

# **VHPD1394**

## **Versatile High Performance IEEE 1394 Device Driver for Windows**

### **Application Design Guide**

**Version 2.21**

**April 07, 2010**

---

Thesycon® Systemsoftware & Consulting GmbH  
Werner-von-Siemens-Str. 2 · D-98693 Ilmenau · GERMANY

Tel: +49 3677 / 8462-0

Fax: +49 3677 / 8462-18

<http://www.thesycon.de>



Copyright (c) 1999-2010 by Thesycon Systemsoftware & Consulting GmbH  
All Rights Reserved

## **Disclaimer**

Information in this document is subject to change without notice. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use, without prior written permission from Thesycon Systemsoftware & Consulting GmbH. The software described in this document is furnished under the software license agreement distributed with the product. The software may be used or copied only in accordance with the terms of the license.

## **Trademarks**

The following trade names are referenced within this document:

Microsoft, Windows, Win32, Windows NT, Windows XP, Windows Vista and Visual Studio are either trademarks or registered trademarks of Microsoft Corporation.

Other brand and product names are trademarks or registered trademarks of their respective holders.



---

## Contents

|   |           |
|---|-----------|
| <b>Table of contents</b>  | <b>5</b>  |
| <b>1 VHPD1394 Object Model</b>  | <b>10</b> |
| <b>2 How to Enumerate and Open Devices</b>                                | <b>10</b> |
| <b>3 Device Plug&amp;Play Notifications</b>                               | <b>11</b> |
| <b>4 Virtual IEEE 1394 Devices</b>  | <b>12</b> |
| 4.1 Virtual Devices Overview . . . . .                                    | 12        |
| 4.2 Managing Virtual Devices . . . . .                                    | 12        |
| 4.3 VHPD1394 API Differences for Virtual Devices . . . . .                | 12        |
| <b>5 Asynchronous Transfer Mode</b>                                       | <b>14</b> |
| 5.1 Asynchronous Mode Overview . . . . .                                  | 14        |
| 5.2 Initiating Asynchronous Transactions . . . . .                        | 15        |
| 5.3 Asynchronous Transactions and Bus Reset . . . . .                     | 15        |
| 5.4 Receiving Asynchronous Requests . . . . .                             | 17        |
| 5.5 Address Range Buffer Store Mode . . . . .                             | 18        |
| 5.6 Address Range Buffer Queue Mode . . . . .                             | 18        |
| 5.7 Double-buffered Data Transfer . . . . .                               | 19        |
| <b>6 Isochronous Transfer Mode</b>  | <b>21</b> |
| 6.1 Isochronous Mode Overview . . . . .                                   | 21        |
| 6.2 Isochronous Streaming Concept . . . . .                               | 21        |
| 6.3 Buffer Queue Streaming Method . . . . .                               | 23        |
| 6.4 Operating an Isochronous Channel using Buffer Queue Method . . . . .  | 26        |
| 6.5 Shared Buffer Streaming Method . . . . .                              | 29        |
| 6.6 Operating an Isochronous Channel using Shared Buffer Method . . . . . | 32        |
| 6.7 Isochronous Packet Reception . . . . .                                | 34        |
| 6.8 Packet Formats in Listen Mode . . . . .                               | 36        |
| 6.9 Isochronous Packet Transmission . . . . .                             | 39        |
| 6.10 Packet Format in Talk Mode . . . . .                                 | 39        |



## References

- [1] Windows Platform SDK Documentation,  
<http://msdn2.microsoft.com/en-us/Library>
- [2] Anderson, Don, FireWire System Architecture, IEEE 1394a, MindShare, Inc., 2nd edition, 1998
- [3] IEEE Std 1394-1995, Standard for a High Performance Serial Bus
- [4] IEEE Std 1394a-2000, Standard for a High Performance Serial Bus – Amendment 1
- [5] IEEE Std 1394b-2002, Standard for a High Performance Serial Bus – Amendment 2



This document discusses general design issues to be considered when creating solutions based on the VHPD1394 device driver. It contains general descriptions of how to realize a data transfer from a device to the PC, or vice versa, with VHPD1394 when using either the asynchronous or the isochronous transfer method. The intent is to provide an overview and to help in making basic design decisions.

The document compares the two basic data transfer methods defined by the IEEE 1394 specification: asynchronous transfer mode and isochronous transfer mode. Note that the discussion of transfer modes is not complete. It provides an overview only and introduces important concepts required in order to understand the basic concepts of the VHPD1394 device driver. For more detailed information, refer either to the IEEE 1394 standard [3, 4, 5] or to MindShare's book on IEEE 1394 technology [2].

## 1 VHPD1394 Object Model

The VHPD1394 driver provides a communication model that is based on objects. The following object types exist:

- **Device Object**

A device object represents a physical device connected to the 1394 bus. The VHPD1394 driver automatically creates a device object for each device attached to the driver. If the corresponding device is disconnected from the system then the driver automatically destroys the device object and frees all associated resources.

- **Address Range Object**

An address range object represents a range of the local node's 1394 address space. Address range objects are created and destroyed by the driver on application request. An application uses such objects to receive data written to the local address space by other devices on the bus or to transmit data read from the local address space by other devices on the bus. An address range object is always associated with a device object.

- **Isochronous Channel Object**

An isochronous channel object represents a 1394 isochronous channel. The object is used by an application to transmit or receive isochronous data over this channel. Isochronous channel objects are created and destroyed by the driver on application request. An isochronous channel object is always associated with a device object.

Objects are created, destroyed and managed by the VHPD1394 driver. Object management is based on device handles. An application can open any number of handles for a particular device. A handle always refers to a device object.

When an application creates an address range object or an isochronous channel object then the object will be attached to the current device handle. The application has to use that handle to perform subsequent operations on the object. It is not possible to attach more than one of these objects to a device handle. The application has to create a separate handle for each object it intends to use. This concept is illustrated in figure 1.

## 2 How to Enumerate and Open Devices

The steps to be performed by an application to enumerate the devices currently connected to the computer are described below.

1. **Create a list of available devices.**

An application calls `CVhpd::CreateDeviceList` to enumerate the devices currently attached to the VHPD1394 driver. A unique software interface identifier (GUID) needs to be passed to this function. This allows to identify devices exported by the VHPD1394 driver unambiguously. A user should create a private GUID and configure the driver to use it.

The function `CVhpd::CreateDeviceList` returns a handle that represents the list of devices created as a result of the enumeration.

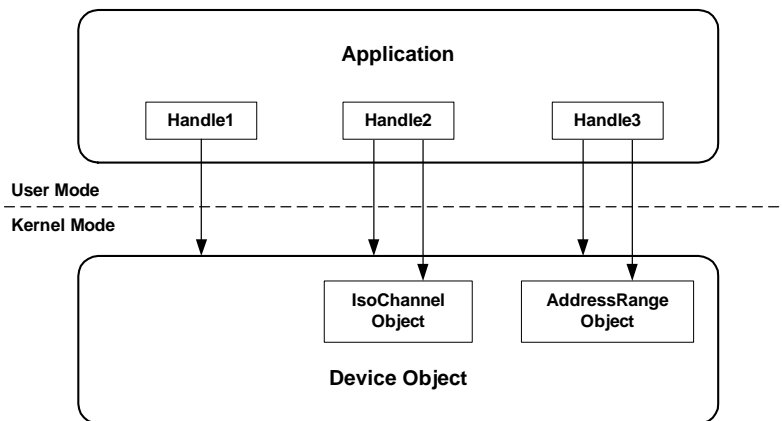


Figure 1: VHPD1394 objects and device handles

## 2. Scan the device list.

The application creates a loop that scans the device list returned by **CVhpd::CreateDeviceList**. Devices in the list are identified by an index. For each device the application calls **CVhpd::Open** and retrieves basic information on the device provided by the configuration ROM. This may include the device's vendor and model textual leaf, the unique ID (also called EUI-64 or GUID), etc. An application has to use those information to identify a device.

The index passed to **CVhpd::Open** is only valid until the device list is destroyed by a call to **CVhpd::DestroyDeviceList**. If a new device list is created by another call to **CVhpd::CreateDeviceList** then the order of devices may change.

## 3. Open a device.

After an application has selected the device it wants to use, it opens one or more handles for the device. Note that every **CVhpd** instance contains a handle and needs to be initialized with a call to **CVhpd::Open**.

Device enumeration is demonstrated by the source code sample scan.

## 3 Device Plug&Play Notifications

An application that uses the VHPD1394 device driver is able to receive notifications when a new device is connected to the computer, or when a device is disconnected. Notifications are issued by means of Window messages. A Windows application registers for device change notifications by means of the Win32 function `RegisterDeviceNotification`. See the Windows Platform SDK [1] for more information. For an implementation example, refer to the source code of the VHPD1394 demo application VHPDAPP.

A console mode Windows application cannot easily register for device change notifications because it does not contain a main window that will receive the notification messages. Such an application has to create a hidden window and to use a worker thread to process messages posted to this window. This is implemented by the class **CPnPNotificator** which is contained in the VHPDLIB class library. Usage of this class is demonstrated by the source code sample scan.

## 4 Virtual IEEE 1394 Devices

### 4.1 Virtual Devices Overview

Normally, to open a device handle it is necessary that a physical IEEE 1394 device is connected to the system. This device has to support the enumeration process of the Windows operating system, i.e. it has to expose a valid configuration ROM.

