StreamStor Real-Time Storage Controller

Installation and User's Guide

Copyright and Trademarks

Printed in the United States.

Publication date: November 15, 2002

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.

About This Manual

This manual is intended serve the following purposes:

* to provide an overview of StreamStor Real-time Storage

* to act as a guide for hardware installation

* to act as a reference for the programmer

It is suggested that you periodically check the Conduant web site for the most recent software updates, application notes, and technical bulletins.

If you are unable to locate the information you need, please feel free to contact us by e-mail or phone.

Chapter

1

Introduction

About the StreamStor System

Thank you for purchasing a Conduant StreamStor Real-Time Storage System. Your StreamStor system is a disk-based, real-time recording system for PCI bus computers. The StreamStor system consists of controller card that plugs into the PCI bus, high performance disk drives, device drivers, software development tools, and additional utility software.

The PCI bus is a high performance I/O bus designed for attaching peripheral devices to computer systems. It is found in computing systems from many different manufacturers and is supported by nearly all major operating systems. By utilizing the PCI bus instead of a proprietary bus interface, StreamStor provides an open platform recording system. PCI data acquisition cards (digital oscilloscopes, frame grabbers, telemetry interfaces, etc) are available from many manufacturers to collect data and record it to system memory in real time (as it is collected). StreamStor provides a large capacity and cost effective alternative to system memory for these applications.

The StreamStor Storage System is able to receive data over the PCI bus directly from the data acquisition device at very high average (sustained) data rates. Virtually all of the available PCI cards that can record data to system memory are compatible with StreamStor. Only minor software modifications are generally required to redirect data to the StreamStor PCI card. This capability is often in the software provided by the manufacturers of data acquisition devices. Conduant maintains a list of tested configurations, please contact technical support for more information.

StreamStor was specifically designed to record sequential data without interruption at very high data rates. This is in contrast to traditional storage systems that are designed for data processing purposes and cannot sustain these high data rates. Unlike typical computer disk storage solutions that are designed for optimum performance doing random data reads and writes, StreamStor has been designed for optimum performance in sequential read and write operations. The StreamStor system has also been designed to operate without host computer intervention. This eliminates any bottlenecks or interruptions in the data stream due to heavy computer loads or delays.

The device drivers and API (Application Programming Interface) provide for a smooth integration of StreamStor with the data acquisition device and/or analysis software. Periodically check the Conduant web site for new releases of the software components. Please feel free to offer suggestions and request new features.

What you need to get started

To set up and use the StreamStor system, you will need the following:

* StreamStor controller card
* Disk drives
* Disk drive mounting brackets
* Disk drive interface cables
* StreamStor Software Development Kit
* A computer and chassis with sufficient space for mounting all disk drives.
* An empty full length PCI slot or 3U CompactPCI slot (depending on model).
* This manual

Software Programming Choices

The StreamStor Software Development Kit (SDK) includes a Windows DLL library, a Linux function library and drivers providing control and data retrieval functions necessary for using the StreamStor system. 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.

Unpacking

Carefully inspect all shipping packages for any sign of damage. In particular, look for wrinkled or bent corners, holes, or other signs of bad handling or abuse. If you notice any damage to the packaging, immediately open the boxes and inspect the contents for damage. Pay close attention to the components near the area where the packing material was damaged. Report any damage to the carrier and Conduant immediately.

Controller Board

The StreamStor controller board is shipped in a specially designed antistatic box to prevent electrostatic damage to the board. To avoid damage in handling the board, take the following precautions:

* Ground yourself with a grounding strap or grasp a conductive, grounded object to dissipate any static charge while handling the board.

* Always store the board in its antistatic box when not installed in a computer system.

* Inspect the board carefully before installing in the computer. Notify Conduant immediately if the board appears damaged. Do not install a damaged board into your computer.

* Never touch any exposed connector pins or component leads.

* Avoid bending or twisting the board.

Disk Drives

Hard disk drives such as those that may have been included with your system are very susceptible to excess shock and careless handling, it is recommended that you leave the drives in their packaging until you are ready to install them in the chassis. Please observe the following handling precautions:

* Allow the hard drive to reach room temperature BEFORE installing it. This may take several hours depending on shipping conditions.

* Handle the hard drive by the sides; DO NOT touch the printed circuit board.

* Do not drop, jar or bump the drive. Even setting the drive on a hard surface too roughly can damage the mechanical components inside the drive.

* Never disconnect/connect drive cables while power is on.

* Observe anti-static handling guidelines as outlined above for circuit board handling.

Chapter

2

Installation

Components

StreamStor systems generally consist of the following components:

- StreamStor Disk Controller Card (PCI-408, PCI-816, PCI-816XF, PCI-816XF2 or CPCI-408)
- In-System tested disk drives
- Disk Drive Mounting Hardware
- Disk Drive Cables
- User Manual
- Installation Software (CDROM)

☠ **CAUTION:** ***Please read the entire installation section before starting to install the StreamStor system. This manual assumes that the user is knowledgeable and comfortable with basic computer cabling, power connections, inserting cards into the PCI bus, and use of the computer operating system. If you are unsure as to how to proceed, please contact Conduant.***

Planning Your Installation

The StreamStor controller board uses anywhere from 2 to 8 flat cables (depending on the configuration) to connect the controller to up to 16 disk drives. Not all computer chassis layouts easily facilitate this type of cabling. The configuration and layout of the computer chassis will greatly affect the ease of installing the StreamStor system. Contact Conduant if you need help in choosing or designing an appropriate chassis. Extension chassis systems are also available to avoid impacting an existing computer system.

The 408 and 816 models can have one or two disk drives per cable. Each interface cable must have one MASTER drive (see Drive Configuration below). If there is a second drive on the same cable it must be configured as a SLAVE drive. If there is only a single drive on the interface cable, attach the drive to the end connector. Prior to mounting anything in the chassis, lay out the disk drives and the StreamStor card on a flat, static free surface and model the routing and placement of the cables. Pay close attention to the connector keys because these may define which way the disk drives must be mounted. Avoid pinching or routing over sharp edges to prevent cable damage.

The orientation of the disk drives can greatly affect the ease of cable routing. In horizontal orientations, mounting the drive with the board facing down prevents debris from inadvertently damaging or shorting the electronics and is the preferred orientation. Generally, all drives should be mounted in the same orientation to avoid twists in the cables.

☠ **CAUTION:** ***When removing cables from the StreamStor board, ALWAYS use the ejector tabs to gently free the cables from the board. NEVER pull on the cables to free them from the board.***

The cables supplied with your system are the maximum recommended length; longer cables may cause intermittent data loss and should be avoided. Removable drive carriers add several connections to the interface bus that can also cause intermittent data problems. Several removable drive carriers are known to work with StreamStor so please contact Conduant if you have special requirements for them.

The CompactPCI card (CPCI-408) can be configured to allow the drive cables to exit from either side of the card. To route the cable to the side opposite of the connector, you must remove the front faceplate. The cables can then be routed through the slot between the front of the card and the faceplate. Be careful when replacing the faceplate to avoid pinching the cables.

Hardware Installation

The StreamStor Storage System comes in 3 models: 408, 816, and 816XF. The first digit refers to the number of drive buses available on the controller card and the next two digits indicate the number of disk drives used. The following table shows the relationship of these models.

Model	Drive Busses	Drive Cables	Drives
408	4	4	8
816	8	8	16
816XF	8	8	16

All StreamStor models are upgradeable for increased storage capacity and high sustained data rates. Please contact Conduant for more information on upgrades.

Controller Card

The PCI versions of the StreamStor controller are full-length PCI cards that meet the PCI 2.1 specification (figure 1). Installation requires a PCI slot that can accommodate a full size card and has a card support guide. Clearance is also required for the drive cables exiting from the controller card. The CompactPCI versions of the StreamStor controller are standard 3U size cards that can be installed in any available CompactPCI slot. To install into a 6U slot you will need an adapter board. The drive cables can be routed to either side of the CompactPCI cards.

