LynxSoft™ 1394 Software Application Programmer User's Guide

SLLU003
February 24, 1998

Preliminary
Version 2.2

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

This application programmer interface is derived from the Microsoft 1394 BUS Interfaces document. Portions of this document are copyrighted by Microsoft in their 1394 BUS Interfaces document and are reprinted here with the permission of Microsoft Corporation. The interfaces described herein are not guaranteed to remain static in the future. Users should migrate to the Microsoft Win32 DDK when it is available.

Read This First

The LynxSoft™ Application Programmer User's Guide discusses the theory of operation for the LynxSoft Application Programmer Interface (API). The 1394 bus driver API commands are also covered. The commands are given as API functional descriptions or device function requests. Parameter titles for each function always appears in italics within the parameter listing. An installation procedure is provided followed by the test application and the source code needed. The configuration ROM is also described in this user guide.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477-8924**
Ask questions about product operation or report suspected problems	Call Texas Instruments Mixed Technical Support: (972) 480-4546 E-Mail: ANSW@msg.ti.com

Trademarks

Contents

Tables

Figures

Introduction

1 Introduction

This document describes the Texas Instruments (TI™) implementation of an Application Programmer Interface (API) for the 1394 bus. This API closely follows the Microsoft bus interface to allow easy migration of software applications to the new Win95/WinNT Microsoft 1394 support. However, due to this interface being a monolithic driver set and not based on the Windows Device Model, there are some differences, both additions and deletions. The goal was to make the data structures, calling sequences, and control mechanisms similar in order to make this transition easier.

LynxSoft API Theory of Operation

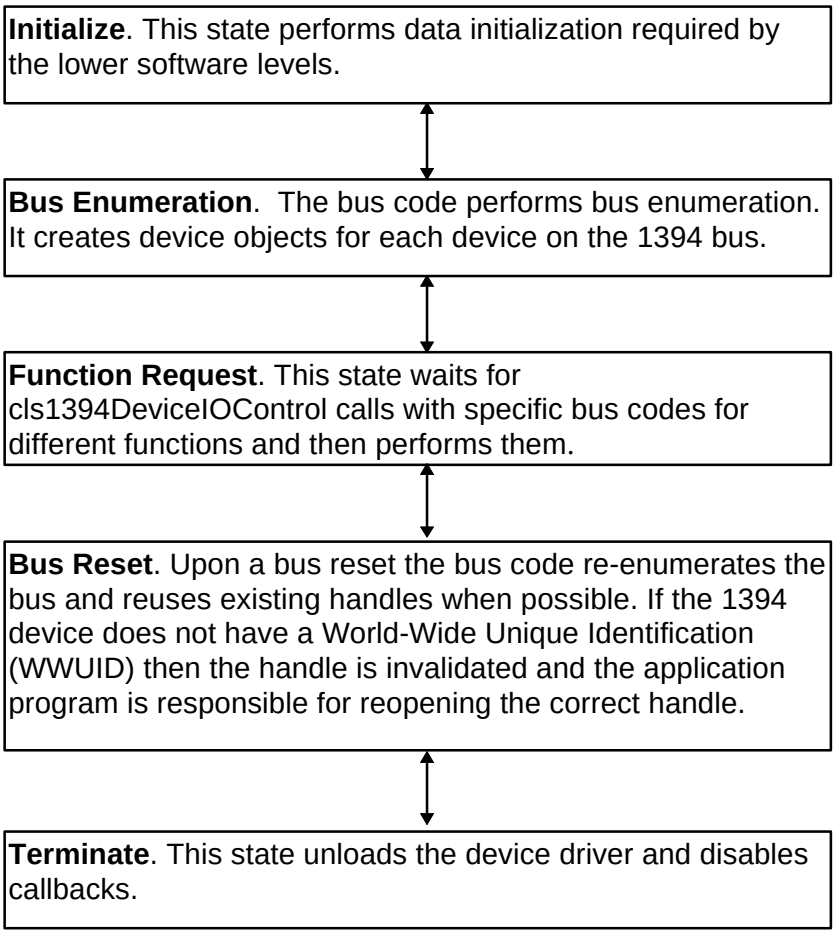
2 Theory of Operation

This section describes the architecture of the LynxSoft software product. This section also provides an overview of the services provided by the software, and a description of initialization provided by the software. Also described are operations required during asynchronous and isochronous transmissions.

2.1 LynxSoft API Description

The 1394 Lynx API performs the necessary functions required to do 1394 operations of both a synchronous and asynchronous nature. Hereafter the API code is referred to as a bus driver. Device objects are created upon bus initialization and 1394 bus resets that describe the currently known properties of the 1394 bus, including device speed, device nodes, isochronous bandwidth, and isochronous channel allocations. The use of callbacks allows the programmer to have a method of controlling the 1394 bus without requiring a large amount of polling. The states that the bus code exists in are Initialize, Bus Enumeration, Function Request, Bus Reset and Terminate. The states with a short description are shown in Figure 2–1.

Figure 2-1 1394 Bus States



The 1394 bus driver acts as a bus enumerator for the 1394 bus. The 1394 hardware tree is built by discovering hardware devices on the 1394 bus. The

discovery of a device has the effect of creating a new device object for that device. There is a device object created for every device that is found on the 1394 bus. Handles to the device objects created by the 1394 bus driver are used by the application code to address the 1394 device for which a particular function is targeted. The attempt is to shield the user application code from the inner-workings of the 1394 bus. For example, when a bus reset occurs and the actual device that is pointed to by the device object handle has a World-Wide User ID (WWUID), then the handle stays valid and the application need not be concerned that a bus reset has occurred. However, if the actual device has not implemented a WWUID (non-compliant device), then the handle must be invalidated and the application program must reopen a handle to the device object in question. In this case the application would have to connect to the device by opening a handle to a specific node and then either knowing the bus configuration or interrogating the non-compliant device. Attempts to perform a function using an invalid handle results in an error return.

The 1394 bus can be reset infinitely during the course of normal operations. The bus driver does not attempt to reclaim isochronous resources through a bus reset. The application programmer is responsible for returning bandwidth and isochronous channels to the bus driver.

2.2 Isochronous Transmission

Isochronous transmission has a specific sequence that needs to be followed for successful operation. The sequence of events is shown in Table 2–1, as well as illustrated in the example code contained in this document.

Note:

It is very important to follow the sequence shown in Table 2–1; unpredictable results can occur when the sequence is not followed. Bandwidth and Channel allocation/de-allocation are sequence independent relative to each other.

Table 2-1 Isochronous Transmission Sequence

Operation	Result
cls1394Initialize	Initializes the device driver and LynxSoft API code.
Cls1394CreateFile	Locates and obtains a handle to the device that is to be transmitted to and received from.
Cls1394IsochAllocateBandwidth	Ensures that there is enough bandwidth still available on the 1394 bus for the operation that is to be performed.
Cls1394IsochAllocateChannel	Ensures that there exists an isochronous channel for transmission.
Cls1394IsochAllocateResources	*Must* perform this operation before the buffers are attached.
Cls1394IsochAttachBuffers	Attaches buffers to be transmitted to or from. This function *must* follow allocation of resources. The handle passed in *must* be valid.
Cls1394IsochListen	Begins the operation and monitors the callback routines for status. The callbacks *must* be handled in a timely fashion. If callbacks cannot be handled they start to back up in the queue and eventually cause a system crash. When the bandwidth is insufficient to perform all processing before the next callback occurs, the amount of data transferred per buffer or the packet size transferred should be decreased. The watermark callback also can allow the processing to begin before the buffer has completely filled.
Cls1394IsochTalk	Begins the operation and monitors the callback routines for status. The callbacks *must* be handled in a timely fashion. If callbacks cannot be handled correctly they back up in the queue and eventually cause a system crash. When the bandwidth to perform all processing before the next callback occurs, the amount of data transferred per buffer or the packet size transferred should be decreased. The watermark callback also can allow the processing to begin before the buffer has completely filled.
Cls1394IsochStop	Halts the transfer. This *must* be done before the buffers or resources are de-allocated.
Cls1394IsochDetachBuffers	Detaches the buffers used in the transfer. This action *must* be completed before the resource handle is freed. However, the allocate resource handle can be reattached to another buffer.
Cls1394IsochFreeResources	Frees the resources. This action *must* performed *after* the buffers have been detached.
Cls1394IsochFreeBandwith	Frees bandwidth
cls1394IsochFreeChannel	Frees channel allocation
cls1394CloseHandle	Releases the handle opened.
Cls1394Terminate	Terminates the application.

2.3 Asynchronous Transmission

Asynchronous transmission does not require as precise a calling order as isochronous transmission. However there are a few rules that must be followed. Table 2-2 below describes some asynchronous functions and some of the required sequence considerations before using the functions.

Table 2-2 Asynchronous Transmission Sequence

Operation	Result
cls1394Initialize	Initializes the device driver and LynxSoft API code.
Cls1394CreateFile	Locates and obtains a handle to the device that is to be transmitted to or received from.
Cls1394AllocateAddressRange	Provides a buffer to handle asynchronous traffic. The application software can read/write to any configuration status register (CSR) space that has been made available by the target device. In the case of another LynxSoft API, the target device must have allocated their address range for writing.
Cls1394AsyncWrite, cls1394AsyncRead	Performs a read/write to any CSR space that has been made available by the target device. In the case of another LynxSoft API, the target device must have allocated their address range for writing.
Cls1394FreeAddressRange	Frees the allocated address range.
Cls1394CloseHandle	Releases the handle opened.
Cls1394Terminate	Terminates the application.

2.4 Bus Enumeration

The bus enumeration process allows the application programmer to access a device without knowing the device characteristics (bus node, speed...etc.). The enumeration process happens upon either a bus reset or the execution of the cls1394Initialize function. This function causes a bus reset to occur. Bus enumeration is needed to allow the LynxSoft API the opportunity to become the bus manager and to find devices and create device objects for them. The enumeration process is described below.

2.4.1 Initial Bus Reset

Upon a bus reset the LynxSoft device driver begins receiving the self-ID packets from all nodes on the 1394 bus. The API requests the topology and speed maps from the bus manager functions.

2.4.2 Bus Enumeration

Query all nodes on the bus and request their Configuration Info Block. This block contains the WWUID for each device. When a WWUID does not exist then the bus enumeration classifies that device as non-compliant (NC) and creates an NC device object. All of the device objects are ordered depending on their WWUID and their bus node addresses. For example, if two Sony™ cameras are on the bus they are ordered as Sony camera #1 and #2. This allows the application software to open a Sony camera device object and ask for #1 or #2. The application can then open Sony camera #2 and perform reads of the configuration block to determine if that is the type desired.

In the case of an NC device (one without a WWUID), the bus enumeration puts it in a list according to its node ID. Therefore, an application can open the nth NC device and communicate. This requires a strong knowledge of the topology of the 1394 bus. After opening the device, the application can then query the device, if possible, to determine which device it is. As soon as all 1394 devices have a WWUID then this is not necessary, the application can read the information block.

2.4.3 Bus Reset After Initial Enumeration

When a bus reset occurs after the 1394 bus has been enumerated, the devices that contained a WWUID retains the same handle returned during the cls1394CreateFile function. This allows the application software to continue operation without needing knowledge of the bus reset. The handle would continue to point to the same 1394 device. However, in the case of an NC device, the handle is no longer valid and must be reopened for communication. In that case the LynxSoft API returns an error upon attempting an operation with an invalid handle.

1394 Bus Driver API

3 LynxSoft API

The LynxSoft API is a set of services that enable 1394 actions. Each service consists of an input parameter to the class 1394 Device IO Control function and a structure containing input parameters. Only cls1394DeviceIOControl is actually used as a function call. Several other functions provide information that will be needed for the call and others provide utility type services.

3.1 cls1394Initialize

Description Initializes the 1394 bus driver software

Action This function performs needed initialization of the 1394 bus driver software. The function must be invoked to initiate the connection to the underlying device drivers. No bus functions can operate if initialization has not taken place. This function is invoked as follows:

Syntax typedef BusResetCB(pBusResetInfo InfoBlock)

BOOL cls1394Initialize(BusResetCB theProc);

Parameters **theProc** – Is a pointer to the Applications bus reset callback routine. This allows the CLASS to indicate to the application that a bus reset has occurred. Can be NULL. The application should minimize operations in this callback.