For Windows XP and later operating systems Microsoft provides a virtual device mechanism. This mechanism allows the user to create a device object without a dedicated hardware connected to the system and to create device handles using this device object. In other words, no dedicated physical device is required to access the IEEE 1394 bus. Thus, virtual devices provide a mean to establish a programming interface that provides a network level view of the IEEE 1394 bus. If the VHPD1394 device driver is installed on a virtual device then its API provides this level of access to the IEEE 1394 bus.

Using a virtual IEEE 1394 device may be useful in various scenarios. Some examples are given below.

- Using a virtual device it is possible to capture any isochronous stream that is present on the IEEE 1394 bus. It is not necessary to install the VHPD1394 driver for a particular physical device. All devices on the bus can be controlled by their default drivers.
- Using a virtual device established on a PC it is possible to transmit an isochronous stream on an arbitrary isochronous channel, independently of physical devices and other PCs.
- Using a virtual device it is possible to enumerate all nodes present on the bus, including nodes that are not recognized by Windows because there is no valid config ROM. How to do this is demonstrated by the source code sample vdevscan.
- Using a virtual device it is possible to send asynchronous requests to any node on the bus, including nodes that are not recognized by Windows. It is not required to install the VHPD1394 driver for a particular physical device.

### 4.2 Managing Virtual Devices

The VHPD1394 Development Kit provides a dynamic link library (DLL) that exports functions to create and delete virtual devices.

In addition, the development kit contains a Virtual IEEE 1394 Device Wizard (v1394DevWiz.exe) that supports interactive creation and removal of virtual devices.

Once a virtual device is created the VHPD1394 Installation Wizard can be used to install the VHPD1394 device driver on this virtual device.

### 4.3 VHPD1394 API Differences for Virtual Devices

Virtual devices have no corresponding hardware node and thus no corresponding identification information. As a result of this fact, there are some minor API differences depending on whether VHPD1394 is used for a physical or a virtual device.

The following I/O control requests are not supported on virtual devices:

- `IOCTL_VHPD_GET_CONFIG_INFO_PARAMS`
- `IOCTL_VHPD_GET_CONFIG_INFO_BLOCK`
- `IOCTL_VHPD_SET_DEVICE_XMIT_PROPERTIES`

The following I/O control requests exhibit different behavior depending on whether they are used on a physical or a virtual device:

- `IOCTL_VHPD_GET_ADDR_FROM_DEVICE_OBJECT`
- `IOCTL_VHPD_ASYNC_WRITE`
- `IOCTL_VHPD_ASYNC_READ`
- `IOCTL_VHPD_ASYNC_LOCK`
- `IOCTL_VHPD_ALLOC_ADDR_RANGE`

For more information on the differences and limitations, refer to the description of these requests.

I/O control requests that are not explicitly mentioned here behave in the same way regardless whether they are used on a physical or a virtual device.

## 5 Asynchronous Transfer Mode

### 5.1 Asynchronous Mode Overview

Asynchronous data transfers are either point-to-point or broadcast transfers. The data transfer is based on packets. An asynchronous packet is initiated by a particular node of the IEEE 1394 network and addressed to another node of the network. Broadcast packets are a special case. Such packets are issued by an individual node and received by all nodes on the bus. A two-stage handshake protocol ensures that the data packets are received without errors and successfully processed by the target node.

Transmission of asynchronous packets is best-effort. There is no bandwidth reserved in advance. The total bandwidth available for asynchronous transfers is shared among all nodes.

Every asynchronous packet contains a header with addressing information. This header contains the node ID of the sender (source node ID) and the node ID of the target node (target node ID). Furthermore, the header specifies a 48-bit target address offset within the target node's address space.

The specification defines two basic asynchronous packet types: request packets and response packets. To initiate an asynchronous transaction the source node issues a request packet. The target node receives and processes the request. Depending on the type of request the target node has to send a specific response packet back to the initiator.

Asynchronous transfer mode uses two levels of handshaking as described below.

- **Level 1: Acknowledges**

Level 1 of the handshaking protocol is implemented by a node's hardware, specifically the link layer controller. When the node receives an asynchronous packet then it verifies the checksum and depending on the result it sends an acknowledge back to the initiator of the packet. If there is a checksum error or any other problem with packet reception a negative acknowledge code is sent and the source node will repeat the transfer of the asynchronous packet. Because this handshaking mechanism is implemented completely in hardware level 1 error recovery is very efficient.

Note that both asynchronous request packets and asynchronous response packets need to be acknowledged by the target node.

- **Level 2: Response packets**

When a node successfully received an asynchronous request packet then it may indicate that a level 2 handshake will follow by sending a special acknowledge code of `ack_pending`. If the initiator receives this acknowledge code then it assumes that the target node has received the request undamaged. The software running on the target node has to process the request and send a asynchronous response packet back to the initiator. From the initiator's point of view this will complete the transaction. The header of the response packet contains a response code that indicates the completion status of the request.

So, level 2 handshaking is implemented in software. The target node's software determines the final result after it finishes processing of a request and generates an appropriate response packet. Note that level 2 handshaking (i.e. sending of a response packet) is mandatory for some request types, specifically read requests, and optional for other types of requests, specifically write requests.

On a Windows system, generation of response packets is completely handled by the 1394 driver stack. An application does not have to care about it.

## 5.2 Initiating Asynchronous Transactions

The VHPD1394 driver API offers functions that permit an application to issue asynchronous requests to the device, specifically read, write and lock requests. So the PC initiates an asynchronous transaction that needs to be handled by the device. In this scenario, the PC is also called the master and the device is called slave.

The driver API supports asynchronous write and read requests of any size. However, it is important to understand that an asynchronous request packet that is sent on the bus is limited in size. The maximum payload size of asynchronous request packets that is supported by a particular device is reported in the device's configuration ROM (see [2]). In addition, the payload size is limited by the transfer speed. For S400 speed the maximum payload size of asynchronous request packets is 2048 bytes.

If an application issues an asynchronous write or read request that exceeds the maximum payload size supported by the device then the driver will automatically split the request into a sequence of asynchronous packets.

Example: The maximum payload size supported by the device is 2048 bytes. An application issues a write request of 4200 bytes. The driver will send the following sequence of packets to the device: asynch write (2048 bytes), asynch write (2048 bytes), asynch write (104 bytes). In case of a read request of the same size the driver issues three read request packets and the device needs to reply with a response packet for each of those requests. There will be three response packets (2048 bytes, 2048 bytes, 104 bytes) containing the requested data bytes.

The driver request issued by the application will complete when all of the asynchronous request packets have been processed by the device. For the example described above this means that the driver API will complete the application's write request when all three write request are transferred (and optionally three write responses were received). The read request will be completed when all three response packets arrived at the PC.

Asynchronous requests are implemented very efficiently in the driver. The payload of asynchronous write request packets is read directly from the application's buffer by means of DMA. There is no intermediate data copying in the driver.

### Usage example

You can use asynchronous write requests to transfer large amounts of data from the application to the device. This is very efficient if large buffers are used and the driver will automatically split a buffer into a sequence of request packets.

Note that asynchronous read and write transactions are demonstrated by the source code samples read, write and txasy.

## 5.3 Asynchronous Transactions and Bus Reset

A bus reset event can occur at any time. Normally, a bus reset is generated when a node is added to the 1394 network or removed from the network. A bus reset event triggers a self ID phase. During this phase every node on the bus issues one or more self ID packets. The purpose of the

self ID mechanism is to assign a unique node ID to every node that is present on the bus. The node ID value that will be assigned to a particular node depends on the current bus topology, on timing behavior and on other factors. Generally, one has to assume that node IDs will change with every bus reset event.

Because asynchronous packets are addressed to a particular node this leads to a specific problem. If a bus reset occurs after an asynchronous request packet is issued by the application software, it could reach the wrong node because node IDs have changed after the packet was submitted but before it was transmitted on the bus. In order to avoid this addressing problem every node implements a bus reset generation counter. This is a counter, implemented in hardware, that is incremented by one every time a bus reset occurs. The current counter value can be read by software via a specific hardware register.

If application software issues an asynchronous request packet then it has to query the current bus reset generation counter value and to attach it to the packet when it is passed to hardware. This functionality is implemented in the VHPD1394 device driver. Normally, an application does not have to care about the bus reset generation counter. Note that this driver feature can be disabled, if required.

However, there are cases where a submitted packet can be rejected by the 1394 host controller hardware because the current generation count does not match the count specified in the packet. This can happen if a bus reset occurs in the time interval between submission of the packet and transmission on the bus. Because this time interval is small the failure situation is quite unlikely.

If an asynchronous request packet is rejected by the hardware then it is completed with an error status of `VHPD_STATUS_INVALID_GEN_COUNT`. An application should check for this status code and try to handle this specific failure situation. There are several approaches to handle failures caused by bus reset events. For example, the packet may be retransmitted, the packet may be discarded and the device may be reinitialized to restart data transfer, etc. Which strategy is best suited is defined by the particular application and depends on the design of the communication protocol. For this reason, a failure recovery mechanism is not implemented in the VHPD1394 driver nor in the `CVhpd` class.

The simplest recovery method is to retry a failed asynchronous request transmission. If an application detects the `VHPD_STATUS_INVALID_GEN_COUNT` error code it tries to issue the request packet again. This simple recovery strategy is demonstrated in the source code samples read and write. If an application needs to implement a more advanced recovery method it should register with the driver to receive bus reset notifications.

