor's FIFO is greater than or equal to the amount of data requested. If there is not enough data present, `XLRReadFIFO` will wait up to 5 seconds for enough data to complete the request. Should not enough data be present after 5 seconds, `XLRReadFIFO` will return status `XLR_FAIL`. Subsequent calls to `XLRGetLastError` and `XLRGetErrorMessage` will yield a "no data" error.

☞ ***NOTE: A "no data" error does not necessarily mean that there are 0 bytes to be read, only that there are fewer bytes than the requested size.***

Checking the FIFO length

The StreamStor SDK provides the `XLRGetFIFOLength` function to provide the real time ability to check the amount of data that is available for output. This function returns a 64 bit integer that is the number of bytes available for reading at that time. This function is provided for informational purposes and is primarily used in the situation where input data flow is slow enough that the `XLRReadFIFO` timeout of 5 seconds is not adequate. In that case, user applications can make calls to `XLRGetFIFOLength` to ensure there is enough data present before the call `XLRReadFIFO`.

Ending a FIFO operation

Stopping data forking or passthru requires the use of **two** calls to `XLRStop`. The first `XLRStop` will shutdown the receiving hardware, but leave the sending operation (over the PCI bus) still running. After the first stop, call `XLRGetFIFOLength` to find out exactly how much data is left in the FIFO to read. Next, call `XLRReadFIFO` (with the amount returned from `XLRGetFIFOLength` – **make sure the buffer is big enough**) to read out the remaining data. Note that after XLRStop is called, you are only allowed

to call `XLRReadFifo` once. Finally, call `XLRStop` for the final time to take the StreamStor out of record mode.

For an example of how to use FIFOs please see the example `XLRGetFIFOLengthExample.c` in the SDK example directory.

Overflows

Data forking and passthru operate in a real time fashion. If data is coming in faster than it is leaving, StreamStor's on board RAM buffer will eventually fill and an overflow condition will arise. Overflow conditions are signaled by the Overflow member of the `S_DEVSTATUS` structure. This structure is filled by calls to `XLRGetDeviceStatus`. See the function reference for more information.

☠ **CAUTION:** ***Once an overflow condition arises, the integrity and order of output data can no longer be guaranteed. The only way to "recover" from an overflow situation is to stop and restart StreamStor.***

☞ NOTE: *In forking mode, the recording to disk will continue accurately and uninterrupted – only the order of the output data stream will be inaccurate.*

Chapter 10
Technical Support

(303) 485-2721

support@conduant.com
www.conduant.com/support

Technical Support

Conduant wants to be sure that your StreamStor system works correctly and stays working correctly. In the unlikely event, however, that you are unable to get your new system to work properly, or if a working system ceases to function, we will do all that we can to get your system back online.

Solving the problem is largely a matter of data collection and steps that must be taken one at a time. In order for us to better serve you, we ask that you take the time to perform the following steps prior to calling us. This way, you can provide us with the most meaningful information possible that will help us solve the problem.

Is the problem one that obviously requires replacement parts due to physical damage to the system? If yes, then please gather the information described below and report the problem to tech support, by phone or through the Conduant web site.

Have you confirmed that no cabling has been inadvertently disconnected or damaged while working around the equipment?

Is the card properly seated in the PCI slot?

Do all the disk drives have good power connections and voltages?

Does the confidence test sscfg.exe (on Windows) or ssopen/sstest (on Linux) run OK?

Has the software installation been corrupted? Try re-installing software.

Have you checked the Conduant web site for technical bulletins?

Have you recently installed a new Linux kernel or compiler or a new Windows Service Pack?

If the above steps did not resolve the problem, then please call Technical Support or open a support ticket. To open a support ticket, go to www.conduant.com, click on “Support” and then click on “Submit a ticket.”

Please provide the following information:

- StreamStor Card Serial Number
- Software Revision(s)
- Configuration (816XF, 816XF2, disk drive model numbers, etc.)

- Description of third party equipment that StreamStor is working with (i.e. Manufacturer and model numbers, etc.)
- Description of third party software being used with StreamStor
- Computer model and type (Pentium, Pentium II, etc.)
- Operating system version.

We will do all that we can to resolve the problem as quickly as possible.