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.2 cls1394CreateFile

Description Finds and obtains a handle to a device object

Action This function finds a desired device on the enumerated 1394 bus and returns a handle to the device. When the device has a WWUID then this handle lives through a bus reset, otherwise it is invalidated and the application must reopen the handle.

Syntax cls1394HANDLE cls1394CreateFile (ULONG VendorID_DeviceType, WORD DeviceEntry);

Parameters *VendorID_DeviceType* –Is the IEEE vendor ID and vendor device type to open the vendor ID for the device of interest. For example, a Sony desktop camera Vendor ID is 08004602. If the VendorID_Device type is 0, then the function returns a handle to a non-compliant device that is located at the device handle specified by DeviceEntry. This handle would be used when communicating with devices that may not have implemented a WWUID. This number is a hexadecimal number. To open another PCILynx card the VendorID_DeviceType is 08002850.

DeviceEntry - Is the nth entry for the device type requesting a handle. This allows multiple devices of the same type to be opened. When the device ID is 0 and the VendorID_DeviceType is 08002850 then a handle to the PCILynx host adapter is returned. This number is a hexadecimal number.

Return Status When the cls1394HANDLE is null then the last error code should be checked.

3.3 cls1394GetLastError

Description Obtains the error number upon a status failure. ACK codes (00-0F) see 1394 spec (6.2.5.2.2). LLC codes (10-1F) see PCILynx spec PCL status section

Action This function returns the last error set by any of the bus functions. This function is invoked as follows:

Syntax ULONG cls1394GetLastError();

Error Codes

CLASS1394_ACK_00	0x00	rsvd
CLASS1394_ACK_COMPLETE	0x01	
CLASS1394_ACK_PENDING	0x02	
CLASS1394_ACK_03	0x03	rsvd
CLASS1394_ACK_BUSY_X	0x04	
CLASS1394_ACK_BUSY_A	0x05	
CLASS1394_ACK_BUSY_B	0x06	
CLASS1394_ACK_07	0x07	rsvd
CLASS1394_ACK_08	0x08	rsvd
CLASS1394_ACK_09	0x09	rsvd
CLASS1394_ACK_0A	0x0A	rsvd
CLASS1394_ACK_0B	0x0B	rsvd
CLASS1394_ACK_0C	0x0C	rsvd
CLASS1394_ACK_DATA_ERROR	0x0D	hardware error, data unavailable
CLASS1394_ACK_TYPE_ERROR	0x0E	incorrect request packet
CLASS1394_ACK_0F	0x0F	rsvd
CLASS1394_LLC_RETRYOVERRUN	0x10	
CLASS1394_LLC_TIMEOUT	0x11	
CLASS1394_LLC_FIFOUNDERRUN	0x12	
CLASS1394_LLC_13	0x13	
CLASS1394_LLC_14	0x14	
CLASS1394_LLC_NO_PKT_END	0x15	
CLASS1394_LLC_PIPELINE_ERR	0x16	
CLASS1394_LLC_17	0x17	
CLASS1394_LLC_18	0x18	
CLASS1394_LLC_19	0x19	
CLASS1394_LLC_1A	0x1A	
CLASS1394_LLC_1B	0x1B	
CLASS1394_LLC_1C	0x1C	
CLASS1394_LLC_1D	0x1D	
CLASS1394_LLC_CORRUPT_HEADER	0x1E	
CLASS1394_LLC_1F	0x1F	
CLASS1394_SUCCESS successful	0x20	Class Operation code
CLASS1394_GENERIC_FAILURE	0x21	Generic Fail code
CLASS1394_INVALID_REQUEST	0x22	DeviceIoControl
CLASS1394_WWUID_INVALID	0x23	CreateFile
CLASS1394_WWUID_NOTFOUND	0x24	CreateFile
CLASS1394_HANDLE_INVALID	0x25	CloseHandle
CLASS1394_HANDLE_NOTOPEN	0x26	CloseHandle
CLASS1394_SPEEDMAP_ERROR map	0x27	Error accessing speed

CLASS1394_NUM_DESTINATIONS_ERR	0x28	Num of destinations for speed < 0
CLASS1394_RESPONSE_UNEXPECTED	0x29	rsvd rCode in response packet
CLASS1394_RESPONSE_CONFLICT	0x2A	resource conflict, retry request
CLASS1394_RESPONSE_DATAERR	0x2B	HW error, data unavailable
CLASS1394_RESPONSE_TYPEERR	0x2C	Unsupported/invalid field, invalid request
CLASS1394_RESPONSE_ADDRERR	0x2D	address not accessible
CLASS1394_RESPONSE_ZERO_DATA	0x2E	zero data in data payload packet
CLASS1394_RESPONSE_TIMEOUT	0x2F	Timeout - no response in allotted time
CLASS1394_ISOALLOCRES_MEM_FAIL	0x30	Failed to allocate Internal Class Resource Struct
CLASS1394_SPEED_NOT_AVAIL	0x31	Failed to Acquire speed requested by user
CLASS1394_INVALID_ADDRRNG	0x32	Failed to Allocate/Free 1394 Address Range
CLASS1394_INTERNAL_DEV_OPEN	0x33	Failed to Open 1394 Host Device
CLASS1394_INTERNAL_DEV_ADDR	0x34	Invalid 1394 Host Device Address
CLASS1394_INTERNAL_CSC_SPACE	0x35	Failed to Allocate CSR space
CLASS1394_INTERNAL_CSR_INIT	0x36	Failed to Initialize CSR space
CLASS1394_BANDWIDTH_INVALID	0x37	Bandwidth request not valid
CLASS1394_BANDWIDTH_UNAVAIL	0x38	Bandwidth request not available
CLASS1394_CHANNEL_INVALID	0x39	Channel request not valid
CLASS1394_CHANNEL_UNAVAIL	0x3A	Channel request not available
CLASS1394_ISO_PCL_MEMORY_FAIL	0x3B	User Buffer Lock Down failed
CLASS1394_ISO_SCATTERLOCK_FAIL	0x3C	User buffer scatter lock failed
CLASS1394_ISO_RESOURCE_INVALID	0x3D	ISO resource is invalid
CLASS1394_DMA_CHAN_NOT_FOUND	0x3E	DMA channel is not available
CLASS1394_TRANSMISSION_FAILURE	0x3F	Transmission failure occurred
CLASS1394_BUS_RESET_IN_PROGRESS	0x40	Bus reset occurring

3.4 cls1394Terminate

Description Close and terminate the 1394 bus driver software

Action This function terminates the bus driver software and releases resources back to the operating system.

Syntax BOOL cls1394Terminate();

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.5 cls1394CloseHandle

Description Returns a previously received handle to the bus driver

Action This function closes a previously allocated handle to a device object. This allows the bus driver to free some resources for use.

Syntax BOOL cls1394CloseHandle(cls1394HANDLE hHnd);

Parameters *hHnd* - This parameter is a previously allocated handle that was obtained from the cls1394FileOpen function.

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.6 cls1394DeviceIoControl

Description Performs a 1394 function request

Action All of the 1394 function requests are performed by invoking the 1394DeviceIORequest call. This call is meant to emulate the Microsoft IORequestCalls that are used in the upcoming Win95 1394 support. The calls are invoked by performing a cls1394DeviceIoControl call and setting the dwIoControlCode to the appropriate function call.

Syntax

```
BOOL cls1394DeviceIoControl (cls1394HANDLE hDevice
        DWORD dwIoControlCode
        LPVOID lpInBuffer
        DWORD nInBufferSize
        LPVOID lpOutBuffer
        DWORD nOutBufferSize
        LPDWORD lpBytesReturned
        LPOVERLAPPED lpOverlapped)
```

Parameters:

hDevice - Is the device object handle to which the operation is targeted. This handle is obtained by a call to cls1394CreateFile.

dwIoControlCode - Determines which function of the bus library is invoked. Refer to the particular function the user desires to be invoked for the correct input value.

lpInBuffer - Is the input structure required by the function that is desired to be invoked. The caller must cast this structure to the structure of interest.

nInBufferSize - Is not used

lpOutBuffer - Is not used

nOutBufferSize - Is not used

lpBytesReturned - Is not used

lpOverlapped - Is not used

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.6.1 cls1394AllocateAddressRange

Description

Allocates 1394 virtual address space and maps it to a physical buffer on the host node for asynchronous requests. An application cannot overlap 1394 addresses.

Action

This function allocates a 1394 address range to be used in asynchronous requests to the local 1394 node. The caller supplies a buffer, possibly a specific 1394 address, the type of access a remote device has to this memory, as well as optional notification options so that the caller can be notified when this memory has been accessed.

If the function call is successful, the API driver maps 1394 address(es) to the caller-supplied memory region, and returns the newly mapped 1394 address to the caller. The caller can then supply this address to remote 1394 nodes, thus allowing the nodes to perform asynchronous requests to this memory region.

Callers of this API can elect to supply a specific 1394 address as pointed to by the *Required1394Offset* parameter. This parameter is necessary to support devices that issue asynchronous requests to hard coded 1394 addresses. When *RequiredP1394Offset* specifies a required address, then this 1394 address is always returned. An arbitrary 1394 address can be assigned by making the *Required1394Offset* parameter NULL. The API will locate a region in 1394 virtual memory to contain the request. The virtual 1394 address as well as a list of PHYSICAL address will be returned to the caller. An important point to consider when using this API is that *lp1394Address* points to *an array of PHYSICAL addresses* to be returned. The array of returned addresses holds the physical memory locations spanned by the specified buffer. These PHYSICAL addresses can be used for performing DIRECT DMA from remote node to HOST.

Callers of this API can choose *not* to supply a buffer (i.e. *lpBuffer* == NULL). This has the effect of allocating a 1394 address range that does not map to any real PC memory. When incoming requests try to access this 1394 address range, the *lpCallback* routine (must be specified for *lpBuffer* == NULL) is called and returns a packet pointer to the transferred data. The application has the responsibility of moving the data to the desired local-memory location.

This API is invoked by calling the 1394DeviceIOCtl function with the dwIoControlCode equal to the published value of IOCTL_P1394_CLASS, the FunctionNumber within the P1394_CLASS_REQUEST being equal to CLS_REQUEST_ALLOCATE_ADDRESS_RANGE, and the request union field filled in with the following structure:

Input

```
struct {
        QUADLET             *lpBuffer;
        ULONG                       nLength;
        ULONG               fulAccessType;
        ULONG                       fulNotificationOptions;
        LPVOID              lpCallback;
        LPVOID              lpContext;
        P1394_OFFSET        Required1394Offset;
        PULONG              lpAddressesReturned;
        PLARGE_INTEGER      lp1394Address;
} clsAllocateAddressRange;
```

Parameters

lpBuffer – When specified, points to the application's buffer where asynchronous operations are to be read, written, or locked. When NULL is specified, then *lpCallback* must be provided as the caller is consulted in order to return whatever results are requested from this address range.

nLength –Specifies the length in bytes of the 1394 address to map.

fulAccessType – When specified, dictates what type of access is allowed to the specified memory region. This field is used to restrict access by specified devices. These bit definitions can have an OR function performed to achieve the desired access such as:

- ❏ AccessTypeRead - The memory region specified can be the target of a read operation by the device.
- ❏ AccessTypeWrite - The memory region specified can be the target of a write operation by the device.
- ❏ AccessTypeLock - The memory region specified can be the target of a lock operation by the device.

fulNotificationOptions –Specifies what kind of post notification the application callback needs when this region of memory is accessed. The different options are enabled by using an OR function with the defines specified below into this parameter. Multiple types of notification are allowed for the same 1394 address. Types of notification are:

- ❏ NotifyAfterRead – This option notifies the application callback after carrying out an AsyncRead operation. This serves only as a notification to the application callback that their address space was accessed.
- ❏ NotifyAfterWrite – This option notifies the application callback after carrying out an AsyncWrite operation. This serves only as a notification to the application callback that their address space was written.
- ❏ NotifyAfterLock - This option notifies the application callback after carrying out an AsyncLock operation. This serves only as a notification to the application callback that their address was the target of an Atomic operation.

lpCallback – Points to the application callback routine. This routine is called by the 1394 class driver at deferred process (DPC) time for post notifications, and possibly at the interrupt level on pre-notification. Pre-notification callbacks only occur when the *lpBuffer* parameter (above) is NULL, which indicates that the application wants to handle each request to this address range itself.

