LynxSoft™ 1394 Software Application Programmer User's Guide

SLLU003
October 1996

Texas Instruments

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.

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

LynxSoft and TI are trademarks of Texas Instruments Incorporated.
Sony is a trademark of the Sony Corporation.

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.

Contents

Tables

Chapter 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.

Chapter 2

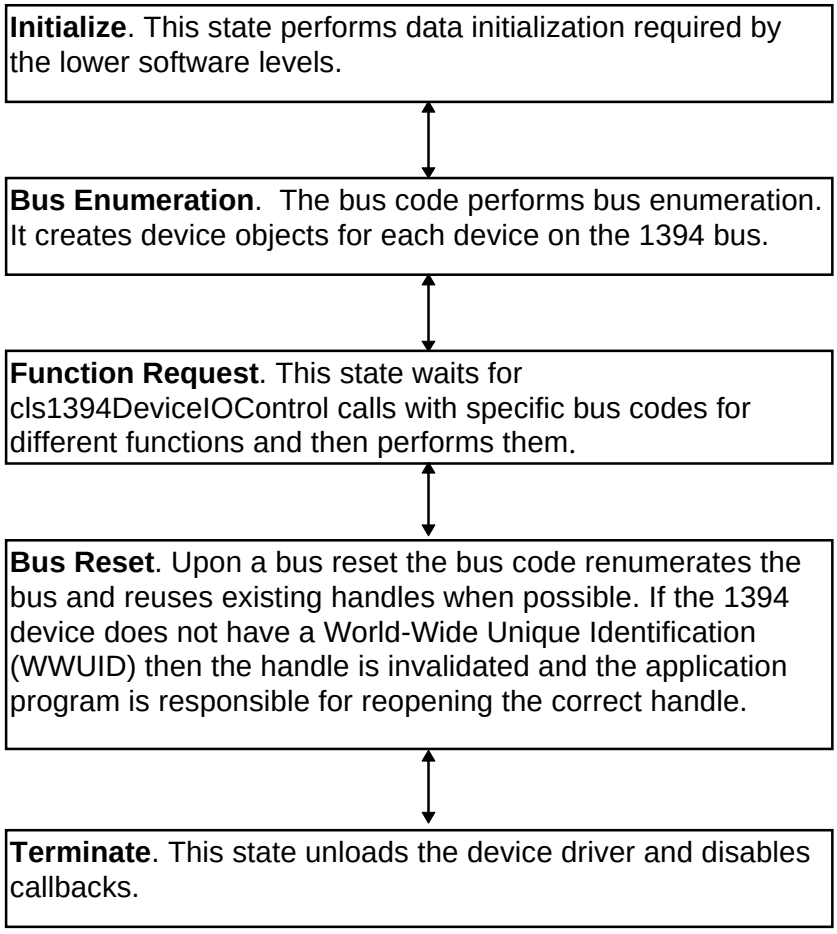
LynxSoft API Theory of Operation

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 (noncompliant 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 noncompliant 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.

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.
cls1394AllocateResources	*Must* perform this operation before the buffers are attached.
cls1394AttachBuffers	Attaches buffers to be transmitted to or from. This function *must* follow allocation of resources. The handle passed in *must* be valid.
cls1394Listen	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.
cls1394Talk	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. **Note:** In the case of the talk function, the user must detach and re-attach the buffers before issuing another talk command after a stop. This is a known bug that is planned to be corrected in the future.
cls1394Stop	Halts the transfer. This *must* be done before the buffers or resources are de-allocated.
cls1394DetachBuffers	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.
cls1394FreeResources	Frees the resources. This action *must* performed *after* the buffers have been detached.
cls1394FreeBandwith	Frees bandwidth
cls1394FreeChannel	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, cls1394Read	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 noncompliant (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.

Chapter 3

1394 Bus Driver API

The LynxSoft API is a set of services that enable 1394 actions. Each service consists of an input parameter to the cls 1394 Device IO Control function and a structure containing input parameters. Only cls1394DeviceIOControl is actually used as a function call.