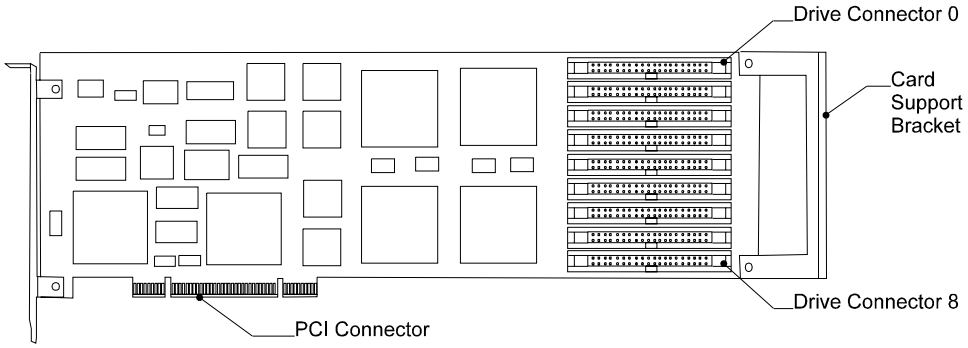


Figure 1 - StreamStor PCI Controller Card

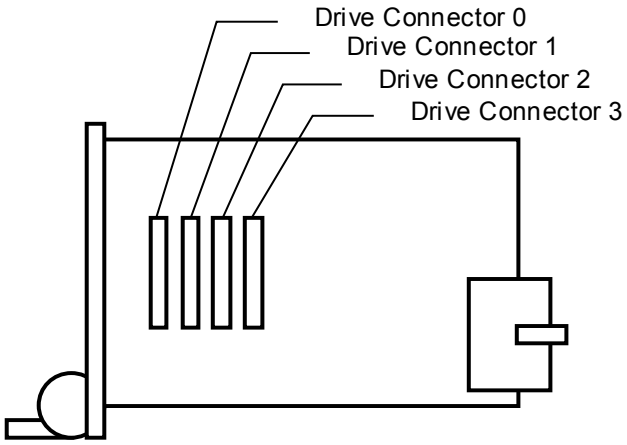


Figure 2 - StreamStor CPCI Card

Drive Number Assignments		
Connector	Master	Slave
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15
8	16	17

The following are general instructions for installing your StreamStor controller. You should also consult your computer user manual or technical reference for more specific instructions and warnings.

☠ **CAUTION:** ***Over flexing the circuit board will damage the controller.***

☞ **NOTE:** ***You may find it easier to attach the drive interface cables BEFORE installing the controller board. Be careful to prevent damage to any components on the back side of the circuit board if you lay the card down.***

1. Turn off and unplug your computer.
2. Remove the top cover or access port to the I/O bus.
3. Remove the expansion slot cover on the back panel of the computer for the slot into which you intend to install the StreamStor controller.
4. Insert the StreamStor controller board into the chosen PCI slot. Gently rock the board to ease it into place. It may be a tight fit but do not force the board into

place. Make sure that the card support bracket lines up correctly with the support provided in the computer chassis.

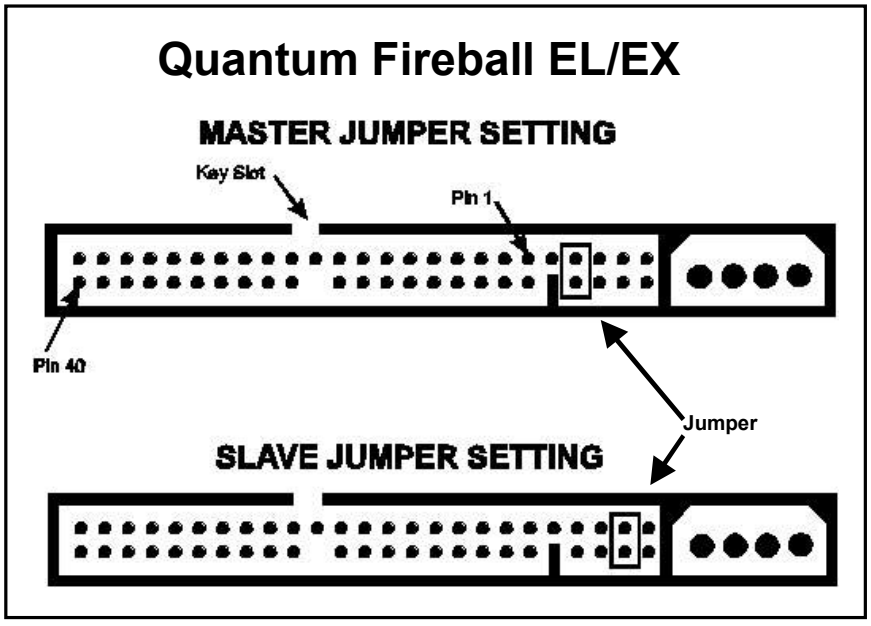
5. Screw the mounting bracket to the back panel of the computer chassis.

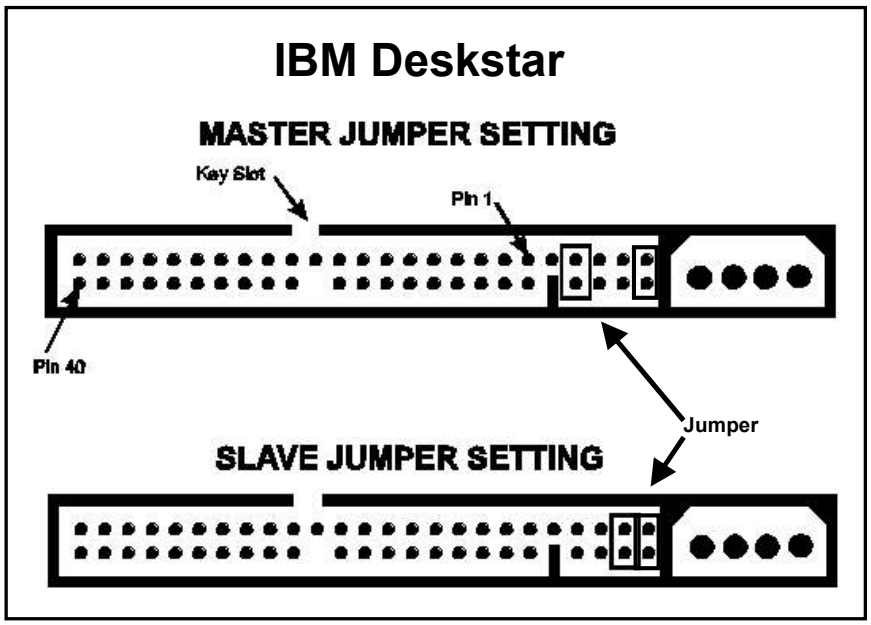
6. Proceed to drive installation and cabling.

Drive Configuration

StreamStor controllers can be configured with a second disk drive attached to each drive cable. Of those two disk drives, one must be configured as a MASTER and the other as a SLAVE. Configuring a drive as a MASTER or SLAVE is accomplished through the adjustment of jumpers on the disk drives. The exact jumper settings for MASTER and SLAVE modes may vary from one disk drive manufacturer to another. When adjusting jumpers, it is recommended that the user look at the jumper guide that is usually printed on the drive. If there is no such guide, the user should refer to other shipped material, the web site of the disk drive manufacturer, or contact Conduant for assistance.

The following illustrations show the jumper configurations for the Quantum Fireball EL/EX and IBM Deskstar hard drives. Please consult the documentation included with your system for jumper settings for other drive types.





Drives shipped from Conduant will be pre-configured with the correct number of MASTER and SLAVE drives for the shipped configuration. The drives are marked with stickers labeled MASTER or SLAVE.