When using post notifications, the callback return code (RESPONSE_CODE) is ignored. Modifying any of the other parameters also has no effect.

When using pre-notification callbacks in all asynchronous cases, the application callback function must return an appropriate 1394 response code, which is put into the 1394 response packet RCODE field.

If the incoming asynchronous request was a Read or Lock, then the application callback function must also set *lpData* function (*lpData) to point at a buffer containing the response data, as well as set *lpLength* to be the length of *lpData*.

If the incoming asynchronous request was a Write request, then *lpData* and *lpLength* specify where the write data is contained in memory and how much is present. The callback function for an Asynchronous Write request can do whatever with *lpData* and *lpLength* as long as an appropriate response code is returned.

Callback Syntax

```
RESPONSE_CODE ddNotificationCallBack(
        IN LPVOID lpBuffer,
        IN ULONG ulOffset,
        IN PVOID * lpData,
        IN PULONG * lpLength,
        IN DWORD dwSourceAddress,
        IN ULONG fulNotificationOption,
        IN LPVOID lpContext,
        );
```

Callback Parameters

lpBuffer - is a pointer to the buffer that was originally submitted in the call to cls1394AllocateAddress.

ulOffset - Specifies the byte offset within lpBuffer where the 1394 operation is pending. This offset is from the base 1394 virtual address that was mapped.

lpData - Points to the pointer which in turn points to a buffer where request/response data is stored. When the incoming asynchronous request is a Write, then the Write request data is pointed at by lpData. When the incoming asynchronous request is a Read or Lock, the callback function fills in lpData to point at response data. This lpData field is only used to pre-notify AllocateAddressRange conditions (i.e. original lpBuffer == NULL).

lpLength - Is a pointer which in turn points to the length in bytes of the requested 1394 operation. When incoming asynchronous request is a Write, then the Write request length is pointed at by lpLength. If the incoming asynchronous request is a Read or Lock, the callback function should fill in lpLength to point at the desired length of data to be returned.

SourceAddress - Specifies the 1394 address (6-bit node number and 10-bit bus number) that is requesting the operation.

fulNotificationOption - Is the notification option bit that triggers the notify callback on an asynchronous operation.

lpContext - Points to the application supplied context data.

The application callback returns a RESPONSE_CODE that is used by the miniport driver for the response code (RCODE) in the 1394 response packet.

lpContext – Points to the application context data that is passed to the application callback routine when a notification event occurs.

Required1394Offset – If not equal to NULL, specifies a virtual 1394 address being requested. This is not granted if previous callers have already allocated this 1394 address. When NULL, an arbitrary virtual 1394 address is selected and returned through this parameter. This means that after the call to this API function, this parameter will no longer be NULL.

Note:

The user cannot allocate a 1394 memory address that, when combined with the buffer length, exceeds a 32-bit address.

For example, a user could not allocate 0xFFFA FFFF FE00 and a buffer length of 400H bytes since this would exceed a 32-bit address.

lpAddressesReturned – ONLY returned by the API if *Required1394Offset* is NULL. Points to a location containing the number of addresses being returned in *lp1394Address* (below).

lp1394Address – ONLY returned by the API if *Required1394Offset* is NULL. If this call request completes successfully, points to an array of LARGE_INTEGER (64 bits). This array contains the list of PHYSICAL addresses corresponding to the user's supplied buffer. The low part contains a PHYSICAL address while the high part indicates the number of contiguous bytes at this physical location. This list of PHYSICAL addresses can be converted to quasi 1394 addresses and used by a remote node for DIRECT DMA addressing with Write Block request only. See the LYNXHALScatterLock function for a more detailed description.

NOTE: All requests from remote nodes using the virtual 1394 address contained in the *Required1394Offset parameter will be handled by the API and the application will be notified. If the DIRECT DMA feature is used, the API has no knowledge of the request as these are written straight into the user's buffer.*

Return Status

If successful, a STATUS_SUCCESS code is returned an the application can provide *Required1394Offset* to remote 1394 nodes for them to use in subsequent asynchronous operations. It is the responsibility of the caller to ensure that *lpBuffer* and *lpCallback* remain valid until the mapping is freed with the cls1394FreeAddressRange function.

3.6.2 cls1394FreeAddressRange

Description Frees previously allocated address range

Action This function releases a virtual 1394 address allocated by cls1394AllocateAddressRange.

This API is invoked by submitting an IOCtl IRP with the dwIoControlCode equal to the published value of IOCTL_P1394_CLASS, the function number within the P1394_CLASS_REQUEST being equal to the CLS_REQUEST_FREE_ADDRESS_RANGE, and the request union field filled in with the following structure:

Input

```
struct {
        ULONG                   nAddressesToFree;
        PLARGE_INTEGER      lp1394Address;
} clsFreeAddressRange;
```

Parameters *nAddressesToFree* – Specifies how many virtual 1394 addresses are specified in *lp1394Address.*

lp1394Address – Specifies a pointer to 1394 address(es) to be released. The array pointed at by lp1394Address is an array of LARGE_INTEGER (64 bits), however, only the lower 48 bits of each LARGE_INTEGER entry are inspected. The extra 16 bits of each array element are unused, but are helpful for alignment purposes. These address(es) were returned in a prior successful call to cls1394AllocateAddress.

Return Status A STATUS_SUCCESS code is returned and the virtual 1394 addresses specified in lp1394Address are now invalidated.

3.6.3 cls1394AsyncRead

Description Performs the Asynchronous Read from the 1394 Node

Action This function performs an Asynchronous Read operation to the device specified.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ASYNC_READ, and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        PLARGE_INTEGER    DestinationAddress;
        ULONG             nNumberOfBytesToRead;
        ULONG             nBlockSize;
        ULONG             fulFlags;
        PVOID             lpBuffer;
} clsAsyncRead;
```

Parameters

DestinationAddress – Specifies the P1394 48-bit destination address for this Asynchronous Read operation.

nNumberOfBytesToRead – Specifies the number of bytes to be read from the host/remote 1394 node.

nBlockSize – Is not used, this parameter should be set to 0.

fulFlags - Upper 16 bits are used to specify a “Wait” time different from the 1 second default wait time. This value is in milliseconds.

lpBuffer – Points to a user-allocated memory location for which data is received from the remote node.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned along with the received data placed into the linear address that the *lpBuffer* represents. All other errors are reported using cls1394GetLastError.

3.6.4 cls1394AsyncWrite

Description Performs the Asynchronous Write to the 1394 Node

Action This function performs an Asynchronous Write operation to the device(s) specified.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ASYNC_WRITE, and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        PLARGE_INTEGER    DestinationAddress;
        ULONG             nNumberOfBytesToWrite;
        ULONG             nBlockSize;
        ULONG             fulFlags;
        PVOID             lpBuffer;
} clsAsyncWrite;
```

Parameters

DestinationAddress – Specifies the P1394 48-bit destination address for this Asynchronous Write operation.

nNumberOfBytesToWrite – Specifies the number of bytes to write to the host/remote 1394 node.

nBlockSize – If nonzero, specifies the size of each individual block within the data stream that is written as a whole to the remote node. When this parameter is zero, then the maximum packet size for the speed selected is used in breaking up these write requests.

fulFlags - Upper 16 bits are used to specify a “Wait” time different from the 1 second default wait time. This value is in milliseconds.

lpBuffer – Points to a user-allocated memory location that is transmitted to the remote node.

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.6.5 cls1394AsyncLock

Description Performs the Asynchronous Lock to the 1394 Node

Action This function performs an Asynchronous Lock operation to the device specified. This API is invoked by ***calling*** the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ASYNC_LOCK and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        PLARGE_INTEGER    DestinationAddress;
        ULONG             nNumberOfArgBytes;
        ULONG             nNumberOfDataBytes;
        ULONG             fulTransactionType;
        ULONG             fulFlags;
        ULONG             Arguments[2];
        ULONG             DataValues[2];
        PVOID             lpBuffer;
} clsAsyncLock;
```

Parameters

DestinationAddress – Specifies the P1394 48-bit destination offset for this lock operation.

> **Note:**
>
> Unless the caller specified the 1394 class-driver Device Object, the upper 16 bits are ignored in addressing.

nNumberOfArgBytes – Specifies the number of argument bytes used in performing this Asynchronous Lock operation.

nNumberOfDataBytes – Specifies the number of data bytes used in performing this Asynchronous Lock operation.

fuTransactionType – Specifies which subfunction to use on the remote 1394 node. Currently, only the following operations are valid transaction types:

- ❏ MaskSwap - refer to IEEE 1394 specification for more details
- ❏ CompareSwap - refer to IEEE 1394 specification for more details
- ❏ FetchAdd **-** refer to IEEE 1394 specification for more details
- ❏ LittleAdd - refer to IEEE 1394 specification for more details
- ❏ BoundedAdd - refer to IEEE 1394 specification for more details
- ❏ WrapAdd - refer to IEEE 1394 specification for more details

fulFlags - Upper 16 bits are used to specify a “Wait” time different from the 1 second default wait time. This value is in milliseconds.

Arguments – Specifies that this array contains the arguments used in this Lock operation.

DataValues – Specifies that this array contains the data values used in this Lock operation.

lpBuffer – Points to a buffer that lock data values are returned from the remote node.

Return Status

If the function call is successful, a STATUS_SUCCESS code is returned along with the results of the Lock returned to the location pointed at by *lpBuffer*. All other errors are reported using cls1394GetLastError.

3.6.6 cls1394IsochAllocateBandwidth

Description Allocates isochronous bandwidth

Action This function allocates isochronous bandwidth to be used in subsequent operations.

The 1394 bus driver takes the nMaxBytesPerFrameRequested, rounds up to the nearest quadlet, and adds in the overhead required before making the proper allocation of bandwidth. If the bandwidth allocation was successful, a bandwidth handle is assigned in order to free up bandwidth at some later time.

This function performs an Asynchronous Lock operation to the device specified. This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ALLOCATE_BANDWIDTH and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nMaxBytesPerFrameRequested;
        ULONG           fulSpeed;
        PHANDLE         lpBandwidth;
        PULONG          lpBytesPerFrameAvailable;
        PULONG          lpSpeedSelected;
} clsIsochAllocateBandwidth;
```

Parameters

nMaxBytesPerFrameRequested – Specifies the number of bytes per isochronous frame requested. This value is rounded up to the nearest quadlet and the result is added to the overhead required before the bus driver secures this bandwidth from the isochronous resource manager.

fulSpeed – Specifies the speed flag to use in allocating bandwidth. Current speed flags include:

- ❏ Speed100 - 98.304 Mbit/s
- ❏ Speed200 - 196.608 Mbit/s
- ❏ Speed400 - 393.216 Mbit/s
- ❏ SpeedFastest - Uses the fastest speed that the local transmitter supports

lpBandwidth – Points to field that contains the returned bandwidth handle to be used in releasing bandwidth resources at some later time.

lpBytesPerFrameAvailable – Points to field that contains the bytes per frame that is available after the allocation succeeds or fails. Applications should not count on this bandwidth being available, as another application could have allocated bandwidth after this result is returned.

lpSpeedSelected – Points to the speed that was selected in allocating bandwidth. Possible speed flags returned are:

- ❏ Speed100 - 98.304 Mbit/s

- ❏ Speed200 - 196.608 Mbit/s
- ❏ Speed400 - 393.216 Mbit/s

Return Status

If the function call is successful, a STATUS_SUCCESS code is returned and an isochronous bandwidth is secured. In either case, *lpBytesPerFrameAvailable* is filled in. All other errors are reported using cls1394GetLastError.