If a recovery method is implemented in the application there are various points to consider. If the asynchronous write or read request that was completed with `VHPD_STATUS_INVALID_GEN_COUNT` has a payload size that is larger than the maximum packet size supported by the device (e.g. 2048 bytes in case of S400 speed) then the request was splitted into a sequence of asynchronous packets. For a discussion of this concept, see also section 5.2. If the request fails, an application cannot determine which packet was rejected and completed with `VHPD_STATUS_INVALID_GEN_COUNT`. In other words, there is no way to find out how many of the packets are already transmitted to the device. So, if the application re-issues the entire write or read request then it possibly re-transmits payload bytes to the device that were already transmitted during the first attempt. Whether this is acceptable or not depends on the design of the communication protocol.

There is an additional issue in Windows XP SP2. In this system Microsoft has changed the behavior of the 1394 bus driver with respect to device enumeration. Device enumeration takes place every time after a bus reset. The bus driver updates its internal bus reset generation count variable not until it has completed enumeration of all nodes present on the bus. An enumeration requires to read the config ROM of every node on the bus. Depending on the response time of every device the enumeration process can take a long time, e.g. one second. Consequently, it is not possible to issue an asynchronous request in the time interval between a bus reset event and completion of device enumeration. If an application implements a retry strategy it should use the Win32 function Sleep to create a short delay between transmit attempts. Transmission should be retried periodically until the bus driver has finished its enumeration phase and updated its internal bus reset generation count value.

#### **5.4 Receiving Asynchronous Requests**

The PC is also able to be a target for asynchronous requests. This is also called slave mode. So the device can issue read, write or lock requests and the PC will generate the appropriate responses. Generation of response packets is completely handled by the driver. An application is not responsible for handling single asynchronous request packets.

By default, the PC does not expose any active address range to the bus, except for the configuration ROM. This means that write, read or lock requests sent by the device will fail. In order to enable reception of asynchronous request packets, an application has to establish an address range that will be the target for the requests issued by the device. The VHPD1394 driver API offers functions to create, to enable and to disable an address range. An application can create any number of address ranges for a given device. However, the address ranges cannot overlap.

An address range created by an application for an individual device is not visible to any other device on the bus. In other words, the driver manages a private address space for each device on the bus. Consequently, if an application wants to receive asynchronous requests from two devices simultaneously then it has to create an appropriate address range for each of them. The same base address can be used on each device. Because the address space is private for each device, no addressing conflicts can occur.

For each address range there is a buffer in main memory that provides backing store. See the following sections for a discussion of how backing store is managed. It is important to understand that every asynchronous request is targeted to a specific destination address as specified in the request packet header. The location at which the payload data of a request will be placed into the backing store buffer is determined by the destination address of the request. Destination addresses are absolute addresses within the address space of a bus node. When an application creates an address range then it has to specify a base address where the range starts. This base address is also an absolute address within the node's address space. Consequently, the offset at which the payload of an individual request will be placed into the buffer results from the difference between the request's destination address and the address range's base address. This concept implies that a bunch of data that is transferred via several asynchronous request packets does not necessarily need to occupy a contiguous region in the address range backing store. There may be gaps within the backing store buffer where no data bytes are present.

To support implementation of a data transfer based on an address range two different modes are provided to applications: buffer store mode and buffer queue mode. These modes are discussed in the next sections.

The basic functionality that is required to establish an address range is implemented by the class `CVhpdAsyncSlave` which is part of VHPDLIB.

### 5.5 Address Range Buffer Store Mode

In buffer store mode backing store of the address range is provided by the driver. The driver allocates a buffer from kernel memory that has the same size as the address range. Asynchronous requests received from the device are targeted to this buffer. If the device issues an asynchronous read request then the driver will generate a response packet that contains data read from the kernel-mode buffer. If the device issues an asynchronous write request then the driver will write the payload data to the buffer at the appropriate location. The location is determined by the destination address of the write request. See section 5.4 above for details.

Buffer store mode also supports access notifications which an application can optionally use to get informed when the device sent a request to the address range. Every time the address range receives a request the driver issues a notification packet to the application. In case of a write request the notification packet contains the complete payload data of the request.

#### Usage example

Buffer store mode could be used to implement a software register that is accessible by the device. The application initializes the register content before the address range is enabled. The device is always able to read from the register and to write new data to it. Optionally, the application can receive a notification when the device performed an access to the register.

The basic functionality that is required to establish an address range in buffer store mode and to prepare for receiving access notifications is implemented by the class `CVhpdNotifySlave` which is part of VHPDLIB. Usage of this class is demonstrated by the source code sample `pubmem`.

### 5.6 Address Range Buffer Queue Mode

In buffer queue mode backing store of the address range is provided by the application. The application allocates several buffers, each with the same size as the address range. The application submits these buffers to the driver. The driver manages a queue of buffers. The first buffer available in the queue will be used by the driver as backing store for the address range. Asynchronous requests received from the device are targeted to this buffer. If the device issues an asynchronous read request then the driver will generate a response packet that contains data read from the application's buffer. If the device issues an asynchronous write request then the driver will write the payload data to the buffer at the appropriate location. The location is determined by the destination address of the write request. See section 5.4 for details.

Within the address range there is a trigger address defined. When an asynchronous request hits the trigger address then the driver will complete the currently active buffer and switch to the next buffer from the buffer queue. The completed buffer will be returned to the application. Figure 2 below illustrates this approach.

Buffer queue mode is efficient if large amount of data are to be transferred. The transfer of packets from the bus to main memory (or vice versa) happens per DMA. However, because of the nature of asynchronous packet reception the driver needs to copy the payload data from the packet to

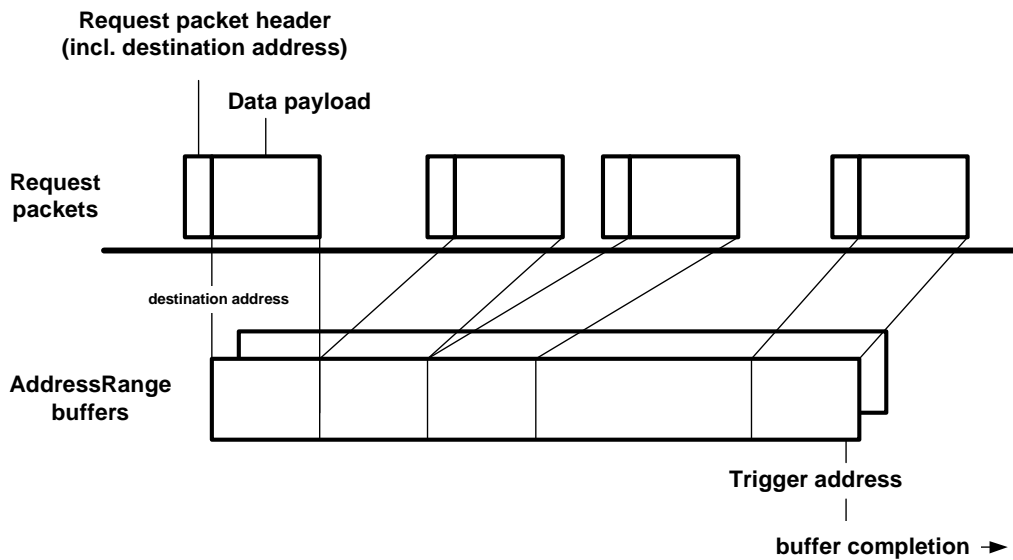


Figure 2: Address range buffer queue mode

the application's buffer (or vice versa). Besides, the driver has to handle an interrupt for each asynchronous request packet sent by the device. This may result in a high interrupt rate. So, depending on the data rate that is transferred buffer queue mode causes some CPU load.

#### Usage example

Use buffer queue mode if large amount of data are to be transferred, and delivery needs to be reliable. If there is a contiguous stream of data then buffer queue mode should be combined with a double buffer approach. This concept is discussed in the next section.

The basic functionality that is required to establish an address range in buffer queue mode and to manage the associated buffers is implemented by the class `CVhpdDataSlave` which is part of VHPDLIB. Usage of this class is demonstrated by the source code sample `rxasy`.

## 5.7 Double-buffered Data Transfer

In address range buffer queue mode, when the trigger address is hit the driver will not be able to react immediately. There will be a latency period that elapses before the driver's interrupt handler will run and switch to a new buffer. Furthermore, if the driver switches to another buffer then it has to temporarily disable the address range and to re-enable it with the new buffer. Consequently, from the device's point of view there is a small period in time where the address range is not accessible. This will prevent a continuous flow of data.

To overcome this, a special approach called double buffering should be used. An application needs to create two equally sized address ranges. Several buffers will be submitted to each of them. So, initially both address ranges are active and be able to accept requests sent by the device. The device starts sending request packets to address range 1 until it reaches the trigger address of this address range. When finished with address range 1 the device targets the requests to address range 2. This way, it is able to send a contiguous sequence of request packets.

When the trigger address of address range 1 is hit the driver's interrupt handler will run and will

switch to another buffer on this address range. While this happens the device will send request packets to address space 2. So the latency involved with the buffer switch is not an issue.

When the device is finished with address range 2 it switches back to address range 1. This will trigger a buffer switch for address range 2 in the driver. The device will continuously toggle between the two address ranges. Figure 3 below illustrates this approach.