Contacting Technical Support

E-mail: support@conduant.com

Phone: (303) 485-2721

Fax: (303) 485-5104

Web: www.conduant.com

Mail: Conduant Corporation

Technical Support

1501 South Sunset Street, Suite C

Longmont, CO 80501

Appendix A – Error Codes

If you are experiencing one of these errors and are unable to determine the cause, please contact Conduant technical support for assistance.

Number	Error Title	Description
2	XLR_ERR_NODEVICE	StreamStor device was not found in system.
3	XLR_ERR_NOINFO	Undefined error occurred.
4	XLR_ERR_WDOPEN	Cannot open device driver.
5	XLR_ERR_SYSERROR	The controller reported a system error.
6	XLR_ERR_NOXLR	No StreamStor cards located.
7	XLR_ERR_INVALID_CMD	An invalid command was received by the controller.
8	XLR_ERR_HANDLE	Invalid handle.
9	XLR_ERR_DMAREADFAIL	A DMA read failure occurred.
10	XLR_ERR_SYSTATUS	Request is incompatible with current system status.
11	XLR_ERR_NOCMDSTATUS	The command did not complete. Communication with controller timed out.
12	XLR_ERR_DMAINCOMPLETE	The data transfer timed out and did not complete.
13	XLR_ERR_APPSTART	The controller failed to initialize RAM application.
14	XLR_ERR_OUTOFMEMORY	The DLL failed to allocate sufficient memory.
15	XLR_ERR_WIN32FAIL	A Win32 API failure occurred.
16	XLR_ERR_WRITENOTACTIVE	System not ready to receive data.
17	XLR_ERR_WDVERSION	Incorrect driver version detected.
18	XLR_ERR_OPENHANDLE	Device reference by handle already opened.
19	XLR_ERR_INVALIDINDEX	Invalid card index value.
20	XLR_ERR_DEVICELOCK	Could not lock device for exclusive access.
21	XLR_ERR_DETECTCARD	Card configuration invalid.
22	XLR_ERR_BUFLOCK	Could not lock user memory buffer.
23	XLR_ERR_READFAIL	Data read error.

24	XLR_ERR_WRITERAM	Firmware write to device memory failed.
101	XLR_ERR_INVALID_LENGTH	An invalid or unaligned transfer length was requested (must be 64 bit aligned).
102	XLR_ERR_SYSBUSY	System is busy. Use XLRStop to before sending other commands.
103	XLR_ERR_CMDFAIL	The controller has failed to execute the command.
104	XLR_ERR_FILENOTFOUND	A required file was not found.
105	XLR_ERR_LOADKEY	A required registry key was not found.
106	XLR_ERR_DLDCHECKSUM	A required file is corrupted or upload failed.
107	XLR_ERR_DRVFAIL	A disk drive is failing to respond.
108	XLR_ERR_NODRIVER	Device driver not found or device already open.
109	XLR_ERR_FIFO_INACTIVE	Invalid command, FIFO inactive.
110	XLR_ERR_INVALIDVR	An unconfigured or invalid VR was selected.
111	XLR_ERR_NOTENABLED	Optional feature not enabled.
112	XLR_ERR_OUTOFRANGE	Request was not in the recorded data range.
113	XLR_ERR_NOTINFIFO	Command valid only in FIFO mode.
114	XLR_ERR_KERNELMEM	Unable to allocate kernel memory.
115	XLR_ERR_INTENABLE	Unable install device interrupt.
116	XLR_ERR_READCOLLISION	Attempt to start multiple reads from single thread.
117	XLR_ERR_READIDLE	Attempted to check status on non-existent read request.
118	XLR_ERR_FIFODRIVES	Current drive configuration incompatible with FIFO mode.
119	XLR_ERR_FWVERSION	Hardware firmware incompatible with API version.
120	XLR_ERR_OSFAIL	A system call failed.
121	XLR_ERR_THREADCREATE	Process thread creation failed.
122	XLR_ERR_EXPECTEDDISKS_ MATCH	The number of expected disks doesn’t equal the actual number of disks.
123	XLR_ERR_BOARDTYPE	Unknown board type found.
124	XLR_ERR_FULL	Insufficient disk space.
127	XLR_ERR_INVOPT	Invalid option value.
142	XLR_ERR_INVALID_ PORTMODE	Port in wrong mode for this operation.
143	XLR_ERR_NOAPPEND	Attempt to delete non-existent append.
144	XLR_ERR_EMPTY	No data.
145	XLR_ERR_INVALID_BANK	Invalid bank name specified.
146	XLR_ERR_NOTINBANKMODE	Command only valid in bank mode.