3.6.7 cls1394IsochAllocateChannel

Description Allocates isochronous channel number

Action This function allocates an isochronous channel to be used in subsequent operations.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ALLOCATE_CHANNEL and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nRequestedChannel;
        PULONG          lpChannel;
        PLARGE_INTEGER  lpChannelAvailable;
} clsIsochAllocateChannel;
```

Parameters *nRequestedChannel* – Specifies a specific channel requested by the application. If 0xffffffff (-1) is specified, then an arbitrary channel is returned. Hardware should be able to use any channel number (0-63) specified.

lpChannel – Points to the field that contains the returned channel when the allocation of the channel is successful. This channel can be used in subsequent isochronous operations.

lpChannelAvailable – Points to the field that contains a bit mask of the available Isochronous channels after the allocation succeeds or fails. Applications should not count on these channels being available, as another application could have allocated channels after this result is returned.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and an isochronous channel is secured. In either case, *lpChannelsAvailable* is filled in. All other errors are reported using cls1394GetLastError.

3.6.8 cls1394IsochAllocateResources

Description Allocates resources for an isochronous stream

Action This function allocates hardware/software resources associated with a given isochronous stream. Successful hardware/software resource allocation must be coupled with the attachment of buffers (see cls1394IsochAttachBuffers below) before the talk or listen function can be performed on an isochronous stream.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ALLOCATE_RESOURCES and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           fulSpeed;
        ULONG           fulFlags;
        PHANDLE     lpResources;
} clsIsochAllocateResources;
```

Parameters *fulSpeed* – This field contains the requested speed for this resource handle. Current speed flags include:

- ❏ Speed100 - 98.304 Mbit/s
- ❏ Speed200 - 196.608 Mbit/s
- ❏ Speed400 - 393.216 Mbit/s
- ❏ SpeedFastest - Uses the fastest speed that the local transmitter supports

fulFlags – Specifies if the isochronous resource is to be used for Talking or Listening operation.

- ❏ ResourceUsedInListening - Used in listening to an isochronous stream
- ❏ ResourceUsedInTalking - Used in talking to an isochronous stream

lpResources – Points to a field which will contain the returned resource handle to be used in releasing hardware/software resources at some later time.

Return Status If this function call is successful, a STATUS_SUCCESS code is returned and hardware/software resources are secured. All other errors are reported using cls1394GetLastError.

3.6.9 cls1394IsochAttachBuffers

Description Attach Isochronous buffers to a resource

Action This function attaches isochronous buffers to a resource. The buffer and resources must be setup prior to performing any Talk or Listen operation on any isochronous channel.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ATTACH_BUFFERS and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        HANDLE                  hResources;
        DWORD                   Channel;
        PISOCH_DESCRIPTOR       lpIsochDescriptor;
} clsIsochAttachBuffers;
```

Parameters *hResources* – Specifies the resources that this buffer is to be associated with.

Channel - Specifies the channel number to attach this buffer to.

lpIsochBuffer – Points to an isochronous buffer to be used with this resource handle. This descriptor should reside in locked memory as the 1394 driver stack could potentially modify this descriptor at interrupt time. The definition of ISOCH_DESCRIPTOR is as follows:

```
typedef struct _ISOCH_DESCRIPTOR {
        struct _ISOCH_DESCRIPTOR        *Next;
        ULONG                   fulFlags;
        PMDL                    lpBuffer;
        ULONG                   ulLength;
        ULONG                   ulSynchronize;
        ULONG                   ulCycle;
        LARGE_INTEGER           SystemTime;
        PVOID                   lpCallback;
        PVOID                   lpWaterLineCallback;
        DWORD                   ulWaterLine;
        PVOID                   lpContext;
        ULONG                   Status;
        ULONG                   PacketSize;
        ULONG                   ulReserved[4];
} ISOCH_DESCRIPTOR, *PISOCH_DESCRIPTOR;
```

ISOCH_DESCRIPTOR Parameters

_ISOCH_DESCRIPTOR **-** Is a singly linked list of isochronous descriptors. The list is allowed to be circular.

fulFlags - Are bit flags used for synchronizing packet acceptance and packet header removal before moving the data to the user buffer. Valid bit fields are:

FLAG_SYNCHRONIZE	0x01
FLAG_STRIP_HEADER	0x02

This is used to synchronize data collection with the synchronous field in the isochronous header packet.

lpBuffer - This pointer represents a buffer in which the data is to be contained.

ulLength - Contains the length of lpBuffer

ulSynchronize - Is the 4-bit field used to synchronize packet acceptance with the “sy” field of the 1394 isochronous packet header.

ulCycle - Is not used

lpCallback - Specifies the device driver callback addresses (if supplied). In this way applications can be notified when the descriptor has finished being processed.

lpWaterLineCallback - Specifies the device drivers waterline callback address (if supplied). In this way applications can be notified when the waterline data mark has been reached.

ulWaterLine - Specifies the amount of data for the attached buffer to process before the waterline callback is invoked. This is 0-100 percent of the attached buffer.

lpContext - Is user-supplied context parameter to be provided at callback time. This is returned to the callback routine.

Status - Is not used

PacketSize - Specifies the size of the packet to transfer or receive. This value is specified in bytes. If the packet encoder is not stripped off, the packet header size must be included in the packet size.

ulReserved[4] - Is not used

Return Status

If the function call is successful, a STATUS_SUCCESS code is returned and this isochronous buffer is associated with the resource handle. This isochronous buffer must eventually be freed using cls1394IsochDetachBuffers. All other errors are reported using cls1394GetLastError.

Callback Examples

The two callbacks both have the same calling sequence:

- extern "C" void WINAPI MyBuffCompCallback(DWORD Context)
- extern "C" void WINAPI MyWaterLnCallback(DWORD Context)

Call Back Parameters

Context - Is a user supplied value. It is used by the application software to determine which callback has been completed. For example, if the application has attached three separate buffers, the context returned allows the application to determine which buffer has completed processing.

3.6.10 cls1394IsochDetachBuffers

Description Detaches previously attached buffers from a resource

Action This function detaches isochronous buffers previously using the cls1394IsochAttachBuffers function.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_DETACH_BUFFERS and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        HANDLE                  hResources;
        PISOCH_DESCRIPTOR       lpIsochDescriptor;
} clsIsochDetachBuffers;
```

Parameters *hResources* – Specifies the resource handle that this buffer is to be detached from.

lpIsochDescriptor - Is not used

Return Status If this function call is successful, a STATUS_SUCCESS code is returned and the isochronous buffer descriptor is detached from the resource handle specified. All other errors are reported using cls1394GetLastError.

3.6.11 cls1394IsochFreeBandwidth

Description Frees previously allocated isochronous bandwidth

Action This function releases isochronous bandwidth allocated using cls1394IsochAllocateBandwidth.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_FREE_BANDWIDTH and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        HANDLE          hBandwidth;
} clsIsochFreeBandwidth;
```

Parameters *hBandwidth* – Specifies the bandwidth handle to release.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and the isochronous bandwidth is returned to the pool of available bandwidth. All other errors are reported using cls1394GetLastError.

3.6.12 cls1394IsochFreeChannel

Description Frees a previously allocated isochronous channel

Action This function releases an allocated isochronous channel using cls1394IsochAllocateChannel.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_FREE_CHANNEL and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nChannel;
} clsIsochFreeChannel;
```

Parameters *nChannel* – Specifies which allocated channel to release.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and the isochronous channel is returned to the pool of available channels.

3.6.13 cls1394IsochFreeResources

Description Frees prior allocated isochronous stream resources

Action This function releases isochronous hardware/software resources allocated using cls1394IsochAllocateResources. All isochronous buffers that attach to this resource must detach prior to issuing this call. When a device driver attempts to free a resource handle with isochronous buffers still attached to it, the handle is not freed and an error is returned.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_FREE_RESOURCES and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        HANDLE          hResources;
} clsIsochFreeResources;
```

Parameters *hResources* – Specifies the resource handle to release.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and the isochronous hardware/software resources are returned to the pool of available resources. All other errors are reported using cls1394GetLastError.

3.6.14 cls1394IsochListen

Description Begins listening on an isochronous channel

Action This function begins listening on an isochronous channel and the resource handle is specified. Resource allocation and attachment of buffers to this resource handle must have already been done prior to issuing this call.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ISOCH_LISTEN and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nChannel;
        HANDLE          hResources;
        ULONG           fulFlags;
        ULONG           nStartCycle;
        LARGE_INTEGER   StartTime;
        ULONG           ulSynchronize;
        ULONG           ulTag;
} clsIsochListen;
```

Parameters

nChannel – Specifies the channel to listen on.

hResources – Specifies the resource handle to listen on.

fulFlags - Is not used

nStartCycle - Is not used

StartTime - Is not used

ulSynchronize - Is not used

ulTag - Is not used

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.6.15 cls1394IsochStop

Description Stops isochronous operations on a channel

Action This function stops all isochronous operations on an isochronous channel.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ISOCH_STOP and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nChannel;
        HANDLE          hResources;
        ULONG           fulFlags;
        ULONG           nStopCycle;
        LARGE_INTEGER   StopTime;
        ULONG           ulSynchronize;
        ULONG           ulTag;
} clsIsochStop;
```

Parameters

nChannel – Specifies the channel to stop isochronous operations on.

hResources – Specifies the resource handle to stop isochronous operations on.

fulFlags - Is not used

nStopCycle - Is not used

StopTime - Is not used

ulSynchronize - Is not used

ulTag - Is not used

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and the isochronous operation stops. All other errors are reported using cls1394GetLastError.

3.6.16 cls1394IsochTalk

Description Begins talking on an isochronous channel

Action This function begins transmitting data on an isochronous channel. Resource allocation and attachment of buffers to this resource handle must have already been done prior to issuing this call.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ISOCH_TALK and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           nChannel;
        HANDLE          hResource;
        ULONG           fulFlags;
        ULONG           nStartCycle;
        LARGE_INTEGER   StartTime;
        ULONG           ulSynchronize;
        ULONG           ulTag;
} clsIsochTalk;
```

Parameters

nChannel – Specifies the channel on which to talk.

hResource – Specifies the resource handle on which to talk.

fulFlags - Is not used

nStartCycle - Is not used

StartTime - Is not used

ulSynchronize - Is not used

ulTag - Is not used

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.6.17 cls1394IsochQueryCurrentCycleNumber

Description Gets the current cycle number

Action This function returns the current isochronous cycle number.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_ISOCH_QUERY_CYCLE_NUMBER and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        PULONG          lpCycleNumber;
} clsIsochQueryCurrentCycleNumber;
```

Parameters *lpCycleNumber* – Points to the returned current cycle number.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned and the isochronous cycle number is returned as in the 1394-1995 specification. The CYCLE_TIME register is shown in Table 3–1 below. The timer is 32 bits wide. The low-order 12 bits (cycle_offset) counts as a modulo 3072 counter, which increments once every 24.576 MHz (40.69 ns). The next 13 high-order bits (cycle_count) are a modulo 8000 counter, which increments on a carry from cycle_offset. The highest seven bits are a modulo 128 counter, which increments on a carry from cycle_count. All other errors are reported using cls1394GetLastError.

Table 3-1 CYCLE_TIME Register

bits 26 - 32	bits 13 - 25	bits 0 - 12
second_count	cycle_count	cycle_offset

3.6.18 cls1394Get1394AddressFromDeviceObject

Description Get the Node/Bus Number

Action This function returns a 1394 node address given a Device Object.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_1394_ADDRESS and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        PP1394_NODE_ADDRESS     lpNodeAddress;
} clsGet1394AddressFromDeviceObject;
```

Parameters *lpNodeAddress* – If successful, points to the field that contains the 6-bit/10-bit Node Address and Bus Number.

Return Status If the function call is successful, a STATUS_SUCCESS code is returned with the *lpNodeAddress* filled in. All other errors are reported using cls1394GetLastError.

3.6.19 cls1394SetDeviceSpeed

Description Sets the transmission speed when given a Device Object

Action This function sets the speed at which the requests are transmitted to a particular device. By default, the 1394 bus driver has access to a speed map that it uses to determine what the maximum speed is when transmitting to a device. However, when the device driver needs to specify a different speed, it can use this service. This is applicable for asynchronous or isochronous requests.

This API is invoked by calling the 1394DeviceIOCntl function with the dwIoControlCode being equal to CLS_REQUEST_SET_DEVICE_SPEED and the lpInBuffer structure filled in with the following structure:

Input

```
struct {
        ULONG           fulSpeed;
} clsSetDeviceSpeed;
```

Parameters *fulSpeed* –Sets the fastest speed for transmitting requests. Current speed flags include:

- ❏ Speed100 - 98.304 Mbit/s
- ❏ Speed200 - 196.608 Mbit/s
- ❏ Speed400 - 393.216 Mbit/s

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.7 cls1394SendLinkOnPkt

Description Sends Link-On packets to device objects.

Action This function provides the capability to send link-on packets to device objects. See section 4.3.4.2 Link-on packet in the IEEE 1394-1995 specification.

Syntax BOOL cls1394SendLinkOnPkt(cls1394HANDLE hDev);

Parameters hDev - Is the device object handle to which the operation is targeted. This handle is obtained by a call to cls1394CreateFile.