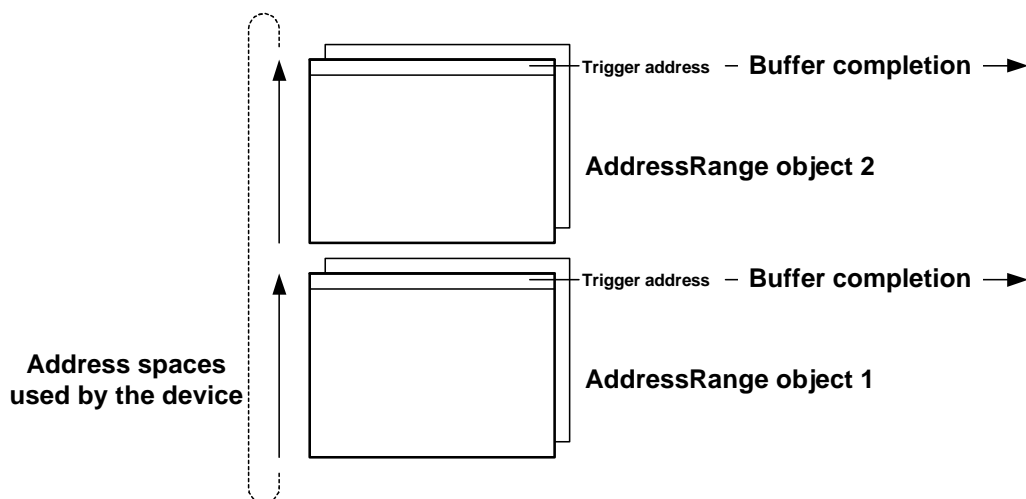


Figure 3: Asynchronous data transfer using the double buffer approach

Using the double buffer concept it is possible to transfer a contiguous stream of data from the device to the PC (or vice versa). The device sends a contiguous stream of asynchronous request packets. Every request packet will cause an interrupt and the driver has to handle the packet. The application has to handle buffer switch events for both address ranges. The buffer size should be chosen in such a way that handling of buffer switch events is not time-critical. It is recommended to choose a buffer size so that a buffer switch event occurs every 10..40 milliseconds. This can easily be handled by a Win32 user-mode thread.

#### Usage example

Use the double buffer concept in conjunction with buffer queue mode if large amount of data are to be transferred and there is a contiguous stream of data. For example, a camera device sends image data to the PC. An image is 400 kilobytes in size and the camera produces an image every 20 milliseconds. So, an application would create two address ranges, each 400 kilobytes in size. The application uses two buffers for each address range. Thus, total buffer space required is 1600 kilobytes. The camera device sends image data to the address range by means of asynchronous write requests. If an image is finished then the device issues a write request to the trigger address and switches to the other address range. The device can immediately continue and write the data of the next image to the second address range. The application detects a buffer completion event and processes the image data while the driver is receiving the next image. The application will receive and process a buffer completion event every 20 milliseconds.

The double buffer concept is implemented by the class `CVhpdBufSlave` which is part of the VHPDLIB class library. Usage of this class is demonstrated by the source code sample `rxasydb`.

## 6 Isochronous Transfer Mode

### 6.1 Isochronous Mode Overview

Isochronous data transfers are point-to-multipoint transfers initiated by a particular node of the IEEE 1394 network. This node is also called talker. Every node on the network is able to receive the isochronous data stream (broadcast transfer). A node that receives a particular isochronous data stream is called listener.

Isochronous data is splitted into packets and the sending node emits a contiguous stream of isochronous packets. There is no handshaking protocol and the transfer of an isochronous packet will never be retried. An isochronous packet contains a header that is one quadlet in size. See [3] and [2] for more information about the format of the isochronous packet header. Each packet includes a checksum for the header and a checksum for payload data. So, integrity of an isochronous packet can be checked by the receiver. However, the sender (talker) will not care about successful reception of the data stream. The checksum is verified by the 1394 host controller hardware. The host controller will only transfer correct packets to main memory. Defective packets will be discarded.

The bandwidth required to transmit an isochronous stream is reserved in advance. So delivery of isochronous data is guaranteed even if there are several talkers on a network. The isochronous packet header specifies the channel a packet belongs to. The standard supports up to 64 isochronous channels, which are numbered 0 to 63. In order to avoid conflicts with other streams, the channel number used by a talker needs to be allocated. Management of isochronous bandwidth and isochronous channels is implemented by the isochronous resource manager. Isochronous resource manager functionality is provided by exactly one node on a given network.

Transmission of isochronous packets is synchronized to the IEEE 1394 cycle clock. The standard defines a cycle clock of 8000 Hz. Every cycle begins with a cycle start packet (CSP) which is issued by a selected node. This node is called the isochronous cycle master. There is exactly one cycle master on a given network. In every cycle there is one packet transferred for each active isochronous channel (if no data is available then a talker may also omit a packet). Consequently, for an isochronous stream the packet rate is 8000 Hz. Because the size of an isochronous packet is variable the resulting bandwidth is variable.

### 6.2 Isochronous Streaming Concept

The VHPD1394 API provides functionality that enables an application to receive and to send an isochronous stream. So both isochronous listen and talk are supported. However, an application is not able to handle an isochronous stream packet by packet. This is because the packet rate is 8000 Hz. This means that an application would need to handle a packet every 125 microseconds. Because of the limited real-time capabilities of the Windows operating system this will typically not work.

In order to process an isochronous stream in software, several packets will be combined into a streaming buffer. An application will have to use several streaming buffers (at least two). The isochronous stream is processed buffer by buffer. The streaming buffer size determines the interval at which the application needs to process data. For example, if every streaming buffer contains 80 isochronous packets then the application has to process a buffer every 10 milliseconds, and it will have to process 80 packets at a time. This concept is illustrated in figure 4.

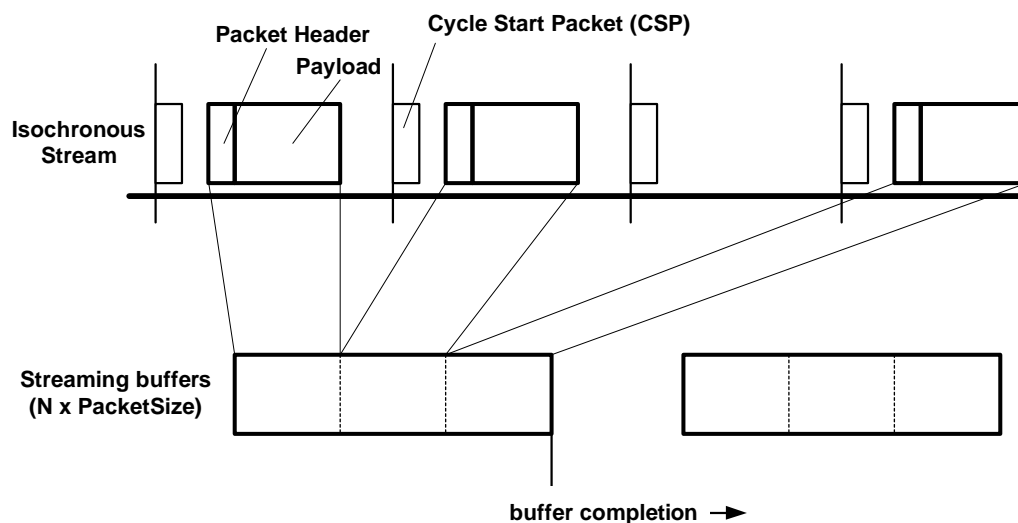


Figure 4: Mapping of packets to streaming buffers

An application needs to use at least two streaming buffers. One buffer is processed by the application while the other buffer is owned by the driver and will be used for the data transfer from/to the IEEE 1394 bus. An application may use more than two buffers, if desired.

It is very important to understand that the concept of streaming buffers is introduced only to overcome the limited real-time capabilities of a Windows PC. The buffers enable an application to process a contiguous stream of packets at certain intervals. But, an application should not make any assumptions on the alignment of packets to streaming buffers. The stream should still be interpreted as a contiguous sequence of packets. Otherwise, an application is not able to handle failure situations that lead to a loss of one or more isochronous packets. In listen mode, if an isochronous packet is lost then it will not be present in the streaming buffer. There will be no indication that a packet is missing. The buffer location of the missing packet is occupied by the subsequent packet. The application is responsible for detecting and handling this situation. In order to accomplish this, the data stream needs to contain additional information, usually presented in an additional packet header, that enable an application to verify consistency of the stream. Consequently, an application-specific streaming protocol needs to be defined. Below, an example scenario is described that illustrates this topic further.

### Example scenario

A camera device produces an isochronous data stream that carries image data. An image consists of 1024 lines. A line consists of 1024 pixels, each pixel is one byte in size. The line frequency is 8000 Hz and is synchronized to the 1394 cycle clock. So, every line is carried by an isochronous packet with 1024 bytes payload. A Windows application needs to receive the isochronous stream and to reconstruct the images. When an image is completely received it is passed to other software components for further processing.

### The wrong way

The application creates two streaming buffers. Each buffer holds 1024 packets. Thus, a streaming buffer holds exactly one image. The application initializes the stream in listen mode, submits both buffers to the driver and instructs the camera device to start the data transfer. When 1024 lines (a complete image) are received, the driver completes the first streaming buffer and immediately

continues to place packets into the second buffer. The buffer completion event notifies the application and a thread resumes execution and will process the buffer. Because the buffer should contain a complete image it can be passed on to other threads for further processing immediately. When done with the image the application submits the streaming buffer to the driver again.