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 BOOL cls1394Initialize();

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 noncompliant 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 GetAdapterAddress

Description Returns a pointer to the 1394 host adapter card

Action This function returns a pointer to the PCILYNX host-adapter card. The pointer can be cast using the structure defined in the file PCILYNX.H. The base structure is defined below showing substruct 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)

3.4 cls1394GetLastError

Description Obtains the error number upon a status failure

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_RESP_COMPLETE	operation completed
CLASS1394_RESP_1	reserved
CLASS1394_RESP_2	reserved
CLASS1394_RESP_3	reserved
CLASS1394_RESP_CONFLICT_ERROR	resource conflict, request can be retried
CLASS1394_RESP_DATA_ERROR	hardware error, data unavailable
CLASS1394_RESP_TYPE_ERROR	incorrect request packet
CLASS1394_RESP_ADDRESS_ERROR	incorrect address
CLASS1394_RESP_8_15	reserved
CLASS1394_GENERIC_FAILURE	generic fail code
CLASS1394_INVALID_REQUEST	DeviceIoControl invalid request
CLASS1394_WWUID_INVALID	CreateFile error
CLASS1394_WWUID_NOTFOUND	CreateFile error
CLASS1394_HANDLE_INVALID	CloseHandle error
CLASS1394_HANDLE_NOTOPEN	CloseHandle error
CLASS1394_SPEEDMAP_ERROR	Error accessing speed map
CLASS1394_NUM_DESTINATIONS_ERROR	Number of destinations for speed < 0
CLASS1394_RESPONSE_UNEXPECTED	Unexpected response packet
CLASS1394_RESPONSE_ZERO_DATA	Zero data in data payload packet
CLASS1394_RESPONSE_TIMEOUT	Time-out - no response in allotted time
CLASS1394_ISOALLOCRES_MEM_FAIL	Failed to allocated Internal Class Resource
CLASS1394_SPEED_NOT_AVAIL	Failed to acquire speed requested by the user.
CLASS1394_INVALID_ADDR_RNG	Failed to Allocate/Free 1394 Address Range

3.5 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.

3.6 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.

3.7 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 cls1394FileOpen.

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.

3.8 cls1394AllocateAddressRange

Description Allocates 1394 space for asynchronous requests

Action This function allocates a 1394 address range to be used in asynchronous requests to the local 1394 node. To use this function, the caller supplies a buffer. This buffer does not have to be locked down or fixed in memory. The device driver performs a scatter/lock on the buffer. The caller supplies the type of access a remote device has to this memory, as well as optional notification options so that the caller can be notified after 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 these newly mapped 1394 address(es) to the caller. The caller can then supply these address(es) to remote 1394 nodes, thus allowing the nodes to perform asynchronous requests to this memory region.

In most cases, the returned 1394 address(es) is an arbitrary one. However, callers of this API can elect to supply a specific 1394 address as pointed to by the *RequiredP1394Offset* parameter. This parameter is necessary to support devices that issue asynchronous requests to hardcoded 1394 addresses. When *RequiredP1394Offset* specifies a required address, then this 1394 address is always returned. An application cannot overlap 1394 addresses.

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.

An important point to consider when using this API is that *lp1394Offset* points to *an array of 1394 addresses* to be returned. The array of returned addresses must be big enough to hold the number of memory locations spanned by the specified buffer.

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 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 with an order to return whatever results are requested from this address range.

nLength –Specifies the length 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 to 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 read by the device.
- ❏ AccessTypeWrite - The memory region specified can be written to 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 device driver 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 device driver after carrying out an AsyncRead operation. This serves only as a notification to the device driver that their address space was accessed.
- ❏ NotifyAfterWrite – This option notifies the device driver after carrying out an AsyncWrite operation. This serves only as a notification to the device driver that their address space was written.
- ❏ NotifyAfterLock - This option notifies the device driver after carrying out an AsyncLock operation. This serves only as a notification to the device driver that their address was the target of an Atomic operation.

lpCallback – Points to the device driver 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 prenotification. Prenotification callbacks only occur when the *lpBuffer* parameter (above) is NULL, which indicates that the device driver 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 prenotification callbacks in all asynchronous cases, the device driver 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 device driver 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*. The device-driver callback function should also set the *lpEvent* parameter (*lpEvent) to point at an initialized event object, which is signaled when the buffer pointed at by (*lpData*)* has been sent, thus signifying that ownership of the buffer has been returned to the device driver.