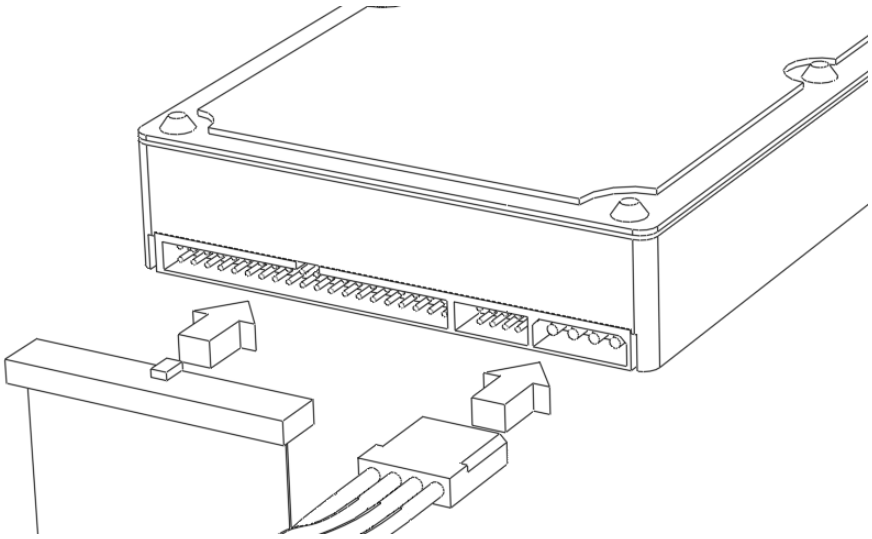
In 408, 816 and 816XF configurations, the StreamStor controller should be attached to a cable END. The drives may attach to the remaining connectors in any order. Be careful to match the connector keys when connecting the drive to the cable and never force the connection, as this could damage the disk drive.

Installing the Drives

The method used to mount the disk drives is left to you. However, the disk drives provided by Conduant require adapters for mounting these standard 3-1/2" disk drives into standard 5-1/4" slots. If you are mounting the drives into 3-1/2" slots these brackets are not necessary. Be sure to follow the handling precautions described above when installing the hard drives.

Connecting Interface and Power Cables

Depending on the orientation you chose for the disk drives, it may be easier to first attach the drive power cables or the drive interface cables. You should first attach whichever connectors are furthest from your reach. Be careful to support the StreamStor PCI card when plugging in the drive cables to avoid over flexing the circuit board. Also be careful to correctly orient the connector since they are all keyed to prevent incorrect insertion.



CAUTION: ***The signal and power connectors are keyed to prevent incorrect insertion. Exerting excessive force with the connectors improperly aligned can cause damage. Even with the correct alignment, care should be taken to not apply excessive force or torque.***

Carefully connect and route the interface cables from the StreamStor controller to the disk drives. Remember that either all cables must have a MASTER drive only or both a MASTER and SLAVE drive. There is no requirement to attach the cables to a particular connector on the controller board for a new installation but they must be used in order starting at connector zero.

Drive auto configuration

The StreamStor controller boards support auto configuration to the number of drives installed. The 408 boards (PCI-408 and CPCI-408) can support 2,3,4,6 and 8 drive configurations. The 816, 816XF and 816XF2 boards (PCI-816, PCI-816XF and PCI-816XF2) can support 2,3,4,5,6,7,8,10,12,14 and 16 drive configurations. If any drive cable has a slave drive than all cables must have a slave drive. Connectors on the controller board should be used in order without gaps starting from connector zero (0). You should always check that all drives are being recognized by the system using the `XLRGetDeviceInfo` function call. The sscfg.exe program also reports this information after initialization.

Installing the Software

Your StreamStor system was shipped with the Software Development Kit on CD-ROM. Please power up your computer and, when ready, run the setup.exe program on the CD-ROM to start the installation process on Windows systems.

Plug and play operating systems such as Windows 98 and Windows 2000 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.

Linux installation instructions are available in the linux sub-directory on the CD-ROM.

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. Also check the Conduant web site periodically for software updates. Software updates may include new features and capabilities as well as important fixes and improved hardware support. Users who do not have access to the Internet can request updates by calling Conduant Technical Support.

Chapter

3

Software Development Kit (SDK)

Introduction

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 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.

The following pages define the functions provided by the library for controlling, recording and retrieving data from the StreamStor system. It is suggested that you periodically check the Conduant Web Site for updates. If you do not have Internet access, feel free to call and ask for technical support. We'll be happy to send you the latest updates.

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 NT 4.0, Windows 2000, Windows XP and Linux operating systems. The drivers are installed automatically by the supplied setup program. The Windows NT/2000 device driver is named windrvr.sys. The Linux device driver is installed as a kernel module. Special instructions for Linux driver installation are included in the Linux sub-directory on the CD-ROM.

Support files

The StreamStor support files (`ssatap.bib, ssatac.bib, sspci.bib, ssatap3.bib` and `sspxf.bib`) located in the installation directory are required for proper initialization of the StreamStor system after power-on or reset. On Windows computers, the location of these files is defined by a registry entry created by the installation program that specifies the installation directory where these files are installed by default. This registry setting may be changed if these files are moved to an alternate directory. The registry path is:

"`HKEY_LOCAL_MACHINE\SOFTWARE\Conduant\StreamStor SDK\BibPath`"

In Linux, the environment variable `STREAMSTOR_BIB_PATH` is used to specify this directory path.

The Windows DLL for the StreamStor API is named `xlrapi.dll`. This file is installed into the main directory where StreamStor files are located. When

developing custom applications you must make sure this file is available in a directory where the operating system searches for DLL files. The Linux library is named `libssapi.a` and all functions are statically linked into the user application from this library archive.

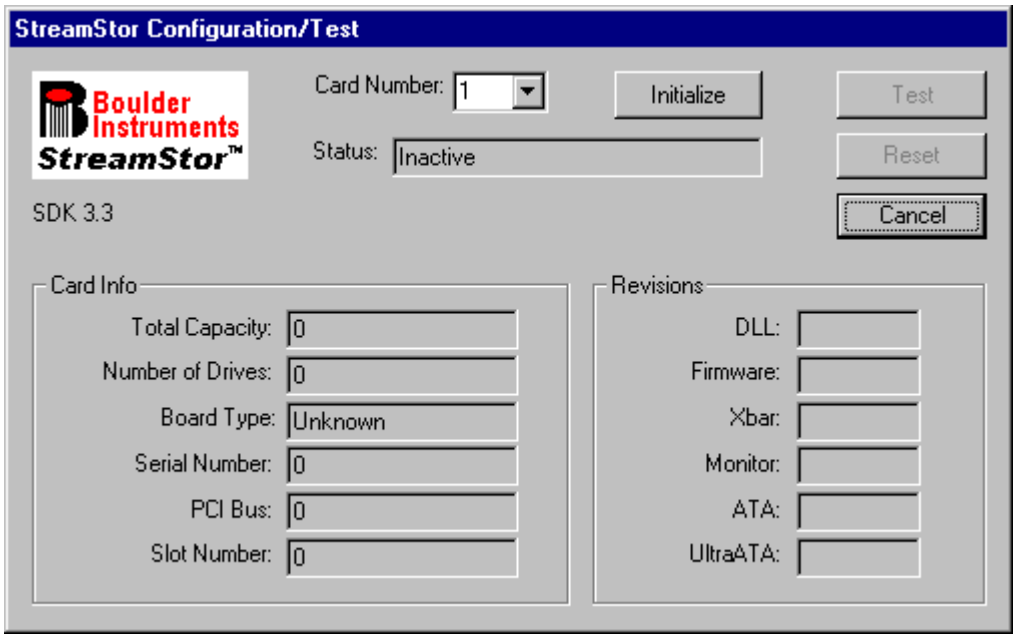
☠ **CAUTION:** ***Modifying the Windows registry incorrectly can irreparably damage your Windows installation.***