148	XLR_ERR_DRIVEMODULE_ NOTREADY	Drive module is not ready.
153	XLR_ERR_CANNOT_RECOVER _DATA	No recovery of data possible.
154	XLR_ERR_NO_RECOVERABLE _DATA	No recoverable data.
155	XLR_ERR_BAD_DISKSET	A disk is missing from a recording or a disk is mounted that was not part of the set when the recording was originally made.
156	XLR_ERR_INVALID_PLAY _LENGTH	Playback length is beyond the end of the recording or is not aligned on an eight-byte boundary.
157	XLR_ERR_INVALID_ WDLICENSE	Invalid driver license.
158	XLR_ERR_WRITE_ PROTECTED	Command invalid on write protected drive modules.
159	XLR_ERR_MAX_CARDS	Maximum number of StreamStor cards exceeded.
160	XLR_ERR_DRVFAIL_BUS0_ MASTER	Master drive on Bus 0 missing or failing.
161	XLR_ERR_DRVFAIL_BUS0_ SLAVE	Slave drive on Bus 0 missing or failing.
162	XLR_ERR_DRVFAIL_BUS1_ MASTER	Master drive on Bus 1 missing or failing.
163	XLR_ERR_DRVFAIL_BUS1_ SLAVE	Slave drive on Bus 1 missing or failing.
164	XLR_ERR_DRVFAIL_BUS2_ MASTER	Master drive on Bus 2 missing or failing.
165	XLR_ERR_DRVFAIL_BUS2_ SLAVE	Slave drive on Bus 2 missing or failing.
166	XLR_ERR_DRVFAIL_BUS3_ MASTER	Master drive on Bus 3 missing or failing.
167	XLR_ERR_DRVFAIL_BUS3_ SLAVE	Slave drive on Bus 3 missing or failing.
168	XLR_ERR_DRVFAIL_BUS4_ MASTER	Master drive on Bus 4 missing or failing.
169	XLR_ERR_DRVFAIL_BUS4_ SLAVE	Slave drive on Bus 4 missing or failing.
170	XLR_ERR_DRVFAIL_BUS5_ MASTER	Master drive on Bus 5 missing or failing.
171	XLR_ERR_DRVFAIL_BUS5_ SLAVE	Slave drive on Bus 5 missing or failing.
172	XLR_ERR_DRVFAIL_BUS6_ MASTER	Master drive on Bus 6 missing or failing.
173	XLR_ERR_DRVFAIL_BUS6_ SLAVE	Slave drive on Bus 6 missing or failing.
174	XLR_ERR_DRVFAIL_BUS7_ MASTER	Master drive on Bus 7 missing or failing.
175	XLR_ERR_DRVFAIL_BUS7_	Slave drive on Bus 7 missing or

	SLAVE	failing.
176	XLR_ERR_NOTIN_RECMODE	Command only valid when in record mode.
177	XLR_ERR_EXT_TO_PCI_ OVERFLOW	External port to PCI overflow.
178	XLR_ERR_INVALID_ INTERFACE	Command is not available for the currently in use interface (PCI bus, Ethernet, or Serial port).
179	XLR_ERR_INVALID_RETURN _FORMAT	Data returned from command is formatted incorrectly (Ethernet and Serial port interfaces only).
180	XLR_ERR_INVALID_ CHANNEL	The channel being selected or bound is invalid.
181	XLR_ERR_INVALID_OP_ON_ CHANNEL	Operation is not permitted on this channel.
182	XLR_ERR_USE_SELECT_ CHANNEL	SS_OPT_FPDPEXTCONN is no longer valid for selecting the front FPDP port. XLRSelectChannel must be used.
183	XLR_ERR_INVALID_SYSTEM _MODE	Requested mode is invalid.
184	XLR_ERR_TOO_MANY_ CHANNELS	Only 1 input or output channel is allowed in this mode.
185	XLR_ERR_NO_INPUT_ CHANNELS	Must have at least 1 input channel.
186	XLR_ERR_NO_OUTPUT_ CHANNELS	Must have at least 1 output channel.
187	XLR_ERR_NOT_VALID_IN_ MULTI	Operation not valid in mutli-channel mode.
188	XLR_ERR_PARTITION_SIZE	Partition size must be multiple of page size.
189	XLR_ERR_INVALID_ PARTITION	Invalid partition.
190	XLR_ERR_TOO_MANY_ PARTITIONS	Only 256 partitions are permitted.
191	XLR_ERR_NOT_EMPTY	System must be empty for this command.
192	XLR_ERR_UNKNOWN_DIR_ VERSION	The directory version found is newer than the current firmware can handle.
193	XLR_ERR_DATA_INTEGRITY	Data integrity check failed.
194	XLR_ERR_HWVERSION	XBAR version incompatible with Firmware version.
195	XLR_ERR_ARRAY_TOO_SMALL	User supplied array is too small.
196	XLR_ERR_READFAIL_FORK	Read failure during fork.
197	XLR_ERR_INVALID_ ALIGNMENT	Offset or transfer length of read request is not aligned on the required 4 or 8 byte boundary.
198	XLR_ERR_CMD_DRIVE_ERROR	Invalid command for this drive.
199	XLR_ERR_INVALID_MAPOPT	SS_OPT_FSMAPPED option invalid for wrap mode.