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

Error Codes If the function call is not successful, a call to cls1394GetLastError will return one of two possible error conditions:

CLASS1394_BUS_RESET_IN_PROGRESS (bus reset occurring)

CLASS1394_TRANSMISSION_FAILURE (transmission failure)

3.8 cls1394GetHandleFromNodeID

Description Determines the HANDLE associated with a specific node ID.

Action This function provides provides the capability to send link-on packets to device objects. See section 4.3.4.2 Link-on packet in the IEEE 1394-1995 specification.

Syntax BOOL cls1394GetHandleFromNodeID(WORD NodeID, cls1394HANDLE * hDevice);

Parameters **NodeID - the 16 bit Node ID composed of the 10 bit Bus number and the 6 bit node ID.**
hDevevic - Is a pointer to the location to store the device object handle of the node requested.

Return Status A STATUS_SUCCESS code signifies successful completion of this function. All other errors are reported using cls1394GetLastError.

3.9 GetDeviceInfo

Description Retrieves internal class information about a device on the bus.

Action This function retrieves data from the internal class structures that is being maintained for each device object.

Input

```
typedef struct{
    BOOL    Active;              // True if currently on bus
    WORD    DeviceEntry;         // Entry number for this type device
    WORD    NodeID;              // Current node id on the bus
    ULONG   VendID_DevType;      // Vendor/Device ID from device
    ULONG   SerialNo;            // Serial number from device
    VendorName Vendor;           // Textual vendor name
    ModelName  Model;            // Textual device model name
} DeviceInfo;
```

Syntax BOOL GetDeviceInfo(WORD DeviceEntry, DeviceInfo* sDevInfo);

Parameters *DeviceEntry* – Is a zero based index into the class device list . This value is not related to the DeviceEntry parameter for the cls1394CreateFile function.

sDevInfo - A pointer to the structure to contain the returned data.

Return Status If the function returns TRUE, the data is valid, if FALSE, there are no more devices on the bus. The VendID_DevType and DeviceEntry fields of the structure may be used in the cls1394CreateFile call to open a specific device.

3.10 GetAdapterAddress

Description Returns a pointer to the 1394 host adapter card

Action This function returns a pointer to the PCI host-adapter card. The pointer can be cast using the structure defined in the file PCILYNX.H. The base structure is defined below showing sub structure fields that may not be complete.

Syntax PVOID GetAdapterAddress();

Example

```
typedef struct {
union {
        QUADLET LynxArr[ 0x1000 / 4 ];
        struct {
        LynxPCICfgSpace         PCIRegs;
        LynxAuxPortRegs         AUXRegs;
        LynxDMACtrlRegs         DMARegs;
        LynxFIFORegs            FIFORegs;
        LynxLLCRegs             LLCRegs;
} PCILynxRegStruct, *pPCILynxRegStruct;
```

Parameters

LynxArr – Is an array of quadlets that maps over all PCILYNX registers

LynxPCICfgSpace - Is a struct that maps the peripheral component interface (PCI) Configuration registers (000 - 03C)

LynxAuxPortRegs - Is a struct that maps the auxiliary (AUX)-port registers (040 - 0FC)

LynxDMACtrlRegs - Is a struct that maps the direct-memory access (DMA)-control registers (100 - 9FC)

LynxFIFORegs - Is a struct that maps the first-in, first-out (FIFO) control registers (A00 - AFC)

LynxLLCRegs - Is a struct that maps the link-layer control registers (B00 - FFF)

3.11 bReadPhyReg

Description Retrieves the requested PHY register.

Action This function retrieves data from the link layer controller phy access register. It will return the 8 bit phy register requested.

Syntax BOOL bReadPhyReg(DWORD dwReadPhy, DWORD *pdwParam);

Parameters *dwReadPhy* – The desired phy register to be read (ie 0, 1, .. 7).

pdwParam - A pointer to store the data read from the phy.

Return Status If the function returns TRUE, the data is valid, if FALSE, there was an error reading the phy register.

3.12 CrcCalculate

Description Calculate 16-bit CRC value for given data.

Action This function computes the 16-bit CRC value over the given data quadlets per the IEEE 1394-1995 specification. This functions provides both the 1991 calculation as well as the 1994 corrected version of the algorithm. Some devices may still exist that use the 1991 version. See section 6.2.4 Primary Packet Components in the IEEE 1394-1995 specification.

Syntax WORD CrcCalculate(DWORD* pQuadData, int nNumQuads, BOOL CRC1991);

Parameters *pQuadData* – A pointer to the quad array containing the data to compute the CRC value over..

nNumQuads - The number of elements in the array of data.

CRC1991 - If false, calculate CRC based on 1994 spec, if true, calculate CRC based on 1991 spec..

Return Status Returns the calculated 16-bit CRC value.

3.13 ctDelay

Description Provides a delay using the Cycle Timer of the link layer controller.

Action This function provides a delay based on ISO cycles or 125 usec time periods.

Syntax void ctDelay(ULONG N_x_125usecs);

Parameters *N_x_125usecs* – The number of ISO cycles to delay.

Return Status No return status available.

3.14 ctMillisec

Description Provides a delay using the Cycle Timer of the link layer controller.

Action This function provides an accurate delay in Millisecond intervals.

Syntax void ctMillisec(ULONG Millisecs);

Parameters *Millisecs* – The number of milliseconds to delay.

Return Status No return status available.

3.15 ctMicrosec

Description Provides a delay using the Cycle Timer of the link layer controller.

Action This function provides an accurate delay in Microsecond intervals.

Syntax void ctMicrosec(ULONG Microsecs);

Parameters *Microsecs* – The number of microseconds to delay.

Return Status No return status available.

3.16 LYNXHALScatterLock

Description Locks the user's buffer into physical memory and retrieves the list of PHYSICAL addresses associated with the memory buffer.

Action This function provides the list of PHYSCIAL addresses and lengths associated with a caller's memory buffer.

Syntax

```
BOOL LYNXHALScatterLock( LPVOID buffer, DWORD length,
                         PULONG lpAddressesReturned, LPVOID lp1394Address );
```

Parameters

buffer - the memory buffer being requested to be mapped to PHYSICAL memory locations.

length - Is the length in bytes the memory buffer occupies.

lpAddressesReturned – Address of a location to contain the number of addresses being returned in *lp1394Address* (below).

lp1394Address – If this call request completes successfully, points to an array of LARGE_INTEGER (64 bits). This array contains the list of PHYSICAL addresses corresponding to the user's supplied buffer. The low part contains a PHYSICAL address while the high part indicates the number of contiguous bytes at this physical location. This list of PHYSICAL addresses can be converted to quasi 1394 addresses and used by a remote node for DIRECT DMA addressing with Write Block request only.

To convert the PHYSICAL address to a virtual 1394 address to be used for DIRECT DMA, simply set the upper 16 bits of the 48 bit 1394 address to 0x0000 and use the 32 bit physical address for the lower 32 bits. The length would then imply the size of the buffer at this virtual 1394 address.

Return Status If TRUE, signifies successful completion of this function. All other errors are reported using cls1394GetLastError. Possible errors are:

ERROR_OUT_OF_MEMORY
ERROR_SCATTERLOCK_FAILED

3.17 LYNXHALScatterUnLock

Description Unlocks the user's memory buffer from physical memory.

Action This function frees the user's memory from being locked down in physical memory. It undoes the effects of LYNXHALScatterLock.

Syntax BOOL LYNXHALScatterUnLock(LPVOID lpAddrList);

Parameters *lpAddrList* - The array of physical addresses that was returned by the call to LYNXHALScatterLock.

Return Status If TRUE, signifies successful completion of this function.

3.18 LYNXHALPageAllocBuffer

Description Obtains memory that is physically contiguous in memory.

Action This function provides the capability to allocate large chunks of memory that are physically contiguous in memory. This is useful for creating large buffers to be used in DIRECT DMA accesses by remote nodes.

Syntax

```
BOOL LYNXHALPageAllocBuffer( VOID** buffer, DWORD length,
                             PHANDLE memHandle, DWORD* PhysAddr );
```

Parameters

buffer - The address to store the pointer to the newly allocated memory.

Length - The length in bytes of memory to be allocated.

MemHandle - The handle associated with this memory buffer.

PhysAddr - The PHYSCIAL address associated with the starting location of the memory allocated.

Return Status If TRUE, signifies successful completion of this function.

3.19 LYNXHALPageFreeBuffer

Description Releases memory that was obtained through a call to LYNXHALPageAllocBuffer.

Action This function provides the capability to free up the memory that was allocated by a call to the LYNXHALPageAllocBuffer function.

Syntax BOOL LYNXHALPageFreeBuffer(HANDLE memHandle);

Parameters *MemHandle* - The handle associated with the memory buffer to be freed and was returned by a call to LYNXHALPageAllocBuffer.

Return Status If TRUE, signifies successful completion of this function.

Installation

4 Installation

The installation procedure for the LynxSoft API software is defined below. Additional information is contained in the readme.txt file supplied with this software.

NOTE: Do not run both versions of the code. LynxSoft versions 1.x and Version 2.x are not compatible with each other.

IMPORTANT

- If you are currently a user of LynxSoft 1.0 or 1.1, you must do the following steps before installing LynxSoft 2.0.

    1. Open up the system.ini file in the Windows directory. Remove the following line from the [386Enh] section:

        [386Enh]
        device=c:\lynxsoft\pcilynx.vxd

        Where \lynxsoft\ is the path to where you put the PCILynx.VxD file.

    2. Go into the Win95 Device Manager (Start | Control Panel | System, Device Manager tab). Make sure the “View by Device Type” radio button is selected. Double-click on the “Other Devices” section (with the question mark icon) to display all devices in that section. The PCILynx card will show up as “PCI Card”. Click on “PCI Card” to highlight it, and then click on “Remove” to remove the card from the system.

    3. Click the OK button to close the System Properties dialog box.

Software Installation Procedure

1. If you have installed a previous version of LynxSoft, then execute Windows Explorer and delete the LynxSoft directory. Also be sure that you remove the device statement in the SYSTEM.INI as explained above. Bring up a DOS box on the screen display.

2. Insert disk 1 into the floppy drive. If down loaded from the WEB, go to next step.

3. Select the "Run" command off the WINDOWS "Start" menu.

4. Execute the “SETUP" program from the floppy disk or in the directory that you downloaded the code to install the LynxSoft files. This will create the LynxSoft directory structure.

C:\LYNXSOFT - Test Application runtime files
C:\LYNXSOFT\1394DOC - Documentation provided
C:\LYNXSOFT\1394TST - Test App source and build files
C:\LYNXSOFT\1394TST inc\1394api.h - LynxSoft API interface file.

Hardware / Driver Installation Procedure

1. If you have already installed the PCILynx EVM card in your system and told Win95 there was no driver for the card when prompted, you must remove the card from the Win95 system.

   To remove the card from the system, Go into the Win95 DeviceManager (Start | Control Panel | System, Device Manager tab). Make sure the "View by Device Type" radio button is selected. Double-click on the "Other Devices" section (with the question mark icon) to display all devices in that section. The PCILynx card will show up as "PCI Card". Click on "PCI Card" to highlight it, and then click on "Remove" to remove the card from the system.

2. Shut down Win95, power off the machine, and insert the PCILynx EVM card into any available PCI slot. If the EVM kit contents list includes a power cable, then you should have a TSBKPCI card that requires external power. Make sure to plug the power cable provided into the PCILynx EVM card and into an available lead from your PC's power supply. If the kit contents list does not include a power cable then the TSBKPCI card is powered from the PCI bus and external power is not required. Power up the machine.