This simple implementation will work as long as no packet is lost during transmission on the IEEE 1394 bus. Remember, in isochronous transfer mode there is no retransmission of packets. A packet that encounters a checksum error will simply be discarded by the receiver. Such a packet will not be present in the streaming buffer and the subsequent packet will be placed at its location. The streaming buffer will be completed and passed to the application when it has received 1024 packets. Consequently, if a packet loss occurs the application will receive a buffer that contains 1023 lines of the current image and the first line of the next image. There is no way for the application to detect this error situation and it will receive invalid image data. Furthermore, all subsequent images will also be defective even if no more packet losses occur. This is because there is a constant offset of one packet in every subsequent buffer.

### **The right way**

An application-specific 1-quadlet header is added to every isochronous packet. A packet will now carry 1028 bytes payload. The header contains a line number that identifies the position of a line within its image. The first line of an image has line number zero, the second line has number one and the last line of an image has the number 1023. The camera device has to create the isochronous packets according to this simple protocol definition.

The Windows application creates two streaming buffers. Each buffer holds 160 packets which corresponds to an interval of 20 milliseconds. The application initializes the stream in listen mode, submits both buffers to the driver and instructs the camera device to start the data transfer. When 160 lines are received the driver completes the first streaming buffer and immediately continues to place packets into the second buffer. The buffer completion event notifies the application, a thread resumes execution and will pass the buffer to a stream parser which has to process the data packet by packet. For every packet the parser checks the line number presented in the header. It will verify the contiguous sequence of line numbers and thus will be able to detect packet losses. If a packet loss is detected then the parser will discard the current image and resynchronize itself to the next image start which is indicated by a line number of zero. If the parser is done with a streaming buffer the buffer will be submitted to the driver again.

Because the streaming buffer contains image data which is interleaved with headers the stream parser needs to copy the image data to a separate image buffer which is used to assemble an image. The image buffer holds a complete image without the additional header information. When an image has been verified by the parser the image buffer can be passed to other threads for further processing.

This simple example scenario illustrates how a very robust solution can be created if a streaming protocol and an appropriate stream parser is used. An application will be able to detect any kind of packet loss or stream interruption and to resynchronize itself if streaming continues. So it will tolerate failure situations.

## **6.3 Buffer Queue Streaming Method**

The VHPD1394 driver supports two alternative methods for exchanging streaming data between application and driver: buffer queue method and shared buffer method. The former method is

described in this section, and the latter method is described in the next section.

If the buffer queue method is used for isochronous streaming then the application allocates a pool of streaming buffers and establishes a circulation of buffers. This is illustrated in figure 5.

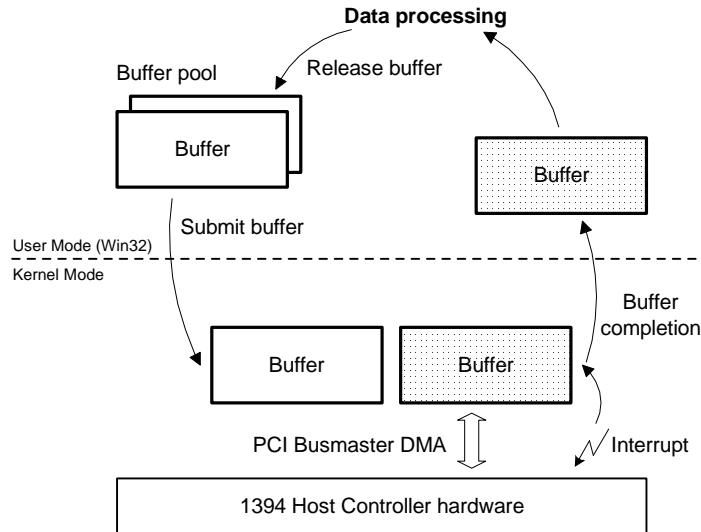


Figure 5: Buffer queue streaming method

Typically, a dedicated worker thread is used by the application to implement the buffer circulation. This worker thread handles buffer submission and buffer completion. Processing of streaming data is either be handled by this thread as well, or is implemented by other threads of the application.

Below, the exchange of streaming data is described in detail. This description assumes that an application is receiving an isochronous stream from the IEEE 1394 bus (listen mode). Note that talk mode (transmission of an isochronous stream) works in a very similar way.

1. The application creates a buffer pool and a worker thread. The pool has to contain at least two buffers.
2. Initially, the worker thread submits all available buffers from the buffer pool to the driver. The driver will attach the buffers to the 1394 host controller. If all available buffers are submitted then the worker thread instructs the driver to start the data transfer.
3. The worker thread waits for completion of the current buffer which is the buffer that was submitted first and will be completed next by the driver. The thread performs a wait operation on the Win32 event object that is attached to the buffer's OVERLAPPED structure. Basically, the VHPD1394 driver maintains the order of buffers, i.e. buffers will be completed in the order they were submitted.
4. The host controller writes isochronous packets to the current buffer as they are received from the bus. One packet will be written every 125 microseconds. The packet data is transferred to host memory by means of bus master DMA. Thus, data transfer proceeds with no intervention by the host CPU. When the current buffer is completely filled with packets the host controller issues an interrupt and switches to the next buffer.

5. The interrupt handler of the device driver completes the current buffer. From the applications point of view this results in completing the appropriate pending I/O operation. This will signal the Win32 event object that is attached to the buffer's OVERLAPPED structure and thus, awake the worker thread.
6. The worker thread has to check the completion status of the buffer. It may also query how many bytes were written to the buffer. If the buffer status is OK then the packets contained in the buffer will be processed. Typically, the buffer is passed to a stream parser that processes the data packet by packet. See also section 6.2 for a discussion of this topic.
7. If processing of a buffer is finished then the buffer is released. This returns the buffer to the pool. The worker thread immediately submits the buffer to the driver again and continues with step 3. This way, a loop is created that continuously transfers data.

In talk mode, the buffer circulation is implemented in the same way. Of course, data processing varies slightly. An application has to fill a buffer with isochronous packets before it is submitted to the driver. The host controller will read the packet data from the buffer by means of PCI busmaster DMA and issue an interrupt if all packets of a buffer are transmitted. On buffer completion, the worker thread passes the current buffer to the data processing unit that will fill the buffer again.

In order to guarantee a continuous flow of data at a reasonable CPU load, there are some points to consider when implementing the worker thread. These are discussed below.

At any moment in time, there has to be a buffer available at the 1394 host controller. If the host controller runs out of buffer space a stream interruption occurs. If two streaming buffers are used then the following sequence of actions needs to take place within the time interval that corresponds to a single buffer: the driver's interrupt handler is called, the event object is signalled, the worker thread resumes execution, the buffer is processed and is resubmitted. There is a latency associated with both the driver's interrupt handler and the worker thread. Particularly, the latency involved with worker thread resumption can be large. Consequently, an application has to choose an appropriate buffer size. A buffer size of 80...160 packets (10...20 milliseconds) should work reliable. An application should try to find the best compromise between stability and overall latency in stream processing.

An application should increase the scheduling priority of the worker thread that implements the buffer circulation (see the Win32 function `SetThreadPriority`). This will reduce the thread's latency significantly and increase stability.

### **Error handling**

When the driver handles an interrupt and is about to complete the current buffer it checks if there is still buffer space attached to the 1394 host controller. This will be the case if the application has handled completion of the previous buffer in time and submitted the buffer again (by a call to either `IOCTL_VHPD_ISOCH_SUBMIT_READ_BUFFER` or `IOCTL_VHPD_ISOCH_SUBMIT_WRITE_BUFFER`). If the application has not submitted the buffer in time (for example, because of excessive thread latencies) then most likely the host controller will run out of buffer space. This will cause a discontinuity in the stream of isochronous packets. If the driver detects this situation it completes the current buffer with a special status code of `VHPD_STATUS_DISCONTINUITY`. If the application receives a buffer with this status code it may assume that the isochronous packets contained in that buffer were still transferred correctly. But, most likely a stream discontinuity occurred immediately after this buffer.

Note that isochronous streaming using the buffer queue method is implemented by the classes

**CVhpdIsoListener** and **CVhpdIsoTalker** which are contained in the VHPDLIB class library. Usage of these classes is demonstrated by the source code samples rxiso and txiso respectively.

## 6.4 Operating an Isochronous Channel using Buffer Queue Method

### Initialization Steps

The steps to be performed by an application to set up an isochronous channel that uses the buffer queue streaming method are described below.

1. **Allocate isochronous bandwidth.**

This step is optional, but bandwidth must be reserved at the isochronous resource manager prior to starting the data transfer. This can be done by either the talker or the listener. Typically, bandwidth will be reserved by the talker.

Operations:

```
IOCTL_VHPD_ISOCH_ALLOC_BANDWIDTH  
CVhpd::IsoAllocBandwidth
```

2. **Allocate an isochronous channel.**

This step is optional, but the channel must be reserved at the isochronous resource manager prior to starting the data transfer. Typically, the channel will be reserved by the talker.

Operations:

```
IOCTL_VHPD_ISOCH_ALLOC_CHANNEL  
CVhpd::IsoAllocChannel
```

3. **Allocate resources.**

This step is required. It reserves kernel-mode resources needed to handle the data path. With the **IOCTL\_VHPD\_ISOCH\_ALLOC\_RESOURCES** call, the isochronous channel is initialized either in talk mode or listen mode.

Operations:

```
IOCTL_VHPD_ISOCH_ALLOC_RESOURCES  
CVhpd::IsoAllocResources
```

4. **Submit buffers.**

An application has to allocate streaming buffers and to submit those buffers to the driver. In order to guarantee contiguous streaming, buffers are to be submitted prior to starting the stream. Typically, the application creates a worker thread that handles the data transfer.