200	XLR_ERR_NOT_LAST_ PARTITION	Command valid only on the last partition on the device.
201	XLR_ERR_RESIZE_EXCEEDS_ DEVCAP	Resize value requested exceeds the device capacity.
202	XLR_ERR_RESIZE_ZERO_ INVALID	Resize value of zero is invalid on an empty partition.
203	XLR_ERR_RESIZE_IN_DATA_ RANGE	Resize value requested is within the recorded data range.
204	XLR_ERR_NOT_PARTITIONED	Command invalid on unpartitioned device.
205	XLR_ERR_REMOTEVERSION	Remote Protocol Version is not supported.
300	XLR_ERR_PORT_NOT_FOUND	Port is unavailable (Serial/Ethernet interfaces only).
301	XLR_ERR_PORT_ACCESS_ DENIED	Port access is denied (Serial/Ethernet interfaces only).
302	XLR_ERR_PORT_TIMEOUT	Port operation has timed out.
303	XLR_ERR_CONNECT_ REFUSED	Connection refused by target.
304	XLR_ERR_IPADDRCONVERT_ FAILED	IP address conversion failed.
305	XLR_ERR_EACCESS	Permission denied.
306	XLR_ERR_SOCKET_ EAFNOSUPPORT	Address family not supported.
307	XLR_ERR_SOCKET_EINVAL	Unknown protocol or family.
308	XLR_ERR_EMFILE	Process file table overflow.
309	XLR_ERR_SOCKET_ENOBUFS	Insufficient buffer memory.
310	XLR_ERR_SOCKET_ENOMEM	Insufficient memory.
311	XLR_ERR_SOCKET_ EPROTONOSUPPORT	Protocol type not supported.
312	XLR_ERR_SOCKET_CREATE_ FAILED	Cannot create socket.
313	XLR_ERR_SOCKET_CONNECT _FAILED	Cannot connect to socket.
314	XLR_ERR_SOCKET_EADDRIN USE	Local address already in use.
315	XLR_ERR_SOCKET_EAGAIN	No more free local ports.
316	XLR_ERR_SOCKET_ EALREADY	Connection attempt incomplete.
317	XLR_ERR_EBADF	Bad File descriptor.
318	XLR_ERR_SOCKET_EFAULT	Invalid socket address.
319	XLR_ERR_SOCKET_ EINPROGRESS	Socket connection in progress.
320	XLR_ERR_EINTR	Interrupted system call.
321	XLR_ERR_SOCKET_EISCONN	Socket already connected.
322	XLR_ERR_ENETUNREACH	Network is unreachable.
323	XLR_ERR_SOCKET_ ENOTSOCK	File descriptor not associated with a socket.
324	XLR_ERR_ENFILE	Open file limit reached.
325	XLR_ERR_REMOTEPROTOCOL	Remote protocol error.

326	XLR_ERR_REMOTEINIT	Cannot initiate communication with remote device.
327	XLR_ERR_SOCKET_RECV_MSG	Socket receive message error.
401	XLR_ERR_DMA_TIMEOUT	DMA timeout.
402	XLR_ERR_UINT_ATTN	Unit attention reset detected.
403	XLR_ERR_BLOCK_QUEUE	Error in buffer queue.

End of Document