Windows Uninstall

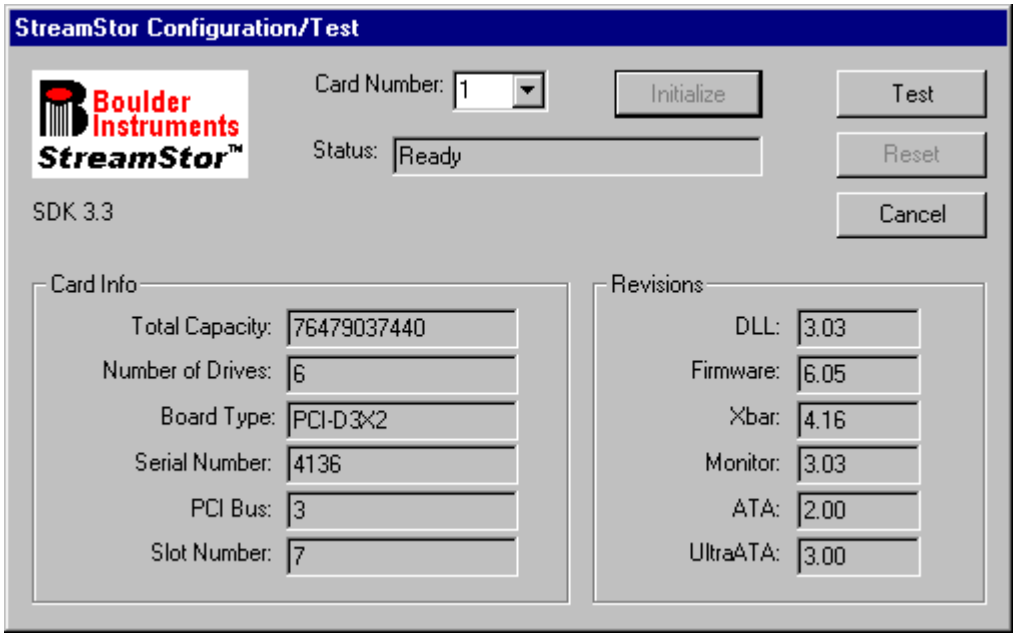
The StreamStor SDK can be easily uninstalled in Windows by using the "Add/Remove Software" wizard in the control panel. Simply select "StreamStor SDK" and all installed components will be automatically removed. You can also select "Remove StreamStor SDK" in the StreamStor menu.

Windows Configuration/Test Utility

The utility program `sscfg.exe` is included with the SDK for the purpose of testing the StreamStor system for proper configuration and functionality. If you have just received your StreamStor system or you are experiencing problems, running this program will perform a configuration and confidence test to insure that your system is working properly. The DLL `bisrun.dll` is a required component and 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 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.

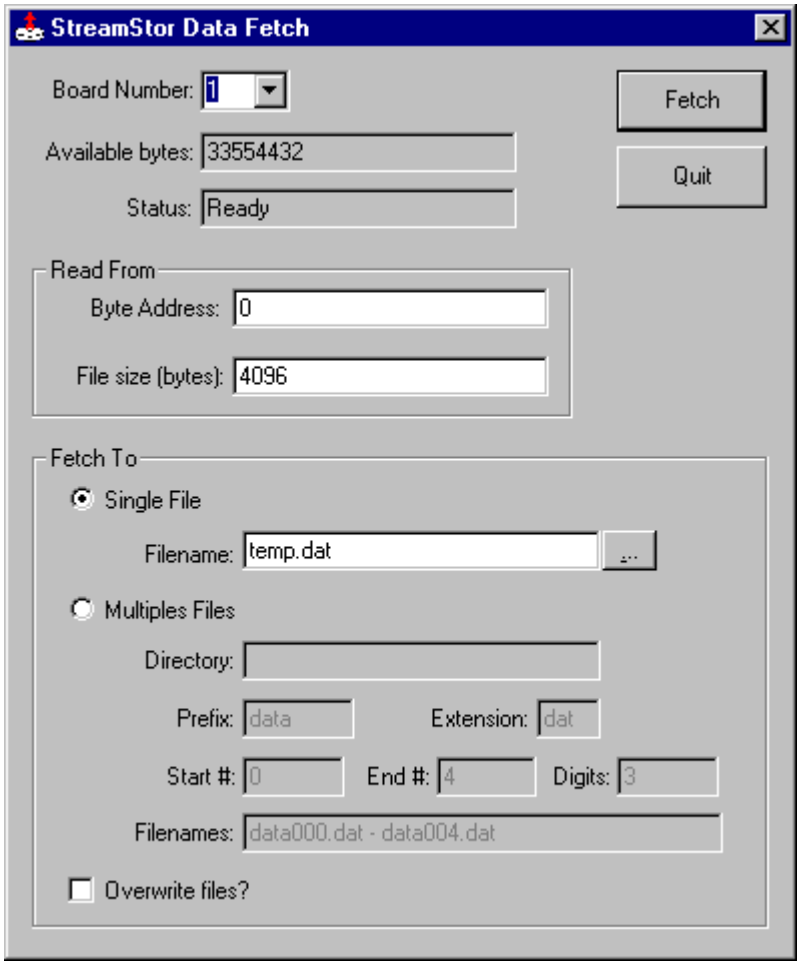
If the initialization has completed successfully you should check the information provided by `sscfg` to insure your system has been correctly identified according to your purchased model and configuration. If you discover any problems please contact Conduant. At this point you should press the Test 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.***

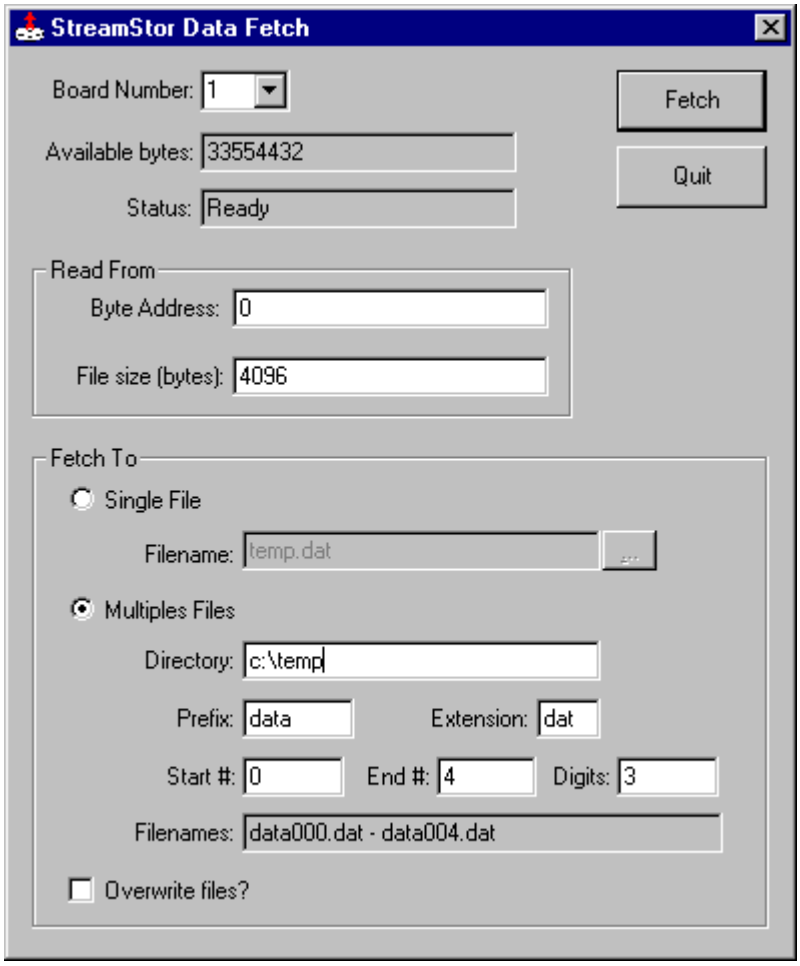
If you get any error messages running this test please follow the instructions in the Troubleshooting section. If this test completes successfully your StreamStor system is functioning normally.

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 library file `xlrapi.lib` is also included for linking the DLL functions to a user program. The required include files are `xlrapi.h` and `xlrtypes.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 Windows system directory.

Linux Library

When the SDK is installed on a Linux system a static function library is installed named `libssapi.a` with all the StreamStor API functions. The required include files are `xlrapi.h` and `xlrtypes.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.