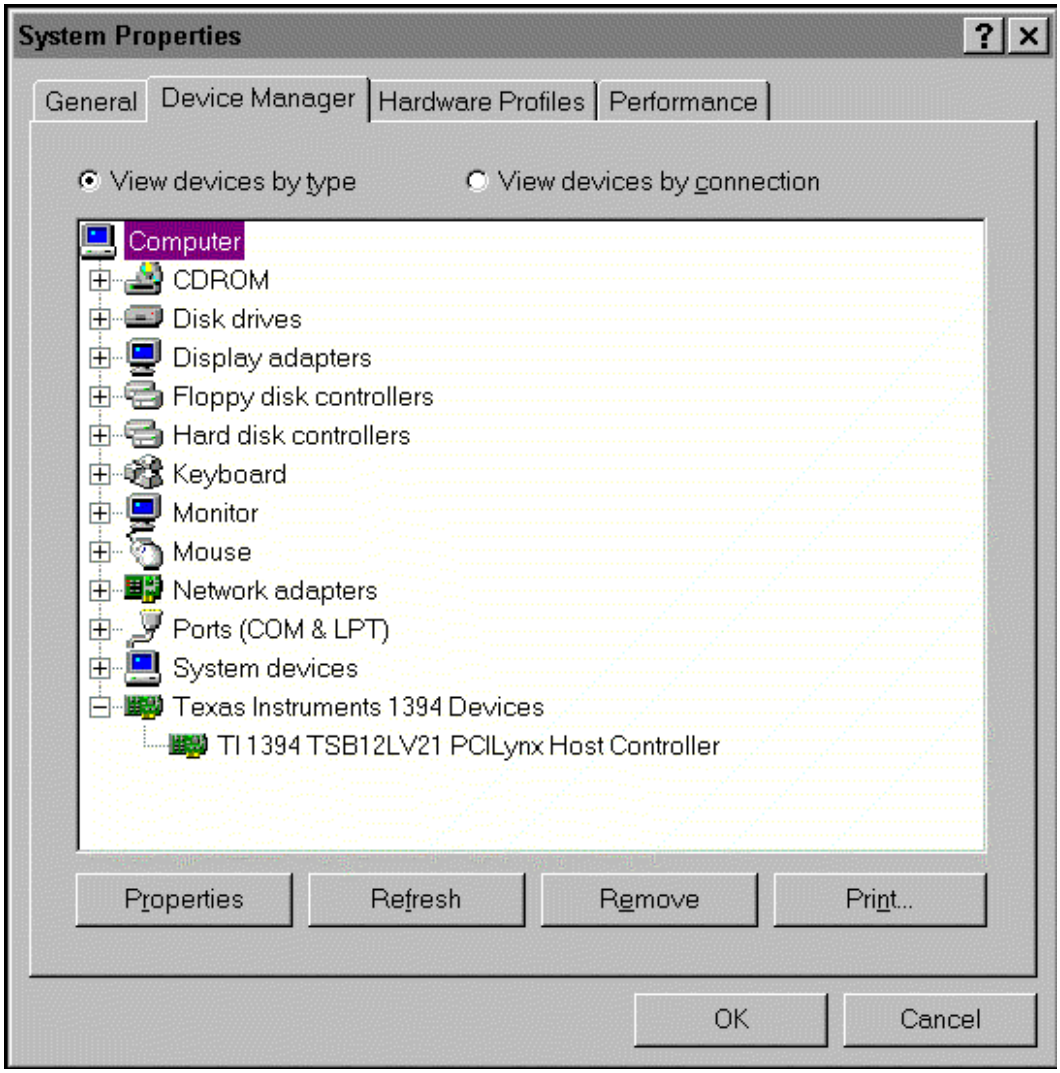
3. During boot, windows will find the PCILynx card in the system. When it does, it will present the following dialog box.

4. Select "Driver from disk provided by hardware manufacturer" and click "OK"

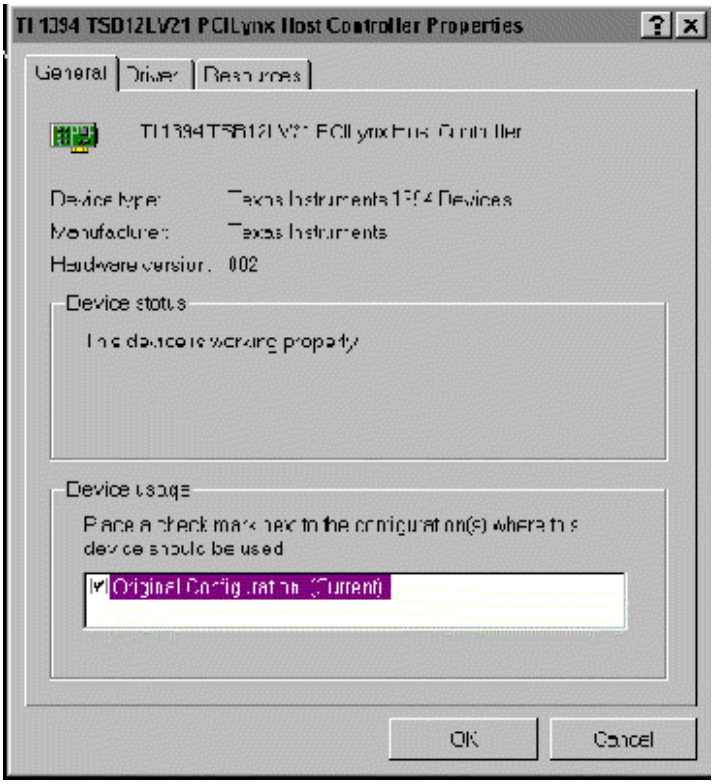
5. Select "Browse" and go to the directory "C:\LYNXSOFT". The filename "PCILYNX.INF" should appear, select and click "OK".

6. Win95 will then copy PCILynx.VxD to the windows system directory and fill in the windows system registry as needed.

7. Win95 will prompt you to reboot your PC to complete the installation, choose “Yes” to reboot your PC. Win95 will now load the PCILynx driver on boot. Upon successful installation, the Device Manager should show the PCILynx EVM as below.



And double-clicking on the PCILynx device should show a working status as shown in the figure below.


TI 1394 TSB12LV21 PCILynx Host Controller Properties
General
Driver
Resources
Device type:
Manufacturer:
Texas Instruments
Hardware version:
002
Device status
This device is working properly.
Device usage
Place a check mark next to the configuration(s) where this device should be used
Original Configuration (Current)
OK
Cancel

Test Application

5 Test Application

The LynxSoft diskette contains a test application for use as well as the source code for that application. It is a Windows application that exercises all of the API functions. Some discussion of what happens initially with the application helps when using it the first time.

In the LynxSoft-to-LynxSoft test application, both applications must be running to have their CSR space enabled. Therefore, as each LynxSoft application is brought up and tries to enumerate the 1394 bus, it may or may not see the other LynxSoft application as a 1394-compliant device. In this case it declares the other LynxSoft application to be a non-compliant device and only allows it to be opened by using the non-compliant utilities of the 1394CreateFile function. To ensure that both applications can "see" the other application as a compliant device, each application should be brought up and then the first application restarted independently of each other. This allows the LynxSoft API to recognize the other LynxSoft application as a compliant device.

The test utility only allows communication with one device object at a time. Therefore, if the user has LynxSoft applications running, to switch from one target to another, the user would have to activate the window of the device object or open a window to the device from the device selection dialogue.

To use the isochronous portion of the test utility the computer must be in 16-bit, 65000 color mode. This is due to the transferred data being in YUV format and is converted to 16bit RGB before it is output to the screen.

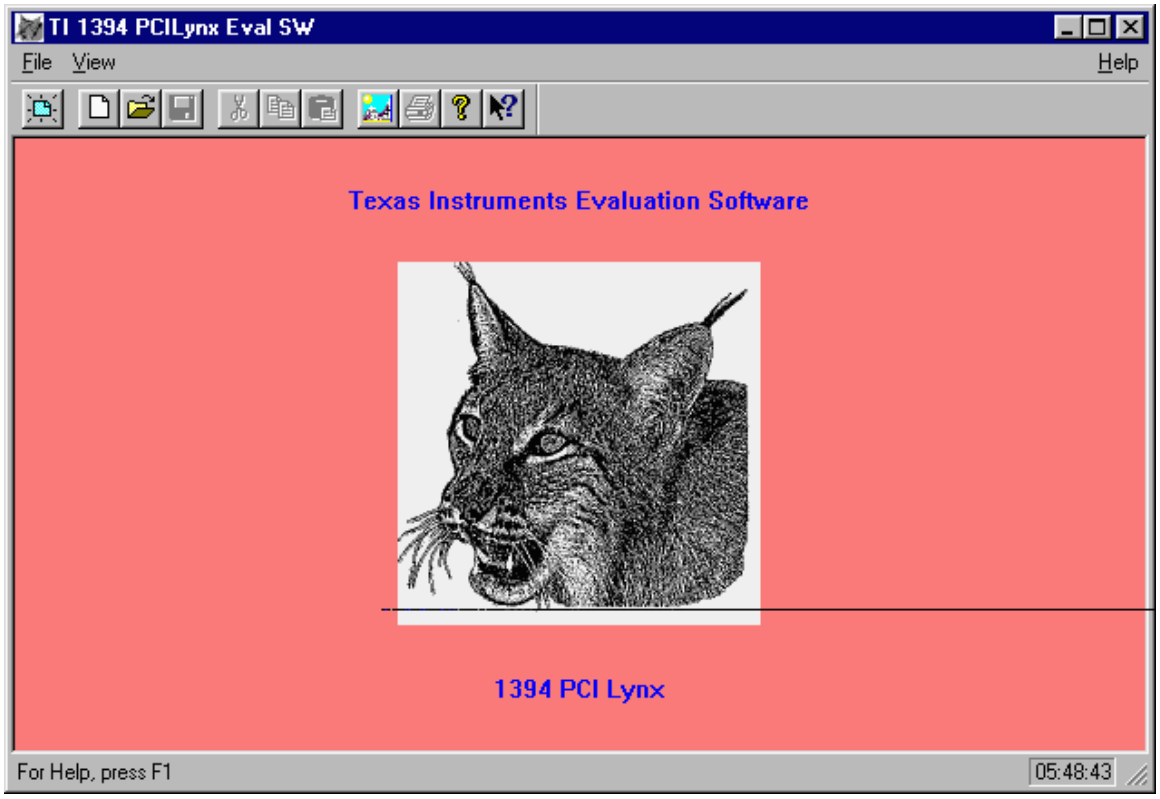
Note:

The generated callbacks can stack up when the computer cannot perform the RGB conversion and output to the screen in a timely manner. The system should be a Pentium-class machine running at 90 MHz (preferably 133 MHz) to allow this software to keep displaying data at 30 frames per second. If the computer cannot keep up with the speed, ultimately the operating system (OS) stacks up too many requests and hangs.

5.1 Test Utility Controls and Dialog Boxes

The main test utility menu contains the File, View, and Help pulldown menus as shown in Figure 5-1 if there are no devices opened by the application. The following paragraphs gives a short explanation of the functionality of these menus.

Figure 5-1 Test Application Main Window



5.1.1 File Menu

5.1.1.1 New

This menu item will create an empty Text Document window. This is equivalent to the “New Document” toolbar button.

5.1.1.2 Open

This menu item will open a file and create Text Document window. This is equivalent to the “File Open” toolbar button.

5.1.1.3 Device

This menu item will call the Device Selection Dialog and allow you to open a connection to a device. This will create a Text Document window for messages. This is equivalent to the “New Device” toolbar button.

5.1.1.4 Print Setup

This menu item will call the Printer Setup Dialog to allow you to set printer options.

5.1.1.5 Exit

This menu item will Exit the application closing any open devices.

5.1.2 View Menu

5.1.2.1 Tool bar

This menu item will hide/display the toolbar as shown below.

5.1.2.2 Status bar

This menu item will hide/display the status bar as shown below.

5.1.3 Help Menu

5.1.3.1 Help Topics

This menu item will open the help engine.