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 - Describes the buffer that was originally submitted to cls1394AllocateAddress.

ulOffset - Specifies the byte offset within lpBuffer where the 1394 operation is pending.

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 prenotify AllocateAddressRange conditions (i.e. original lpBuffer == NULL).

lpLength - Is a pointer which in turn points to the length 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 device driver-supplied context data.

The device driver 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 device driver context data that is passed to the device driver callback routine when a notification event occurs.

Required1394Offset – If not equal to 0x000000000000, specifies the 1394 address to be returned in *lp1394Address.* This is not granted if previous callers have already allocated this 1394 address.

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. When equal to 0x000000000000, *lp1394Address* is an arbitrary 1394 address.

lpAddressesReturned – Points to a location where the number of 1394 addresses returned in *lp1394Offset* (below).

lp1394Address – If this call request completes successfully, points to the beginning of the 1394 addresses corresponding to the beginning of the application buffer. This address can be provided to remote 1394 nodes. The array pointed at by lp1394 address is an array of LARGE_INTEGER (64 bits); however, only the lower 48 bits of each LARGE_INTEGER entry is filled in with a 1394 offset. The extra 16 bits of each array element are unused, but are helpful for alignment purposes. The user should make no assumptions as to how the mapping between the 1394 address and the physical address is made.

Return Status

If successful, a STATUS_SUCCESS code is returned along with *lp1394Address* filled in. Device drivers can provide *lp1394Address* 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.

3.9 cls1394FreeAddressRange

Description Frees previously allocated address range

Action This function releases a 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 offsets are specified in *lp1394Offset.*

lp1394Address – Specifies a pointer to the 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) are returned in a prior successful call to cls1394AllocateAddress.

Return Status A STATUS_SUCCESS code is returned and the lp1394Address is now invalidated.

3.10 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 {
        P1394_ADDRESS    DestinationAddress;
        ULONG            nNumberOfBytesToRead;
        ULONG            nBlockSize;
        ULONG            fulFlags;
        PVOID            lpBuffer;
} clsAsyncRead;
```

Parameters

DestinationAddress – Specifies the P1394 48-bit destination address for this Asynchronous Read operation. This function is not valid for reading the user-operated node.

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

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

fulFlags - Is not used

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.

3.11 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 {
        P1394_ADDRESS   DestinationAddress;
        ULONG           nNumberOfBytesToWrite;
        ULONG           nBlockSize;
        ULONG           fulFlags;
        PVOID           lpBuffer;
} clsAsyncWrite;
```

Parameters *DestinationAddress* – Specifies the P1394 48-bit destination address for this Asynchronous Write operation. This function is not valid for reading the user-operated node.

nNumberOfBytesToWrite – Specifies the number of bytes to write to the 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 – Is not used

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.

3.12 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 {
        P1394_ADDRESS    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 – Is not used

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.

3.13 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.

3.14 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      lpChannelsAvailable;
} 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.

lpChannelsAvailable – 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.

3.15 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.

3.16 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 may 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.

3.17 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.