API Functions

`XLRApiVersion` - Reports version of API library in use.
`XLRAppend` - Append data to an existing recording.
`XLRCardReset` - Reset an unopened StreamStor card.
`XLRClose` - Closes device and releases exclusive access.
`XLRDeleteAppend` - Delete the last appended data section.
`XLRDeviceFind` - Reports number of StreamStor cards present in system.
`XLRGetBaseAddr` - Get base address (physical) of StreamStor data window.
`XLRGetBaseRange` - Get size of StreamStor data window.
`XLRGetDeviceInfo` - Retrieves hardware configuration information.
`XLRGetDeviceStatus` - Get current status of device.
`XLRGetDirectory` - Get directory info on current recorded data.
`XLRGetDriveInfo` - Get information on an individual disk drive.
`XLRGetErrorMessage` - Get error string for supplied error code.
`XLRGetLastError` - Returns error code of last failure.
`XLRGetLength` - Returns number of available bytes.
`XLRGetLengthHigh` - Returns high word of recording length (bytes)
`XLRGetLengthLow` - Returns low word of recording length (bytes)
`XLRGetLengthPages` - Returns recording length in number of system pages
`XLRGetLengthLowHigh` -Returns low and high word of recording length (bytes)
`XLRGetPlayLength` - Returns the number of bytes played back.
`XLRGetSystemAddr` - Returns the kernel address of StreamStor data window.
`XLRGetVersion` - Reports version of StreamStor firmware components.

XLRGetWindowAddr	- Get user virtual address of StreamStor data window.
XLROpen	- Opens the device for exclusive access.
XLRPlay	- Reads data from StreamStor directly to PCI address
XLRPlayback	- Puts StreamStor into playback mode at the specified data address.
XLRRead	- Read data.
XLRReadData	- Same as XLRRead without structure.
XLRReadFifo	- Read data during a FIFO operation.
XLRReadImmed	- Read from StreamStor but return control without wait for completion.
XLRReadStatus	- Check for completion of XLRReadImmed request.
XLRRecord	- Start recording.
XLRRecoverData	- Recover data when a recording has been interrupted.
XLRReset	- Reset and close an open device.
XLRSetFifoMode	- Put StreamStor into FIFO operating mode.
XLRSetFPDPMode	- Set the operating mode of the FPDP data port.
XLRSetMode	- Set input/output mode of board.
XLRSetPortClock	- Set the clock speed of the external port.
XLRSetReadLimit	- Sets the range of any read accesses performed from an outside bus master.
XLRStop	- Stop recording.
XLRTruncate	- Truncate and existing recording at specified location.
XLRWrite	- Write a memory buffer to StreamStor
XLRWriteData	- Same as XLRWrite without structure

Data Structures

S_DEVINFO	- Device info parameters
S_DEVSTATUS	- Device status flags
S_DIR	- Recording directory
S_DRIVEINFO	- Drive information
S_READDESC	- Parameters defining read request
S_XLRSWREV	- Various device version strings

Chapter

4

Function Reference

XLRApiVersion

Syntax:

```
char *XLRApiVersion( char *versionstring )
```

Description:

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

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

Return Value:

versionstring is returned.

Usage:

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

XLRApiVersion( xlrstring );
printf( "%s", xlrstring );
```

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.

- *`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:

```
S_READDESC       readDesc;
ULONG            myBuffer[40000];
XLR_RETURN_CODE  xlrReturnCode;

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);
```

XLRCardReset

Syntax:

```
XLR_RETURN_CODE XLRCardReset(  UINT 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.

- *`index`* is the card index number (see `XLROpen`).

Return Value:

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

Usage:

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

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.

- *`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 );
```

XLRDeleteAppend

Syntax:

`XLR_RETURN_CODE XLRDeleteAppend( SSHANDLE` *`xlrDevice`*`, ULONG` *`AddrHigh`*`, ULONG` *`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.

- *`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 );
```

XLRDeviceFind

Syntax:

```
UINT XLRDeviceFind( )
```

Description:

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

Return Value:

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

Usage:

```
UINT  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");
}
```

XLRGetBaseAddr

Syntax:

`ULONG 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.

- *`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:

```
ULONG             xlrAddress;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

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

XLRGetBaseRange

Syntax:

`ULONG 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.

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

Return Value:

This function returns the window size in bytes.

Usage:

```
ULONG             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)
```

XLRGetDeviceInfo

Syntax:

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

Description:

`XLRGetDeviceInfo` retrieves info from the StreamStor device about its physical configuration.

- *`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 );
```

XLRGetDeviceStatus

Syntax:

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

Description:

`XLRGetDeviceStatus` retrieves status of the StreamStor device.

- *`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");
```

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. Use `XLRGetLengthPages` for new applications.

- *`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:

```
XLR_RETURN_CODE   xlrReturn;
S_DIR             xlrDir;

xlrReturn = XLRGetDirectory( xlrDevice, &xlrDir );
if( xlrReturn != XLR_SUCCESS )
      return(1);
```

XLRGetDriveInfo

Syntax:

XLR_RETURN_CODE XLRGetDriveInfo(SSHANDLE *xlrDevice*, UINT *Bus*, UINT *MasterSlave*, PS_DRIVEINFO *pDriveInfo*)

Description:

XLRGetDriveInfo retrieves info from the StreamStor drive about its physical configuration.

- *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", drvInfo.Serial );
printf("Drive model number is: %s", drvInfo.Model );
printf("Drive firmware revision: %s", drvInfo.Revision );
printf("Drive capacity (sectors): %d", drvInfo.Capacity );
```

XLRGetErrorMessage

Syntax:

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

Description:

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

- *`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];

xlrReturn = XLRGetDirectory( xlrHandle, &xlrDir );
if( xlrReturn != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( temp, xlrError );
   printf( "%s\n", temp );
   exit(1);
}
```

XLRGetLastError

Syntax:

```
XLR_ERROR_CODE XLRGetLastError( void )
```

Description:

`XLRGetLastError` returns the error code of the most recent API failure.

Return Value:

This function returns the error code (see Appendix A).

Usage:

```
XLR_ERROR_CODE    xlrError;
char              temp[XLR_ERROR_LENGTH];

xlrReturn = XLRGetDirectory( xlrDevice, &xlrDir );
if( xlrReturn != XLR_SUCCESS )
{
   xlrError = XLRGetLastError(  );
   XLRGetErrorMessage( temp, xlrError );
   printf( "%s\n", temp );
   exit(1);
}
```

XLRGetLength

Syntax:

`DWORDLONG XLRGetLength( SSHANDLE xlrDevice )`

Description:

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

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

Return Value:

- Current recording length in bytes.

XLRGetLengthHigh

Syntax:

```
ULONG XLRGetLengthHigh( SSHANDLE xlrDevice )
```

Description:

`XLRGeLengthHigh` returns the upper 32 bit integer of the current recording length in number of bytes. This function is provided for programming environments unable to handle 64 bit integers. Due to possibility of mismatch between subsequent calls of `XLRGetLengthHigh` and `XLRGetLengthLow` it is recommended that `XLRGetLength` or `XLRGetLengthPages` be used instead.

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

Return Value:

- Value of high word (bits 32-63) of recording length.

XLRGetLengthLowHigh

Syntax:

`void XLRGetLengthLowHigh( SSHANDLE` *`xlrDevice`*`, PULONG` *`low`*`, PULONG` *`high`* `)`

Description:

`XLRGeLengthLowHigh` returns the current recording length in number of bytes in two 32 bit variables. This function is provided for programming environments unable to handle 64 bit integers.

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen`.
- *`low`* is a pointer to a ULONG (unsigned int) that will be written with the lower 32 bits of the recording size in bytes.
- *`high`* is a pointer to a ULONG (unsigned int) that will be written with the upper 32 bits of the recording size in bytes.

Return Value:

- none

XLRGetLengthLow

Syntax:

```
ULONG XLRGetLengthLow( SSHANDLE xlrDevice )
```

Description:

XLRGeLengthLow return the lower 32 bit integer of the current recording length in number of bytes. This function is provided for programming environments unable to handle 64 bit integers. Due to possibility of mismatch between subsequent calls of XLRGetLengthHigh and XLRGetLengthLow it is recommended that XLRGetLength or XLRGetLengthPages be used instead.

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

Return Value:

- Value of low word (bits 0-31) of recording length.

XLRGetLengthPages

Syntax:

`ULONG XLRGetLengthPages( SSHANDLE xlrDevice )`

Description:

`XLRGetLengthLow` returns 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.

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

Return Value:

- Recording length in system pages

XLRGetPlayLength

Syntax:

`DWORDLONG XLRGetPlayLength( SSHANDLE` *`xlrDevice`* `)`

Description:

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

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

Return Value:

- Number of bytes played back.

Usage:

```
ULONG  addrHi, addrLow;
DWORDLONG    bytesPlayed;

addrHi = 0;
addrLow = 0xFE120000;

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

// Get the number of bytes that were played back.
bytesPlayed = XLRGetPlayLength(xlrDevice);
```

XLRGetSystemAddr

Syntax:

`ULONG 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.

- *`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:

```
ULONG             xlrAddress;
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrStatus;

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

XLRGetVersion

Syntax:

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

Description:

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

- *`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:

```
S_XLRSWVER   swVersion;

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 );
```

XLRGetWindowAddr

Syntax:

`PULONG 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.