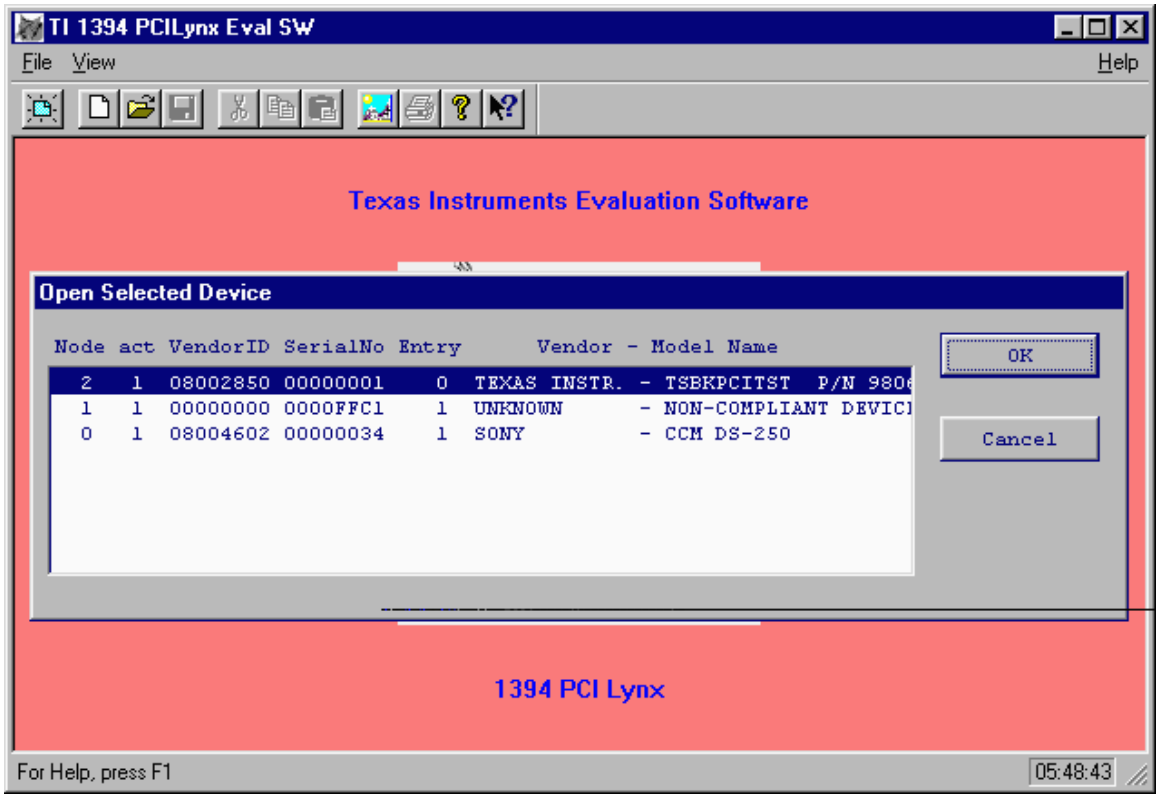
5.1.3.2 About

This menu item displays the version information about the test code.

5.2 Device Selection Dialog

This device selection dialog will appear as shown in Figure 5-2 when the App is run and devices are available on the bus, or when the File|Device menu option is chosen or the Device toolbar button is selected. This will establish a connection to the selected device and provide a Text Document window associated with the device for messages. The active window is the device all menu options are applied to.

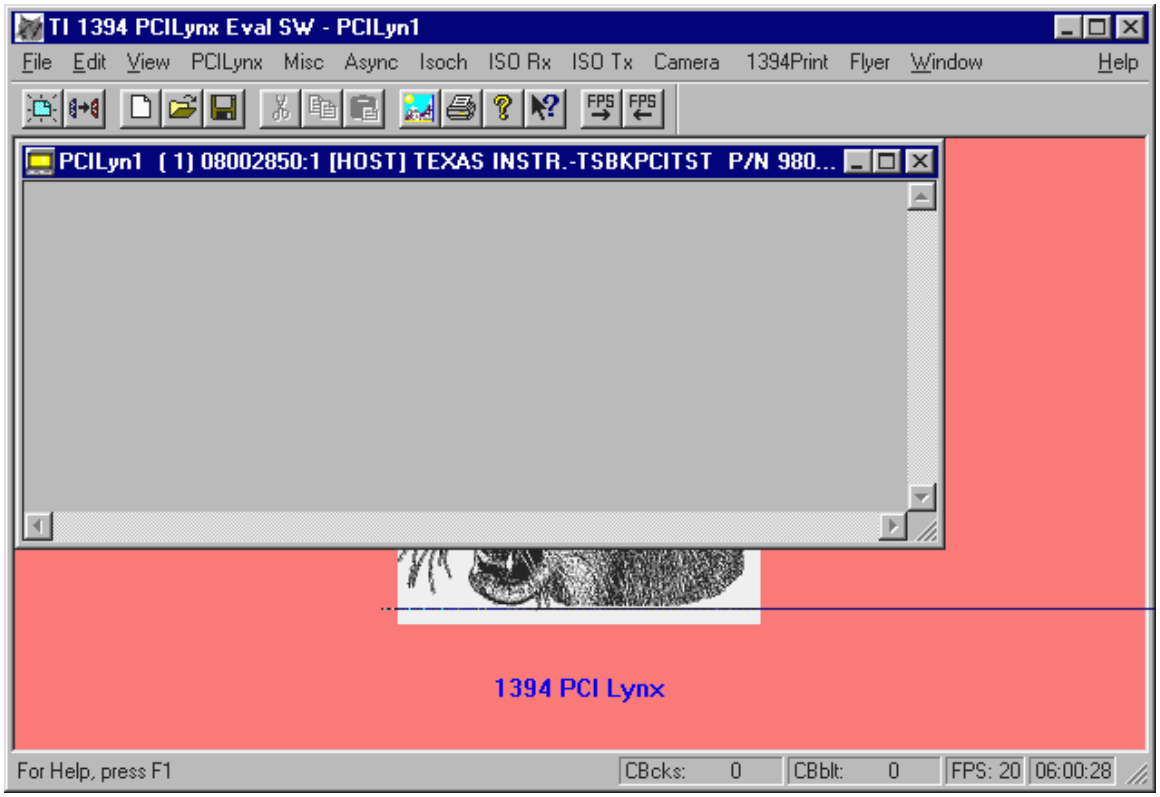
Figure 5-2 Device Selection Dialog



5.3 Host Device Selected

The App screen will appear as in Figure 5-3 when the HOST device is selected from the Device Selection dialog or if no devices are attached to the bus when the App is brought up. All menus for other devices are a subset of the HOST menu

Figure 5-3 Host Device



5.3.1 File Menu

5.3.1.1 Device | New

This menu item will call the Device Selection Dialog and allow you to open a connection to a device. This will create a Text Document window for messages. This is equivalent to the “New Device” toolbar button.

5.3.1.2 Device | Link to

This menu item will allow you to link to any device. This provides all menu options of the HOST device for the “linked” device. All actions will be applied to the node that has been linked to. To resume actions on the HOST, simply perform the “link” operation again on the HOST device. This is equivalent to the “Link Device” toolbar button.

5.3.2 View Menu

5.3.2.1 Color

This menu item will allow you to change the color of the background for the active window.

5.3.2.2 Video (565/555)

This menu item will change the color table mapping from a 555 to a 565 mode. This is needed if your video driver interprets 16 bit RGB as 565 instead of the default mode of 555. The video will look grainy with green and black flecks

5.3.2.3 Beta (2-3/4...)

This menu item will allow you to change the Video “Offset” for the color table mapping. The early Beta cameras from SONY sent data with a different offset. This may be needed if the video looks fluorescent.

5.3.3 PCILynx Menu

5.3.3.1 PCI Regs

This menu item will display the PCI Configuration registers for the TSBKPCI card in the document window.

5.3.3.2 AUX Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.3 DMA Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.4 FIFO Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.5 LLC Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.6 PHY Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.7 ALL Regs

This menu item will display all of the PCILynx registers in the document window.

5.3.3.8 Bus Reset

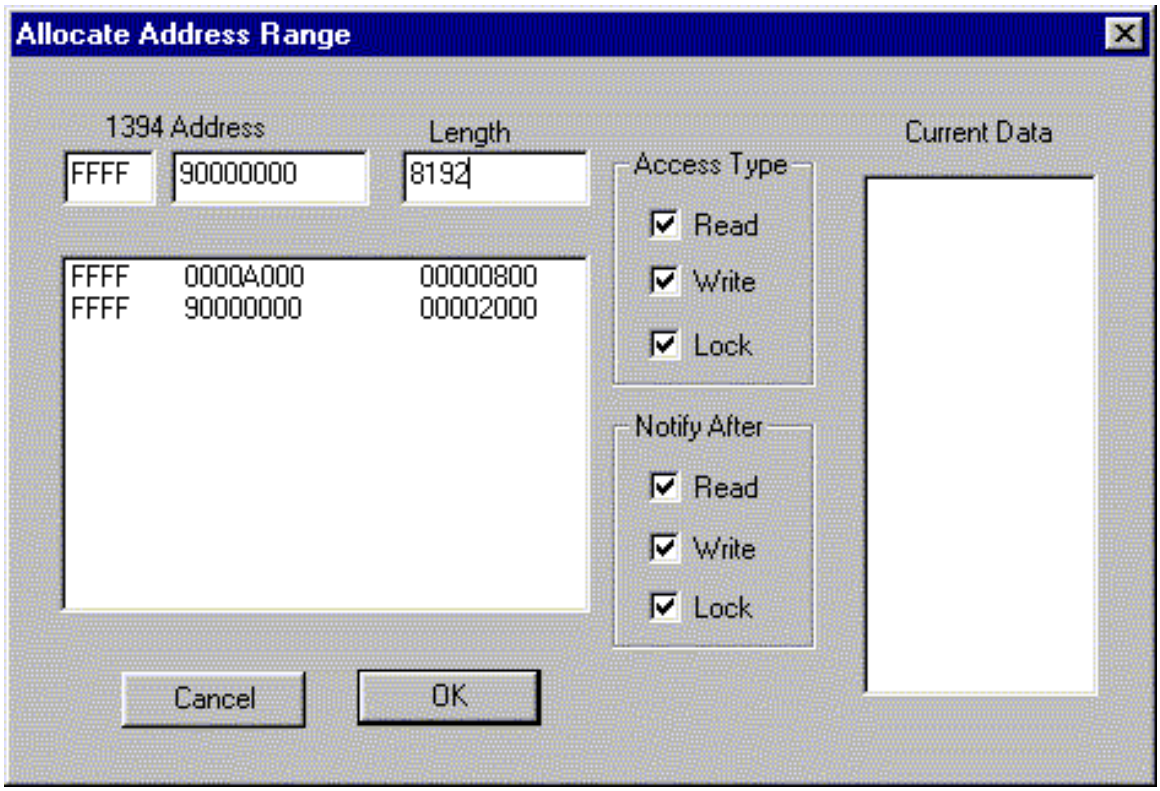
This menu item will force a HW bus reset to occur on the bus.

5.3.4 Misc Menu

5.3.4.1 Address Range | Allocate

This menu item brings up a dialog box as shown in Figure 5-4 that allows the user to allocate an address range that an external 1394 device can access. This dialog box allows the user to specify the 1394 address, the buffer length, the access type and the notification method for the operation. A list of currently allocated ranges will appear in the area below the address/length entry fields. By double clicking on a listed range the data for the range will appear in the list box on the right hand side of the dialog. The API call used for this menu item is defined in section cls1394AllocateAddressRange. HOST only.

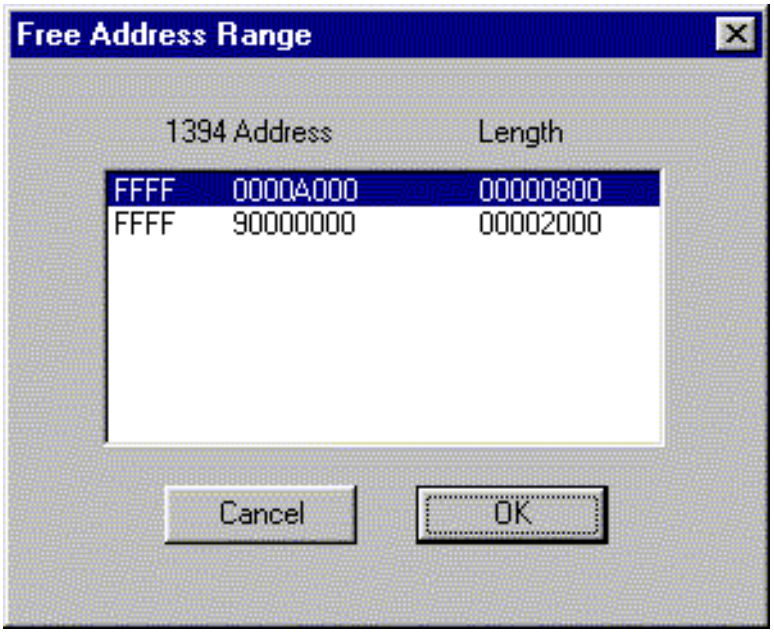
Figure 5-4 Allocate Address Range Dialog



5.3.4.2 Address Range | Free

This menu item brings up a dialog box as shown in Figure 5-5 that allows the user to de-allocate an address range that was previously allocated. HOST only.