Operations in listen mode:

```
IOCTL_VHPD_ISOCH_SUBMIT_READ_BUFFER  
CVhpd::IsoSubmitReadBuffer
```

Operations in talk mode:

```
IOCTL_VHPD_ISOCH_SUBMIT_WRITE_BUFFER  
CVhpd::IsoSubmitWriteBuffer
```

5. **Start streaming.**

This step starts reception or transmission of the isochronous stream at the 1394 host controller.

Operations in listen mode:

```
IOCTL_VHPD_ISOCH_LISTEN
CVhpd::IsoStartListen
```

Operations in talk mode:

```
IOCTL_VHPD_ISOCH_TALK
CVhpd::IsoStartTalk
```

## Data Transfer

If streaming is in progress, an application periodically has to perform the steps described below. See also section 6.3 for a discussion of the buffer queue streaming method.

### 1. Wait for buffer completion.

The application waits until the current buffer is completed by the driver, i.e. a worker thread performs a wait operation on the event object attached to the OVERLAPPED structure of the buffer.

Operations:

```
WaitForSingleObject (Win32 function)
WaitForMultipleObjects (Win32 function)
CVhpd::WaitForCompletion
```

### 2. Check completion status.

The application has to check the completion status of the buffer. If the buffer is completed with a status of **VHPD\_STATUS\_SUCCESS** or **VHPD\_STATUS\_DISCONTINUITY**, the packets contained in the buffer were transferred successfully.

The status code **VHPD\_STATUS\_DISCONTINUITY** indicates that the stream was interrupted because the 1394 host controller has run out of buffer space.

Operations:

```
GetOverlappedResult (Win32 function)
CVhpd::WaitForCompletion
CVhpd::GetCompletionStatus
```

### 3. Process packets.

The application processes the packets contained in the buffer. In listen mode, the buffer should be passed to a stream parser. See section 6.2 for a discussion of this concept. In talk mode, the application has to fill the buffer with packets to be send.

### 4. Resubmit buffer.

When finished with the buffer the application has to submit the buffer to the driver again.

Operations in listen mode:

```
IOCTL_VHPD_ISOCH_SUBMIT_READ_BUFFER
CVhpd::IsoSubmitReadBuffer
```

Operations in talk mode:

```
IOCTL_VHPD_ISOCH_SUBMIT_WRITE_BUFFER
CVhpd::IsoSubmitWriteBuffer
```

## Cleanup Steps

The steps to be performed by an application to shut down and cleanup an isochronous channel that uses the buffer queue streaming method are described below.

**1. Stop streaming.**

This step stops reception or transmission of the isochronous stream at the 1394 host controller.

Operations:

```
IOCTL_VHPD_ISOCH_STOP  
CVhpd::IsoStop
```

**2. Abort pending buffers.**

This operation will cancel buffers that still may be pending at the driver. The buffers will be completed with an error status.

Operations:

```
IOCTL_VHPD_ABORT_IO_BUFFERS  
CVhpd::AbortIoBuffers
```

**3. Wait for completion of pending buffers.**

The application has to ensure that all buffers submitted to the driver are completed. Consequently, it has to check the completion status for each individual buffer and to perform a wait operation if the buffer is still pending. After that step, the application can free the streaming buffers.

Operations:

```
WaitForSingleObject (Win32 function)  
CVhpd::WaitForCompletion
```

**4. Terminate the worker thread.**

The application has to terminate its worker thread.

**5. Free resources.**

This step will free kernel-mode resources allocated for the isochronous channel.

Operations:

```
IOCTL_VHPD_ISOCH_FREE_RESOURCES  
CVhpd::IsoFreeResources
```

**6. Release isochronous channel.**

If an application has allocated an isochronous channel during setup it has to release it at the isochronous resource manager.

Operations:

```
IOCTL_VHPD_ISOCH_FREE_CHANNEL  
CVhpd::IsoFreeChannel
```

**7. Release isochronous bandwidth.**

If an application has allocated isochronous bandwidth during setup it has to release it at the isochronous resource manager.

Operations:

```
IOCTL_VHPD_ISOCH_FREE_BANDWIDTH  
CVhpd::IsoFreeBandwidth
```

## 6.5 Shared Buffer Streaming Method

If the shared buffer method is used for isochronous streaming then the application allocates one streaming buffer. This buffer is shared with the kernel-mode device driver and is used to exchange data with the driver. The structure of the shared buffer is shown in figure 6.

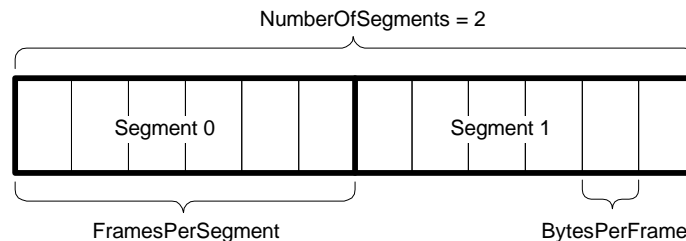


Figure 6: Structure of the shared streaming buffer

The buffer is described by three parameters which are discussed below.

- **BytesPerFrame**

This parameter specifies the number of bytes reserved in the buffer for every isochronous packet. In listen, mode this may have to include the isochronous header quadlet and an additional timestamp quadlet. For a detailed discussion of reception modes, refer to section 6.7.

The `BytesPerFrame` parameter has to be quadlet-aligned, i.e. has to be an integer multiple of 4.

- **FramesPerSegment**

The shared buffer is divided into several equal-sized segments (at least two). This parameter specifies the size, in terms of isochronous frames, of such a segment. If the driver is finished with a buffer segment it signals an event object to notify the application of available data (see also below).

- **NumberOfSegments**

This parameter specifies the number of segments contained in the shared buffer. The minimum number of segments is two.

The total size of the shared streaming buffer, in bytes, calculates as follows:

$$\text{TotalSize} = \text{NumberOfSegments} \cdot \text{FramesPerSegment} \cdot \text{BytesPerFrame}$$

In addition to the shared buffer the application has to provide a Win32 event object which is used by the driver to periodically notify the application to process stream data. Notification takes place every time the driver has finished data transfer from/to a buffer segment, i.e. every `FramesPerSegment` isochronous packets.

Typically, an application uses a dedicated worker thread to operate the shared buffer. Processing of streaming data can either be handled by this thread as well, or can be implemented by other threads of the application.

The shared buffer is operated like a circular buffer. The driver will process buffer segments sequentially, starting at segment zero. If the last segment is reached the driver will wrap around to

segment zero. For each buffer segment there is a state variable contained in a **VHPD\_ISOCH\_SHARED\_BUFFER\_STATUS** structure that is provided by the application and is shared with the driver. Thus, the state variables are accessible for the driver as well as the application. The circular buffer concept is illustrated in figure 7.

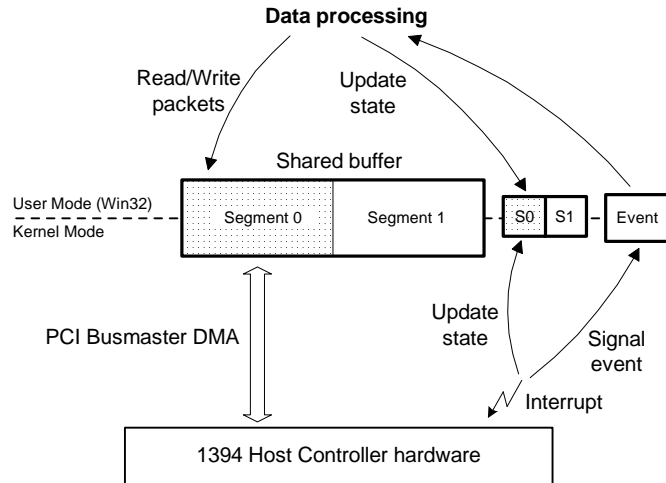


Figure 7: Shared buffer streaming method

At any moment in time a shared buffer segment is in one of two possible states:

**VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER** or **VHPD\_SHBUF\_SEG\_OWNED\_BY\_APP**.

The meaning of these states is described below.

- **VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER**

In this state the buffer segment is owned by the VHPD1394 driver. The 1394 host controller will transfer isochronous packets from or to this segment. The application should not access the segment.

- **VHPD\_SHBUF\_SEG\_OWNED\_BY\_APP**

In this state the buffer segment is owned by the application. The 1394 host controller or the VHPD1394 driver will not access the segment. An application can read or write isochronous packets from or to this segment while it is in this state.

Note that the state variable of a segment is updated by both the driver and the application. The initial state of a buffer segment is **VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER**. When the driver is finished with a segment it sets the state to **VHPD\_SHBUF\_SEG\_OWNED\_BY\_APP**. When the app is finished with a segment it resets its state to **VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER**.

Below, the exchange of streaming data is described in detail. This description assumes that an application is receiving an isochronous stream from the IEEE 1394 bus (listen mode). Note that talk mode (transmission of an isochronous stream) works in a very similar way.

1. The application allocates the shared buffer, allocates a **VHPD\_ISOCH\_SHARED\_BUFFER\_STATUS** structure and creates a Win32 event object (auto-reset type). It prepares the isochronous channel and attaches the shared buffer to it.

The application creates a worker thread and starts streaming on the channel. The worker thread maintains a variable that indicates the current buffer segment (segment index). The current segment index is reset to zero prior to starting the data transfer.