- *`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:

```
PULONG            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 */
}
```

XLROpen

Syntax:

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

Description:

XLROpen opens a StreamStor device and initializes the hardware and firmware in preparation for recording. 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 *xlrHandle* can be used for all subsequent API calls.

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

Usage:

```
SSHANDLE          xlrHandle;
XLR_RETURN_CODE   xlrReturnCode;
ULONG             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 );
```

XLRPlay

Syntax:

```
XLR_RETURN_CODE XLRPlay( SSHANDLE xlrDevice, PS_READDESC pReadDesc
)
```

Description:

XLRPlay reads data from the StreamStor device and writes directly to a supplied PCI 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.

- *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:

```
S_READDESC   readDesc;

readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = XLRGetBaseRange( xlrDevice );
readDesc.BufferAddr = myDeviceAddress;

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

XLRPlayback

Syntax:

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

Description:

`XLRPlayback` puts StreamStor into playback mode where data is made available for reading by an outside device. This function is intended only for moving data between StreamStor and another device on the bus. The supplied address will be used to set the starting point of the data to be made available for read.

- *`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:

```
ULONG  addrHi, addrLow;

addrHi = 0;
addrLow = 0xFE120000;

xlrReturnCode = XLRPlayback( xlrDevice, addrLow, addrHi );
```

XLRRead

Syntax:

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

Description:

XLRRead reads data from the StreamStor device.

- *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:

```
S_READDESC   readDesc;
ULONG        myBuffer[40000];

readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

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

XLRReadData

Syntax:

```
XLR_RETURN_CODE XLRReadData( SSHANDLE xlrDevice, PULONG Buffer,
ULONG AddrHigh, ULONG AddrLow, ULONG XferLength )
```

Description:

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

```
ULONG        myBuffer[40000];

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

XLRReadFifo

Syntax:

```
XLR_RETURN_CODE XLRReadFifo( SSHANDLE xlrDevice, PULONG Buffer,
ULONG Length, BOOLEAN Direct )
```

Description:

XLRRead 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.

- *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:

```
ULONG        myBuffer[40000];

xlrReturnCode = XLRReadFifo(xlrDevice, myBuffer, sizeof(myBuffer), FALSE);
```

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.

- *`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:

```
S_READDESC       readDesc;
ULONG            myBuffer[40000];
XLR_READ_STATUS  readStatus;
XLR_RETURN_CODE  xlrReturnCode;

readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLRReadImmed( xlrDevice, &readDesc );

/* DO OTHER WORK HERE */

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

XLRReadStatus

Syntax:

```
XLR_RETURN_CODE XLRReadStatus( BOOLEAN Wait )
```

Description:

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

- *`Wait`* is a flag to indicate whether or not 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:

```
S_READDESC       readDesc;
ULONG            myBuffer[40000];
XLR_READ_STATUS  readStatus;
XLR_RETURN_CODE  xlrReturnCode;

readDesc.AddrHi = 0;
readDesc.AddrLo = 0xFE120000;
readDesc.XferLength = sizeof( myBuffer );
readDesc.BufferAddr = myBuffer;

xlrReturnCode = XLRReadImmed( xlrDevice, &readDesc );

while( moreWork )
{
   /* DO OTHER WORK HERE */

   readStatus = XLRReadStatus( FALSE );
   if( readStatus == XLR_READ_ERROR )
   {
      /* PROCESS ERROR! */
   }
   else if( readStatus == XLR_READ_COMPLETE )
      break;
}
```

XLRRecord

Syntax:

```
XLR_RETURN_CODE XLRRecord( SSHANDLE xlrHandle, BOOL 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.

- *`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:

```
/* Start recording data but insure that no captured data is overwritten */
xlrReturnCode = XLRRecord( xlrDevice, 0, 1);
```

XLRRecoverData

Syntax:

```
XLR_RETURN_CODE XLRRecoverData( SSHANDLE xlrHandle )
```

Description:

XLRRecoverData attempts to recover data from an interrupted recording. If a recording is ended without calling XLRStop (as might happen if the StreamStor's power fails), StreamStor's directory may be corrupted, making the recording unreadable. XLRRecoverData will examine the data on the disks and attempt to repair the directory accordingly. If successfully repaired, the recording should be readable.

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.

- *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:

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

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.

- *`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:

```
xlrReturnCode = XLRReset( xlrDevice );
```

XLRSetFifoMode

Syntax:

`XLR_RETURN_CODE XLRSetFifoMode( SSHANDLE xlrDevice, UINT Mode )`

Description:

`XLRSetFifoMode` controls the setting and clearing of FIFO modes.

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen`.
- *`Mode`* is a constant that defines the mode of operation. Possible values are: XLR_MODE_FIFO and XLR_MODE_DEFAULT.

Return Value:

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

Usage:

```
// Put StreamStor into FIFO mode of operation
xlrReturnCode = XLRSetFifoMode( xlrDevice, XLR_MODE_FIFO );
```

XLRSetFPDPMode

Syntax:

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

Description:

`XLRSetFPDPMode` is used to set the operating mode of the FPDP port on PCI-816XF and PCI-816XF2 model StreamStor boards.

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen`.
- *`Mode`* is a constant that defines the mode of operation. Possible values (refer to FPDP section of Chapter 8) are:
  `SS_FPDP_RECV` – Sets StreamStor port to FPDP/R mode.
  `SS_FPDP_RECVMASTER` – Sets StreamStor to FPDP/RM mode.
  `SS_FPDP_XMIT` – Sets StreamStor to FPDP/T mode.
  `SS_FPDP_XMITMASTER` – Sets StreamStor to FPDP/TM mode.
  `SS_FPDP_RECVMASTER_CLOCKS` – Sets StreamStor to FPDP/RMCM mode.
- *`option`* is used to specify various options that modify the operation of the FPDP port. Possible values are:
  - `0` (zero) – Disables all options.
  - `SS_OPT_FPDPNRASSERT` – Assert the "Not ready" signal on the FPDP bus when not recording. This prevents data flow on FPDP when StreamStor is not recording.
  - `SS_OPT_FPDPSTROB` – Enables data strobe clock (TTL strobe signals). Default is pstrob clock (PECL strobe signals).
  - `SS_OPT_FPDPEXTCONN` – Selects the connector on the external interface. Default is the card top connector.

Return Value:

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

Usage:

```
// Example 1: Set the FPDP port mode to FPDP/R and use the default
// options.
xlrReturnCode = XLRSetFPDPMode( xlrDevice, SS_FPDP_RECV, 0 );

// Example 2: Enable the data strobe clock and "Not Ready" assert options.
xlrReturnCode =
   XLRSetFPDPMode( xlrdevice,
   FPDP_RECV,  SS_OPT_FPDPSTROB|SS_OPT_NRASSERT );