Figure 5-5 Free Address Range Dialog



5.3.4.3 Query Cycle Number

This menu item returns the current 1394 cycle number. HOST only.

5.3.4.4 Get 1394 Address

This menu item returns the 1394 node number of the current device object.

5.3.4.5 Set Speed

This menu item allows you to set a lower speed when communication with a remote node.

5.3.5 Async Menu

5.3.5.1 Quadlet

This menu item allows the user to perform quadlet reads and writes to remote 1394 devices. The user is allowed to enter the 1394 address and a field is provided for the data. The address may either be read or written. If the HOST device is selected, this will cause a read or write to HOST allocated memory.

5.3.5.2 Block

This menu item allows the user to perform block reads and writes to remote 1394 devices. The user is allowed to enter the 1394 address, a data length and a field is provided for the data. The addresses may either be read or written. If the HOST device is selected, this will cause a read or write to HOST allocated memory

5.3.5.3 Lock

This menu item allows the user to perform lock functions on 1394 devices. The user is allowed to enter the 1394 address, the lock-transaction type, the number of argument bytes, the number of data bytes, and fields that are provided for entering the lock arguments and data values. The API calls used for this menu item is cls1394AsyncLock in Section 3.12.

5.3.5.4 Test Block

This menu item brings up a dialog to allow you to send pre-defined async block packets. You give the starting 1394 address for the first block, the number of blocks to send and the number of quadlets per packet. Each async packet will have the block number in the first quad of the packet, each successive quadlet will contain the quadlet number base 0, and the final quadlet will contain 0xDEADBEEF.

5.3.5.5 Broadcast

This menu item will allow you to send async quadlet or block "broadcast" packets. **Currently not implemented.**

5.3.6 Isoch Menu Options

5.3.6.1 Allocate | Bandwidth

This menu item allows the user to allocate bandwidth from the bus manager. The user is allowed to select a speed and the number of bytes of bandwidth desired. This function returns a bandwidth handle. The user should save this handle for use when the bandwidth is freed.

5.3.6.2 Allocate | Channel

This menu item allows the user to allocate an isochronous channel from the bus manager. The user is allowed to select a channel number desired.

5.3.6.3 Allocate | Resources

This menu item allows the user to allocate isochronous resources. The user is allowed to specify whether the resources are send or receive resources and the speed desired.

5.3.6.4 Free | Bandwidth

This menu item allows the user to free a previously allocated bandwidth. The user should input the bandwidth handle previously allocated.

5.3.6.5 Free | Channel

This menu item allows the user to free a previously allocated channel.

5.3.6.6 Free | Resources

This menu item allows the user to free previously allocated isochronous resources. The user is asked to enter whether the resources were send or receive resources. The resource handle is imbedded in the application and is not required for this call.

5.3.6.7 Buffers | Attach

This menu item allows the user to attach isochronous buffers to an isochronous channel. The user is allowed to set the direction for the buffer, the buffer type (linear or circular), the isochronous flags that allow the headers to be stripped, and synchronize with the synchronous field and set the watermark for this buffer. Also the isochronous channel, number of buffers, buffer size and packets per buffer are input. The API call for this menu item is cls1394IsochAttachBuffers in Section 3.16.

5.3.6.8 Buffers | Detach

This menu item allows the user to detach previously allocated isochronous buffers.

5.3.6.9 Listen

This menu item begins listening on an isochronous channel. Application callbacks begin occurring and data begins to be transferred to isochronous buffers already allocated and attached.

5.3.6.10 Talk

This menu item begins the talking on an isochronous channel. Application callbacks begin occurring and data begins to be transferred using an isochronous channel to a remote node.

5.3.6.11 Stop

This menu item stops all isochronous transmission.

5.3.7 ISO Rx Menu

5.3.7.1 Lynx->Lynx

This menu item performs all of the necessary function calls to begin receiving isochronous data from another PCILynx. The data transmitted is a frame of video captured from a Sony desktop camera or a generated color bar pattern. The data is received, converted, and transmitted to the user screen.

5.3.7.2 Stop

This menu item halts the reception of isochronous data from an external lynx.

5.3.7.3 Snapshot

This menu item will take a snapshot of the current image being displayed and send it to the printer. If the printer output is directed to “FILE:”, then the output will go across the bus to the Epson printer. If the output is directed to the default windows printer, output will be sent out the parallel bus to the default printer.

5.3.8 ISO Tx Menu

5.3.8.1 Still Image

This menu item performs all of the necessary function calls to begin transmitting isochronous data to another PCILynx. The data transmitted is a frame of video captured from a Sony desktop camera. Two files, ISODATA.1 or ISODATA.2 , are provided. The .1 file contains 3 frames of data and the .2 contains 2 frames of data.

5.3.8.2 Color Bar

This menu item performs all of the necessary function calls to begin transmitting isochronous data to another PCILynx. The data transmitted is a vertical color bar pattern.

5.3.8.3 Stop

This menu item halts the transmission of isochronous data to an external PCILynx card.

5.3.9 Camera Menu

5.3.9.1 ON

This menu item sends an ASYNC command that turns on the Sony desktop camera to transmit isochronous data across the bus.

5.3.9.2 Rx

This menu item performs all of the necessary function calls to begin receiving data from a Sony desktop camera. The data is received, converted, and transmitted to the user screen.

5.3.9.3 Stop

This menu item halts the reception of isochronous data from an external camera.

5.3.9.4 Auto WB

This menu item will do a white balance operation on a SONY ds250 camera. This function improves the quality of the image being seen.

5.3.9.5 Controls

This menu item will provide a set of control buttons so that the cameras focus, hue, brightness and so forth could be manually controlled. **Currently not implemented.**

5.3.9.6 Snapshot

This menu item will take a snapshot of the current image being displayed and send it to the printer. If the printer output is directed to “FILE:”, then the output will go across the bus to the Epson printer. If the output is directed to the default windows printer, output will be sent out the parallel bus to the default printer.

Configuration ROM

6 Configuration Rom

The configuration ROM installed in the Texas Instruments evaluation cards is described in Table 6–1. This ROM configuration may or may not be implemented in actual ROM, it can be implemented as a software service but is transparent to the user or remote node.

CSR ROM Description for Texas Instruments 1394 card

The offsets below are added to the start of the CSR ROM offset 0x10 when actually written to the serial EEPROM.

The last hex quad address is 0xFC minus 0x10 from the starting offset, which means that the last possible quad address in this file is 0EC.

> **Note:**
>
> QUADLET is big endian to match specification and makes the ASCII strings appear more readable.

Table 6-1 CSR ROM Values

	Offset	0 - 7	8 - 15	16 - 23	24 - 31	Comments
Bus Info Block	400h	04h	04h	rom crc value		
	404h	31h	33h	39h	34h	'1394'
	408h	1 1 1 1 0h	64h	70h	00h	
	40Ch	08h	00	28	50 51	TSBKPCITST TSBKPCI
	410h	00h	00h	xx	xx	xxxx = Serial #

Root Directory	414h	00h	09h	xx	xx	xxxx = CRC
	418h	03h	08h	00h	28h	Module Vendor ID
	41Ch	81h	00h	00h	09h	Textual Descriptor
	420h	0Ch	00h	02h	00	Node_Capabilities
	424h	8Dh	00h	00h	0Eh	Node_Unique_ID
	428h	C7h	00h	00h	10h	Module_Independent_Info
	42Ch	04h	00h	00h	00h	Module_Hardware_Version
	430h	81h	00h	00h	26h	Textual_Descriptor
	434h	09h	00h	00h	00h	Node_Hardware_Version
	438h	81h	00h	00h	26h	Textual_Descripton

Table 6-1 CSR ROM Values (continued)

	Offset	0-7	8-15	16-23	24-31	Comments
Leaf 1	43Ch	00h	08h	xx	xx	Leaf Len, xxxx = Leaf CRC
Module	440h	00h	00h	00h	00h	
Vendor	444h	00h	00h	00h	00h	
Id	448h	54h	45h	58h	41h	“Texas Instruments”
Textual	44Ch	53h	20h	49h	4Eh	
Descriptor	450h	53h	54h	52h	55h	
	454h	4Dh	45h	4Eh	54h	
	458h	53h	00h	00h	00h	
Leaf 2	45Ch	00h	02h	xx	xx	Leaf_Len, xxxx = Leaf CRC
	460h	08h	00h	28h	01h	Node_Vendor_ID, Chip_ID_Hi
	464h	00h	00h	00h	00h	Chip_Id_Lo
	468h	00h	06h	xx	xx	Dir_len, Dir_Crc
Dir. 1	46Ch	B8h	00h	00h	06h	TI_Module_Name
Module	470h	81h	00h	00h	04h	Textual Descriptor
Dependent	474h	39h	00h	40h	00h	TI_SRAM_QUADS
Info.	478h	3Ah	00h	40H	00h	TI_AUXRAM_QUADS
	47Ch	3Bh	00h	00h	00h	TI_AUX_DEVICE
	480h	00h	05h	xx	xx	leaf_len, xxxx = leaf_crc
Dir 1	484h	00h	00h	00h	00h	
Leaf 1	488h	00h	00h	00h	00h	
TI	48Ch	54h	53h	42h	31h	“TSB12LV21”
Module	490h	32h	4Ch	56h	32h	
Name	494h	31h	00h	00h	00h	
	498h	00h	06h	xx	xx	leaf_len, xxxx = leaf_crc
Dir 1	49Ch	00h	00h	00h	00h	
Leaf 2	4A0h	00h	00h	00h	00h	
Part	4A4h	39h	38h	30h	36h	“980600x-0001”
Number	4A8h	30h	30h	34h	2Dh	
	4ACh	30h	30h	34h	31h	
	4B0h	20h	xxh	xxh	xxh	Revision
Dir 1	4B4h	00h	05h	xx	xx	leaf_len, xxxx = leaf_crc
Leaf 3	4B8h	00h	00h	00h	00h	
Module	4BCh	00h	00h	00h	00h	
Hardware Version	4C0h	54h	53h	42h	4Bh	“TSBKPCITST”, ”TSBPKPCI”
Textual	4C4h	50h	43h	49h	54h	
Descriptor	4C8h	53h	54h	00h	00h	
Dir 1 Leaf 4	4CCh	00h	05h	xx	xx	leaf_len, xxxx = leaf_crc
Node	4D0h	00h	00h	00h	00h	
Hardware	4D4h	00h	00h	00h	00h	
Version	4D8h	54h	53h	42h	32h	“TSB21LV03”
Textual	4DCh	31h	3Ch	56h	30h	
Descriptor	4E0h	33h	00h	00h	00h	

Errata

7 Errata

Table 7–1 is a list of un-implemented functions/limitations of the current software suite along with a schedule for incorporation.

Table 7-1 Errata

Item	Description	Schedule for incorporation
Packets from a MULTIBLOCK transmit	(Class/BusMgr), ie do all blocks have same tLabel or different. Currently VxD sends all packets w/same tLabel if block exceeds max size.	
BusMgr not react to access on CSR space.	BusMgr needs to implement action to some CSR space Some addresses are not implemented.	
BusMgr could send pkts to large for a node to handle.	Reorganize code so that the BusMgr performs the enumeration of devices so that it can track the "max_rec" field of each node. This is needed to send "response" pkts back to a node.	
Attach Buffer call fails	Need a function in the VxD to allow it to clear all "Allocated" PCLs. Used by Class/BusMgr when initializing for 1st time. This would be used if APP crashes and does not free resources.	
Transaction time out for a node.	SPLIT_TIMEOUT register needs to be implemented. CLASS needs to add this info to the device object (read, wait up to 8secs) and use this on ASYNC read/writes to the device.	
Async Tx not ended when bus reset.	Async Tx not terminated by DMA control when bus reset occurs. Need to check for this and terminate the process. This is internal to VxD, just queues events that won't process.	
Need to be able to report ISO Tx fails	Need to report a packet failure on ITF underflow.	
Hang looking for Top/Spd map data	Timing issue when class enumerates and looking for Top/Spd maps. IsoMgrId might be invalid, never get response.	
Busy Off Packets	Need GR PCLs to set busy bit when queue is full.	
Parser	Need parser for Config Rom of remote nodes	
Gapcount est.	Add Gapcount est. when generate speedmap and determine maximum number of hops between nodes.	