2. The worker thread performs a wait operation on the event object.
3. Starting at segment zero, the 1394 host controller writes isochronous packets to the shared buffer as they are received from the bus. One packet will be written every 125 microseconds. The packet data is transferred to host memory by means of bus master DMA. Thus, data transfer proceeds with no intervention by the host CPU. After `FramesPerSegment` packets were written (i.e. a segment is filled completely), the host controller issues an interrupt and continues writing to the shared buffer.
4. The interrupt handler of the device driver is invoked. This routine sets the state of the buffer segment filled by the host controller to `VHPD_SHBUF_SEG_OWNED_BY_APP`. Then it signals the Win32 event object. This will awake the worker thread of the application.
5. The worker thread determines the state of the current buffer segment. If the state is set to `VHPD_SHBUF_SEG_OWNED_BY_APP`, the thread processes the isochronous packets contained in that segment. After that, the worker thread resets the segment's state to `VHPD_SHBUF_SEG_OWNED_BY_DRIVER` and increments its current segment index variable to proceed with the next segment. If the last segment is reached, the index wraps around to zero. Processing stops either if the thread encounters a segment whose state is set to `VHPD_SHBUF_SEG_OWNED_BY_DRIVER` or if the total number of segments contained in the shared buffer is processed.

When stream data processing is finished and segment state variables are updated, the worker thread issues a `IOCTL_VHPD_ISOCH_SHARED_BUFFER_ACK` call to the driver. This will cause the driver to attach the buffer segment(s) processed by the application to the 1394 host controller again. The worker thread continues with step 2. This way, a loop is created that continuously transfers data.

### Example

An application needs to receive an isochronous stream that consists of variable-sized packets. The maximum packet payload size is 1024 bytes. The application uses packet-based mode and `QuadsToStrip=0`. See sections 6.7 and 6.8 for detailed information. In this mode, the packet header and the timestamp quadlet need to be considered. The resulting isochronous frame size is 1032 bytes and thus the application has to set the `BytesPerFrame` parameter to 1032.

The worker thread that handles the shared buffer should be signalled every 20 milliseconds and the shared buffer provides buffer space for two intervals, which is the minimum.

Thus, `FramesPerSegment` is set to 160 frames (corresponds to 20 milliseconds) and `NumberOfSegments` is set to 2. The resulting total size of the shared buffer is  $2 \cdot 160 \cdot 1032 = 330240$  bytes.

In this example scenario, the worker thread will be signalled by the driver periodically at an interval of 20 milliseconds. Every time the worker thread resumes execution there are 160 isochronous packets available for processing.

### Error handling

When the driver handles an interrupt and changes the state of a buffer segment it checks if there is still buffer space attached to the 1394 host controller. This will be the case if the application has

acknowledged the previous buffer event in time (by a call to `IOCTL_VHPD_ISOCH_SHARED_BUFFER_ACK`). If the application has not issued this call in time (for example, because of excessive thread latencies) then most likely the host controller will run out of buffer space. This will cause a discontinuity in the stream of isochronous packets. If the driver detects this situation it increments the `DiscontinuityCounter` member of the `VHPD_ISOCH_SHARED_BUFFER_STATUS` structure by one.

If the 1394 host controller reports an error status during stream processing then the driver stores the status code reported by the host controller driver in the `ErrorStatus` member if this member is not already set to an error status. After that, the driver increments the `ErrorCounter` member of the `VHPD_ISOCH_SHARED_BUFFER_STATUS` structure by one.

Note that isochronous streaming using the shared buffer method is implemented by the classes `CVhpdIsoShBuf` and `CVhpdIsoShBufWorker` which are contained in the VHPDLIB class library. Usage of these classes is demonstrated by the source code samples `rxisosb` and `txisosb` respectively.

## 6.6 Operating an Isochronous Channel using Shared Buffer Method

### Initialization Steps

The steps to be performed by an application to set up an isochronous channel that uses the shared buffer streaming method are described below.

1. **Allocate isochronous bandwidth.**

This step is optional, but bandwidth must be reserved at the isochronous resource manager prior to starting the data transfer. This can be done by either the talker or the listener. Typically, bandwidth will be reserved by the talker.

Operations:

```
IOCTL_VHPD_ISOCH_ALLOC_BANDWIDTH  
CVhpd::IsoAllocBandwidth
```

2. **Allocate an isochronous channel.**

This step is optional, but the channel must be reserved at the isochronous resource manager prior to starting the data transfer. Typically, the channel will be reserved by the talker.

Operations:

```
IOCTL_VHPD_ISOCH_ALLOC_CHANNEL  
CVhpd::IsoAllocChannel
```

3. **Attach the shared buffer.**

This step is required. An application has to allocate a streaming buffer and to pass it to the driver. The driver will map the buffer to kernel-mode address space and allocate all kernel-mode resources needed to handle the data path.

With the `IOCTL_VHPD_ISOCH_ATTACH_SHARED_BUFFER` call, the isochronous channel is initialized either in talk mode or listen mode.

Operations:

```
IOCTL_VHPD_ISOCH_ATTACH_SHARED_BUFFER  
CVhpd::IsoAttachSharedBuffer
```

**4. Initialize the streaming buffer.**

In talk mode, the application has to initialize the shared buffer so that it contains valid packets.

**5. Start streaming.**

This step starts reception or transmission of the isochronous stream at the 1394 host controller. Typically, the application creates a worker thread that handles the data transfer.

Operations in listen mode:

```
IOCTL_VHPD_ISOCH_LISTEN
CVhpd::IsoStartListen
```

Operations in talk mode:

```
IOCTL_VHPD_ISOCH_TALK
CVhpd::IsoStartTalk
```

**Data Transfer**

If streaming is in progress, an application periodically has to perform the steps described below. See also section 6.5 for a discussion of the shared buffer streaming method.

**1. Wait for shared buffer notification.**

The application performs a wait operation on the Win32 event object that is associated with the shared streaming buffer. The driver signals this event if packets are available for processing.

Operations:

```
WaitForSingleObject (Win32 function)
WaitForMultipleObjects (Win32 function)
```

**2. Process packets.**

The application checks the state of the current buffer segment. If the state is set to **VHPD\_SHBUF\_SEG\_OWNED\_BY\_APP** the application processes the packets of this segment and resets the state to **VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER**. The application proceeds with the next segment of the shared buffer. Processing stops if the application encounters a segment that is in state **VHPD\_SHBUF\_SEG\_OWNED\_BY\_DRIVER**. The application has to stop as well if it reaches the segment where it started processing.

In listen mode, the isochronous packets should be passed to a stream parser. See section 6.2 for a discussion of this concept. In talk mode, the application has to place packets into the buffer.

**3. Confirm buffer processing.**

The application issues a **IOCTL\_VHPD\_ISOCH\_SHARED\_BUFFER\_ACK** call to the driver in order to confirm processing of buffer segments.

Operations:

```
IOCTL_VHPD_ISOCH_SHARED_BUFFER_ACK
CVhpd::IsoAcknowledgeSharedBuffer
```

**Cleanup Steps**

The steps to be performed by an application to shut down and cleanup an isochronous channel that uses the shared buffer streaming method are described below.

**1. Stop streaming.**

This step stops reception or transmission of the isochronous stream at the 1394 host controller.

Operations:

```
IOCTL_VHPD_ISOCH_STOP  
CVhpd::IsoStop
```

**2. Terminate the worker thread.**

The application has to terminate its worker thread.

**3. Detach the shared buffer.**

This step will destroy sharing of the streaming buffer and will free kernel-mode resources allocated for the isochronous channel. After that step, the application can free the streaming buffer.

Operations:

```
IOCTL_VHPD_ISOCH_DETACH_SHARED_BUFFER  
CVhpd::IsoDetachSharedBuffer
```

**4. Release isochronous channel.**

If an application has allocated an isochronous channel during setup it has to release it at the isochronous resource manager.

Operations:

```
IOCTL_VHPD_ISOCH_FREE_CHANNEL  
CVhpd::IsoFreeChannel
```

**5. Release isochronous bandwidth.**

If an application has allocated isochronous bandwidth during setup it has to release it at the isochronous resource manager.

Operations:

```
IOCTL_VHPD_ISOCH_FREE_BANDWIDTH  
CVhpd::IsoFreeBandwidth
```

## 6.7 Isochronous Packet Reception

In listen mode, the VHPD1394 driver will place isochronous packets received from the bus into streaming buffers provided by an application. There are two basic strategies for isochronous packet reception: stream-based reception mode and packet-based reception mode. Note that those reception modes apply to both data exchange methods: buffer queue method (section 6.3) and shared buffer method (section 6.5). The reception mode to be used for an isochronous channel is selected by the `VHPD_FLAG_ISOCH_PACK_BASED` flag in the `VHPD_ISOCH_ALLOC_RES` or `VHPD_ISOCH_ATTACH_SHARED_BUFFER` structure.

- **Stream-based mode**

In this mode the driver ignores packet boundaries while it fills the stream buffer. There will be no gaps in the data. The stream buffer contains a contiguous sequence of isochronous packets.

- **Packet-based mode**

In this mode the driver places each packet at a distance from the beginning of the stream buffer that is an integral multiple of the maximum packet size. If a particular packet is smaller than the maximum, the data located between the end of that packet and the start of the next packet is undefined. So there may be gaps in the data.