// Example 3: Enable data strobe clock.
XLRSetFPDPMode( xlrdevice, FPDP_RECV, SS_OPT_FPDPSTROB );
```

XLRSetMode

Syntax:

`XLR_RETURN_CODE XLRSetMode( SSHANDLE xlrDevice, SSMODE Mode )`

Description:

XLRSetMode is used to set the input/output path of StreamStor boards that support external ports.

- `xlrDevice` is the device handle returned from a previous call to `XLROpen`.
- `Mode` is a constant that defines the mode for StreamStor's external port operation. Possible values are:
  `SS_MODE_PCI` - This is the default mode that receives and sends data over the PCI bus and is set when the device is opened.

  `SS_MODE_EXT` - In this mode the data is received or played over the external port. The `XLRRead` command is unaffected and will still allow retrieval of data from StreamStor to system memory. When recording (`XLRRecord` or `XLRAppend`) this command causes data to be recorded to disk from the external port. When `XLRPlay` is used, data is read from disk and output via the external port.

  `SS_MODE_READ_EXT` - This mode bypasses disk storage and uses the StreamStor buffer as a FIFO to read data from the external port. `XLRRecord` intiates the system to receive data into the StreamStor buffer in a first in/first out mode. `XLRReadFifo` is used to retrieve data into system memory from this FIFO buffer.

  `SS_MODE_WRITE_EXT` - This mode bypasses disk storage and uses the StreamStor buffer as a FIFO to write data to the external port. `XLRRecord` intiates the system and `XLRWrite` or memory copies to StreamStor memory put data into the buffer for delivery over the external port.

  `SS_MODE_PCI_FORK` - This mode "forks" data received on the PCI bus and records it to disk and outputs it to the external port. Care must be taken to insure the external port is able to output data fast enough to prevent overflowing the StreamStor buffer.

  `SS_MODE_EXT_FORK` - This mode "forks" data received from the external port and makes it available for reading to system memory.

  `SS_MODE_EXT_PASSTHRU` - This mode bypasses disk storage and cross connects the external ports through the StreamStor buffer. This is a hardware setup and no other StreamStor commands are accepted until a new mode is specified.

Return Value:

On success, this function returns XLR_SUCCESS.

On failure, this function returns XLR_FAIL.

Usage:

```
// Set StreamStor to use the external port
xlrReturnCode = XLRSetMode( xlrDevice, SS_MODE_EXT );
```

XLRSetPortClock

Syntax:

`XLR_RETURN_CODE XLRSetPortClock( SSHANDLE` *`xlrDevice`*`, UINT` *`clock`* `)`

Description:

`XLRSetPortClock` is used to set the operating frequency of the external port if applicable.

- *`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:

```
SS_PORTCLOCK_8MHZ
SS_PORTCLOCK_10MHZ
SS_PORTCLOCK_11MHZ   // 11.4 MHz
SS_PORTCLOCK_13MHZ   // 13.33 MHz
SS_PORTCLOCK_16MHZ
SS_PORTCLOCK_20MHZ
SS_PORTCLOCK_26MHZ   // 26.66 MHz
SS_PORTCLOCK_40MHZ
```

Return Value:

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

Usage:

```
// Set the external clock frequency
xlrReturnCode = XLRSetPortClock( xlrDevice, SS_PORTCLOCK_40MHZ );
```

XLRSetReadLimit

Syntax:

`XLR_RETURN_CODE XLRSetReadLimit( SSHANDLE` *`xlrDevice`*`, ULONG` *`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 recyles to a new starting read address on the PCI bus.

- *`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:

```
ULONG  DMA_size = 0x2000;
PULONG pBuffer;
PULONG pSSAddr;

// 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 = (PULONG)malloc(DMA_size);
pSSAddr = XLRGetWindowAddr( xlrDevice );

for( j = 0; j < loops; j++ )
{
   for( i = 0; i < DMA_size; i += 4 )
   {
      *pBuffer++ = *pSSAddr++;
   }
}
```

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.

- *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:

```
xlrReturnCode = XLRStop( xlrDevice );
```

XLRTruncate

Syntax:

`XLR_RETURN_CODE XLRTruncate( SSHANDLE` *`xlrDevice`*`, ULONG` *`AddrHigh`*`, ULONG` *`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.

- *`xlrDevice`* is the device handle returned from a previous call to `XLROpen`.
- *`AddrHigh`* is the upper 32 bits of the 64 bit address that identifies the location to truncate the recording at.
- *`AddrLow`* is the lower 32 bits of the 64 bit address that identifies the location to truncate the recording at.

Return Value:

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

Usage:

```
SSHANDLE          xlrDevice;
XLR_RETURN_CODE   xlrstatus;
ULONG             AddrHi;
ULONG             AddrLo;

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

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

// Truncate the recording.
AddrHi = 0;
AddrLo = 0xFE120000;

xlrstatus = XLRTruncate( xlrDevice, AddrHi, AddrLo );

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

XLRWrite

Syntax:

XLR_RETURN_CODE XLRWrite(SSHANDLE *xlrDevice*, PS_READDESC p*WriteDesc*)

Description:

XLRWrite writes data from a user memory buffer to StreamStor. StreamStor must be in record mode (XLRRecord or XLRAppend) before calling this function.

- *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.

Return Value:

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

Usage:

```
SSHANDLE     xlrDevice;
S_READDESC   writeDesc;
ULONG        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, &readDesc ) != XLR_SUCCESS )
   return(1);
```

XLRWriteData

Syntax:

```
XLR_RETURN_CODE XLRWriteData( SSHANDLE xlrDevice, PVOID BufAddr,
ULONG TransferSize )
```

Description:

`XLRWriteData` is identical to `XLRWrite` except that the parameters are not passed in a structure.

- *`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:

```
xlrReturnCode = XLRWriteData( xlrDevice, myBuffer, sizeof(myBuffer) );
```

Structure S_DEVINFO

```
typedef struct _DEVINFO
{
   char     BoardType[XLR_MAX_NAME];
   UINT     SerialNum;
   UINT     NumDrives;
   UINT     NumBuses;
   UINT     TotalCapacity;
   UINT     MaxBandwidth;
   UINT     PciBus;
   UINT     PciSlot;
}S_DEVINFO, *PS_DEVINFO;
```

Purpose

This structure is used by the `XLRGetDeviceInfo` function to return data about the StreamStor system configuration.

Members

- *`BoardType`* - String holding the board type (model name).
- *`SerialNum`* - Serial number of the StreamStor board.
- *`NumDrives`* - Number of drives currently connected and configured on the StreamStor controller.
- *`NumBuses`* – Number of ATA buses in use
- *`TotalCapacity`* – Total recording capacity of the StreamStor system in system pages (a page is 4096 bytes typically on Intel based Windows NT/2000 systems).
- *`MaxBandwidth`* – Reserved
- *`PciBus`* – The PCI bus number that this board is installed into.
- *`PciSlot`* – The PCI slot number that this board is installed into.

Structure S_DEVSTATUS

```
typedef struct _DEVSTATUS
{
   BOOLEAN SystemReady;
   BOOLEAN BootmonReady;
   BOOLEAN Recording;
   BOOLEAN Playing;
   BOOLEAN VRActive[XLR_MAX_VRS];
   BOOLEAN RecordActive[XLR_MAX_VRS];
   BOOLEAN ReadActive[XLR_MAX_VRS];
   BOOLEAN FifoActive;
   BOOLEAN DriveFail;
   UINT    DriveFailNumber;
   BOOLEAN SysError;
   UINT    SysErrorCode;
   BOOLEAN CtlrError;
   BOOLEAN FifoFull;
   BOOLEAN Overflow[XLR_MAX_VRS];
}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.
- *`VRActive`* – Indicates that the specified virtual recorder is configured and ready.
- *`RecordActive`* – Indicates that the specified virtual recorder is currently recording.
- *`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.
- *`CtlrError`* – Indicates an ATA controller has failed.
- *`FifoFull`* – Indicates the system is at capacity while in FIFO mode.

- `Overflow` – Indicates the disk drives reached capacity during a record operation.

Structure S_DIR

```
typedef struct _DIR
{
   DWORDLONG   Length;
   DWORDLONG   AppendLength;
   BOOLEAN     Full;
}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 (DWORLDLONG).

Members

- *`Length`* – Length of the current recording in bytes. Note that this parameter is a 64 bit number.
- *`AppendLength`* - 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.