3.18 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;
} clsIsochFreeChannel;
```

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.

3.19 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.

3.20 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.

3.21 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.

3.22 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

3.23 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.

3.24 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.

3.25 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.

3.26 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.

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. There are a few items required before installation of the LynxSoft API code or the hardware. *If the EVM kit contents list includes a power cable, then the user should have a TSBKPCI card that requires external power. If the kit content list does not include a power cable then the TSBKPCI card is powered from the bus and external power is not required.* This involves capturing serial numbers and other items to be placed in the system.ini file. These items are explained below.

Determine from Table 4–1 which board has been ordered and record the serial number. This is used to create a WWUID for the device in the system.ini file. *The numbers in bold represent the VendorID_DeviceType as described in* cls1394CreateFile API *in Section 3.2, and should be the only thing needed when performing a device-open function from the test application.*

Table 4–1. WWUID Configuration

WWUID	TI Order Number/Name	TI Part Number	Serial Number (Obtained from Card)	Description
080028500000xxxx	TSBKPCITST	9806004-0001		PCILynx with ZV port, AUX port
080028510000xxxx	TSBKPCI	9806006-0001		PCILynx Lite

Installation Procedure

1. Bring up a DOS box on the screen display.
2. Insert the floppy disk into the floppy driver. These instructions assume that is the A: drive.
3. Type **CD A:** and change the default drive to the A:
4. Type **INSTALL** to execute the installation batch file. This file creates a LynxSoft directory and overwrites all duplicate files in that directory. The LynxSoft interface file is 1394api.h. The c:\lynxsoft\testutil directory has the test-utility source code and build files.

5. Open up the system.ini file in the Windows directory. Add the following line to the [386Enh] section to load the device driver:

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

6. Add the following section and lines to the system.ini file. The SerialNo field should be filled in with the recorded serial number portion of the WWUID in Table 4-1. For example, if the board serial number were 10 then input xxxx = 0010. This value completes the WWUID for the card. The BoardType entered should be TSBKPCITST or TSBKPCI. The BoardType is logically placed in the board configuration ROM space.

   [PCILYNX]
   DebugFlag = 1
   MaxNumberOfPages = 48
   SerialNo = xxxx
   BoardType = TSBKPCITST or TSBKPCI

7. Reboot the machine after modification of the system.ini file.

Chapter 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 noncompliant device and only allows it to be opened by using the noncompliant utilities of the 1394fileopen function. To ensure that both applications can "see" the other application as a compliant device, each application should be brought up and then 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 close the device handle of the first target and reopen the second.

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 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 a File, Misc, Async, Isoch, ISO Rx, ISO Tx, Camera, and Help pulldown menus. The following paragraphs gives a short explanation of the functionality of this test program.

5.1.1 File|Exit

The file menu item only contains an Exit selection that is valid.

5.1.2 Misc|Device|Open

This menu item brings up a dialog box that allows the user to open a 1394 device that is on the bus. The user is allowed to enter the IEEE Vendor ID/Device Type and the Device Entry. A valid device object must be opened before the test application can communicate with it. Only one device can be opened at a time. The default is targeted toward another PCILynx card. The API is defined in section 3.2.

5.1.3 Misc|Device|Close

This menu item closes the previously opened device object.

5.1.4 Misc|AddressRange|Allocate

This menu item brings up a dialog box 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. The API call used for this menu item is defined in section 3.8.

5.1.5 Misc|Address Range|Free

This menu item frees the address range that was previously allocated.

5.1.6 Misc|Query Cycle Number

This file menu returns the current 1394 cycle number.

5.1.7 Misc|Get 1394 Address

This file menu item returns the 1394 node number of the previously opened device object.

5.1.8 Async|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.

5.1.9 Async|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.

5.1.10 Async|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.

5.1.11 Isoch|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.

5.1.12 Isoch|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.

5.1.13 Isoch|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.

5.1.14 Isoch|Free|Bandwidth

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

5.1.15 Isoch|Free|Channel

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

5.1.16 Isoch|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.

5.1.17 Isoch|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.

5.1.18 Isoch|Buffers|Detach

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

5.1.19 Isoch|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.

5.1.20 Isoch|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.

5.1.21 Isoch|Stop

This menu item stops all isochronous transmission.

5.1.22 ISO Rx|Camera

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.

5.1.23 ISO Rx|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. The data is received, converted, and transmitted to the user screen.

5.1.24 ISO Rx|Stop|Camera

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

5.1.25 ISO Rx|Stop|Lynx->Lynx

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

5.1.26 ISO Tx|Lynx->Lynx

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.

5.1.27 ISO Tx|Stop

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

5.1.28 Camera|ON

This menu item communicates and turns on the Sony desktop camera and then commands it to begin transmitting isochronous data at the frame rate and data size expected from the ISO Rx|Camera menu item.

5.1.29 Help|About 1394test

This menu item displays the version of the test code.

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	90h	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

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

Table 7–1. Errata

Item	Description	Schedule for incorporation
Asynchronous Transmission while performing isochronous transmission	When a PCILynx node is transmitting isochronous data, the node cannot accept asynchronous data. Once isochronous transmission is halted, asynchronous traffic is again enabled. This is not true for the isochronous listener.	Dec. 1996
little_add Lock Function	The little_add Lock function does not perform according to specification. It performs a fetch_add function.	Dec. 1996