Structure S_DRIVEINFO

```
typedef struct _DRIVEINFO
{
   char     Model[XLR_MAX_DRIVENAME];
   char     Serial[XLR_MAX_DRIVESERIAL];
   char     Revision[XLR_MAX_DRIVEREV];
   UINT     Capacity;
   BOOLEAN  SMARTCapable;
   BOOLEAN  SMARTState;
}S_DRIVEINFO, *PS_DRIVEINFO;
```

Purpose

This structure holds information about a disk drive installed in the system. The structure is filled with a call to `XLRGetDriveInfo`.

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 serial number as reported by the disk drive identify command.
- *`Capacity`* – Drive capacity as reported by identify command. Value is number of 512 byte sectors.
- *`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`.

Structure S_READDESC

```
typedef struct _READDESC{
   PULONG   BufferAddr;
   ULONG    AddrHi;
   ULONG    AddrLo;
   ULONG    XferLength;
}S_READDESC, *PS_READDESC;
```

Purpose

This structure is used to define the parameters for a read from StreamStor.

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_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];
}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 software.
- *MonitorVersion* – Boot monitor firmware version.
- *XbarVersion* – Controller logic version.
- *AtaVersion* – ATA controller version.
- *UAtaVersion* – Ultra ATA controller version.

Chapter

5

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 record 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 PCI bus. The PCI bus number and slot number are available from the `XLRGetDeviceInfo` command which can be useful in identifying the appropriate card in a multi-card system.

Chapter

6

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 fills to capacity.

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. 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 `XLRGetLengthPages` 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

7

Disk FIFO...

Disk FIFO

The optional Disk FIFO feature is available on StreamStor 408 and 816 recorders to provide simultaneous Record and Read functionality. This ability can be used to capture real-time data streams while simultaneously reading data for processing or analyzing in a "first-in, first-out" fashion. The StreamStor controller will manage the internal disk resources to insure an uninterrupted record path while still allowing the data to be read out in the order received. The disk resources are divided into two "banks" with a bank dedicated to either the read or write function. The disk banks are utilized in a double buffering technique to insure an uninterrupted real-time recording capability. The minimum available capacity is approximately ½ the full capacity on the 408 and 816 boards.

Setting FIFO mode

The function `XLRSetFifoMode` is used to initiate FIFO operations on StreamStor. This function takes a mode parameter that indicates the FIFO mode to enter. The currently available modes are:

`XLR_MODE_DEFAULT`	–	Default StreamStor operating mode.
`XLR_MODE_FIFO`	–	FIFO Operating mode.

Other modes will be available in future software releases. Whenever an application initiates a session by calling `XLROpen` the recorder will return to the default operating state.

Recording to FIFO

Once StreamStor has been put into FIFO mode, the standard `XLRRecord` function call is used to put the recorder into record mode. Note that the `XLRAppend` functionality is not available while in FIFO mode. Data is routed to StreamStor in FIFO mode in exactly the same manner as in standard mode.

Ending FIFO Record

While in FIFO mode the `XLRStop` function will cause the StreamStor system to stop recording and discard all data still left in the FIFO. To retain the data still in the FIFO you can stop the data-flow from the source card and continue to read until all necessary data has been read. You may also allow the data source to continue sending data until all relevant data has been read. StreamStor will indicate a FIFO Full status (`XLRGetDeviceStatus`) and automatically stop recording data when the disk system is at capacity. Even if the data source continues to send data after the full condition, StreamStor will not record it. Once all data has been read from the FIFO, the `XLRStop` function will end record mode and ready StreamStor for the next operation. Note that the recorder will remain in FIFO mode unless a new call is made to `XLRSetFifoMode` to return StreamStor to its default state.

Reading FIFO data

Reading data from the FIFO is very similar to reading data in default mode. The function `XLRReadFifo` is provided to read data sequentially from the StreamStor system. Data is always read in a "first-in, first-out" fashion. Once the data has been read from the recorder the space is reclaimed for recording new data. The `XLRGetLength` function can be used to determine how much data is currently available in the FIFO. If a read request is made before enough data is available the `XLRReadFifo` function will return `XLR_FAIL` and the last error will be set to `XLR_ERR_OUTOFRANGE`. The `XLRStop` function should not be used until all desired data has been read using `XLRReadFifo`. The data in the FIFO is not preserved after the `XLRStop` function has been executed.

Chapter

8

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 busses 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 StreamStor PCI-816XF and PCI-816XF2 controllers. 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. Sustained speeds up to 200Mbytes/sec are supported on the StreamStor FPDP interface.

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.

Connector Position

The location of the FPDP connector on the StreamStor board is shown in Figure 3.

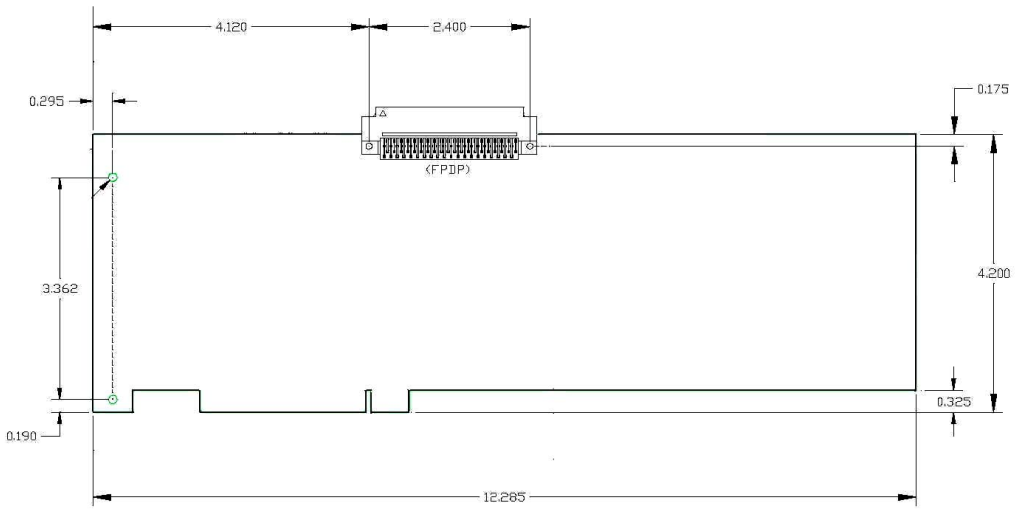


Figure 3 - StreamStor FPDP Connector location

Interface Functions

To ready the StreamStor to transfer data using FPDP, the API routine `XLRSetMode` must be called, as follows:

```
XLRSetMode ( device, SS_MODE_EXT );
```

After StreamStor is in external port mode, an API call to `XLRSetFPDPMode` 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)

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. XLRSetFPDPMODE 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:

```
XLRSetFPDPMode( device, FPDP_RECVMASTER, SS_OPT_FPDPNRASSERT );
```

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 `XLRSetFPDPMode` with the desired clock option. For example, to enable the data strobe clock (TTL):

```
XLRSetFPDPMode( device, FPDP_RECV, SS_OPT_FPDPSTROB );
```

Refer to the FPDP ANSI standard for recommendations and observations about the use of these signals.

[1] StreamStor always provides series termination on the DATA signals as described in Permission 6.4.1 of the ANSI specification.

Operating Frequency Range

In either FPDP/TM or FPDP/RMCM mode, StreamStor can be programmed to synthesize a bus clock in the range from 6 to 50MHz. StreamStor can operate from FPDP clocks supplied by other sources at frequencies down to DC. Note, however, that the ANSI specification limits the clock to 20MHz if a receiver is using the STROB (Data Strobe) clock. To program the clock, use the API function `XLRSetPortClock`. By default, the clock frequency is 8MHZ.

Chapter

9

If You Have Problems...

(303) 485-2721

support@conduant.com

www.conduant.com

Help Us Help You

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 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 in sscfg.exe run OK?

Has the software installation been corrupted? Try re-installing software.

Have you checked the Conduant Web site for technical bulletins?

Have you checked the Software Update page in the Conduant Web Site to be sure that your software is fully up to date? If your software is down level, you may want to update it to determine if this fixes the problem.

If the above steps did not resolve the problem, then please call Technical Support or contact them through the web site. Please provide the following information:

* StreamStor Card Serial Number

* Software Revision(s)

* Configuration (308,816,816XF, 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) 485-1247

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 32

		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_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.
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.

End of Document