Both modes, stream-based and packet-based, have very similar characteristics when all packets are of the same size. But when the packet size varies, the modes have very different characteristics and the resulting buffer layout is different. This is illustrated in figure 8.

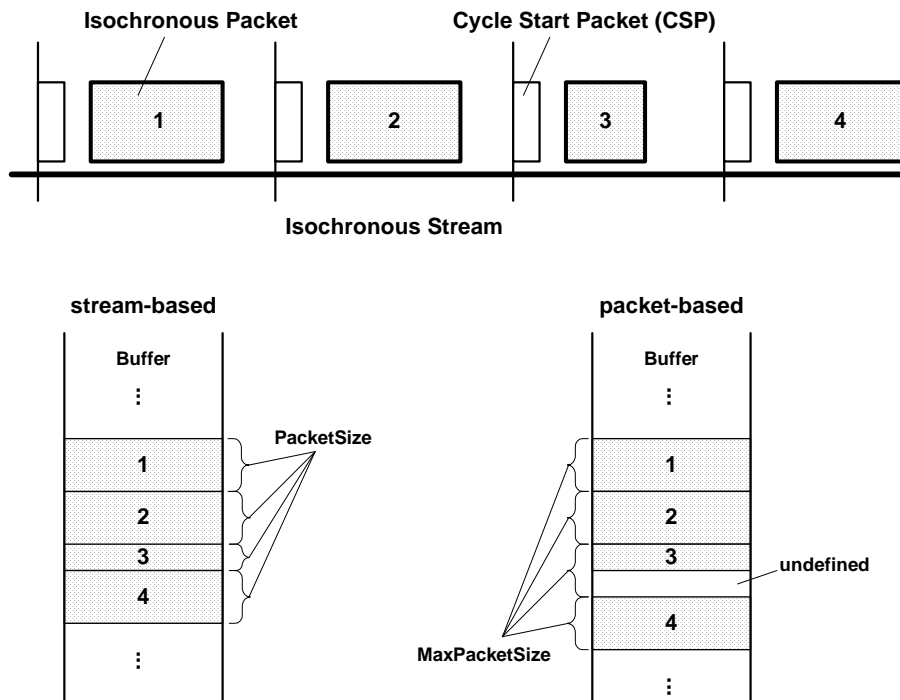


Figure 8: Stream-based versus packet-based reception

Using stream-based mode with variable-sized packets can cause problems because packets will straddle buffer boundaries. This is not correctly supported by the 1394 OHCI device driver included with Windows. Consequently, stream-based mode should not be used if the packet size varies. Thesycon recommends to always use packet-based reception mode. Typically, a stream parser is required in order to process the incoming stream packet by packet. (see also section 6.2). Thus, it is not a problem to skip gaps in the buffer or to discard additional quadlets.

The size of the stream buffers used for stream reception should be an integral multiple of the packet size. In packet-based mode this is the maximum packet size that occurs in the stream. An overview of how the stream buffer size is calculated is given in table 1.

|                               | <b>stream-based</b>                            | <b>packet-based</b>                               |
|-------------------------------|--|---|
| <b>fixed-sized packets</b>    | StreamBufferSize = $N \cdot \text{PacketSize}$ | StreamBufferSize = $N \cdot \text{PacketSize}$    |
| <b>variable-sized packets</b> | should not be used                             | StreamBufferSize = $N \cdot \text{MaxPacketSize}$ |

Table 1: Isochronous packet reception overview ( $N$  is an integer number,  $N \geq 1$ )

## 6.8 Packet Formats in Listen Mode

In listen mode, the format of the isochronous packets placed into the streaming buffer depends on the selected reception mode, either stream-based or packet-based, and additionally on the `QuadsToStrip` parameter that is specified when the isochronous channel is prepared for data reception.

In both reception modes, if `QuadsToStrip` is set to zero then every packet includes a quadlet that contains the isochronous packet header. The layout of the isochronous packet header conforms to the IEEE 1394 standard [3]. You can also refer to MindShares's book on IEEE 1394 technology [2] for more information.

Furthermore, if `QuadsToStrip` is set to zero then an additional quadlet will be placed into the buffer for every packet. This quadlet contains a timestamp that is produced by the 1394 host controller. However, the layout of the timestamp field is not documented. So an application should not use this field and it should simply be ignored. But, it is very important to consider the additional timestamp quadlet when the required streaming buffer size is calculated. The total size of an isochronous packet includes both the header quadlet and the timestamp quadlet.

The `QuadsToStrip` parameter allows to strip off the header and timestamp information from the packet when it is placed into the streaming buffer. If `QuadsToStrip` is set to 1 then the buffer will contain payload data only.

**Packet format in stream-based reception mode**

If `QuadsToStrip` is set to zero then the packet starts with the isochronous packet header. The timestamp quadlet follows the packet's payload field immediately. This packet format is shown in figure 9.

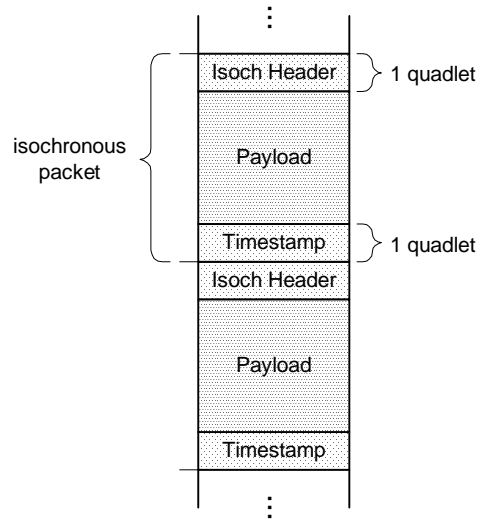


Figure 9: Packet format in stream-based reception mode (`QuadsToStrip=0`)

If `QuadsToStrip` is set to 1 then the buffer will contain payload data only. This is illustrated in figure 10.

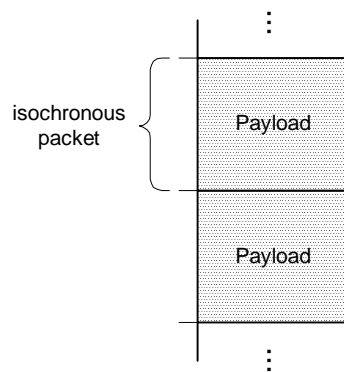


Figure 10: Packet format in stream-based reception mode (`QuadsToStrip=1`)

**Packet format in packet-based reception mode**

If `QuadsToStrip` is set to zero then the packet starts with the timestamp quadlet. The next quadlet contains the isochronous packet header and the payload data follows. This packet format is shown in figure 11.

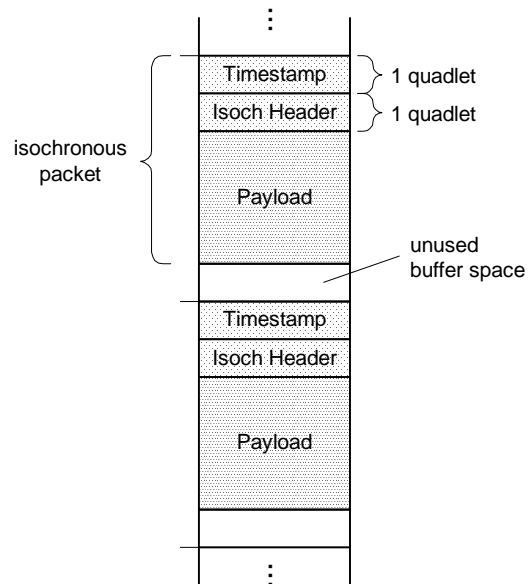


Figure 11: Packet format in packet-based reception mode (`QuadsToStrip=0`)

If `QuadsToStrip` is set to 1 then the timestamp and header quadlet is omitted and the buffer will contain payload data only. This is illustrated in figure 12.

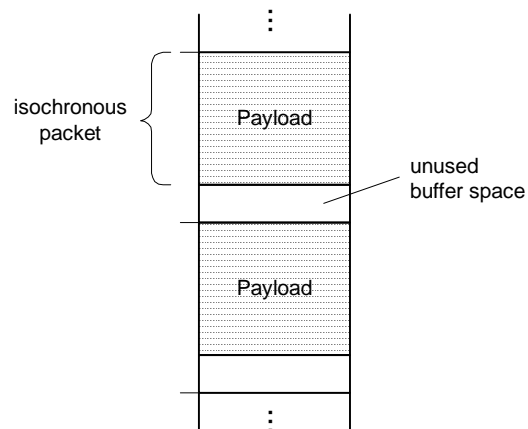


Figure 12: Packet format in packet-based reception mode (`QuadsToStrip=1`)

## 6.9 Isochronous Packet Transmission

In talk mode, the VHPD1394 driver will transmit isochronous packets provided by an application over an isochronous channel. The required packet format is described in the next section.

Note that an application can specify a `BytesPerFrame` parameter per streaming buffer that is submitted to the driver. This implies that all packets contained in a buffer are of the same size. Thus, an application will not be able to send an isochronous stream that consists of variable-sized packets. This limitation applies to both data exchange methods: buffer queue method (section 6.3) and shared buffer method (section 6.5).

If a variable-sized packet payload is needed it is recommended to add an additional header to every packet that contains information on the number of valid data bytes contained in that packet. The rest of the packet is filled with stuffing bytes. This way, a stream of equal-sized packets is created.

## 6.10 Packet Format in Talk Mode

In talk mode, there is only one packet format which is illustrated in figure 13.

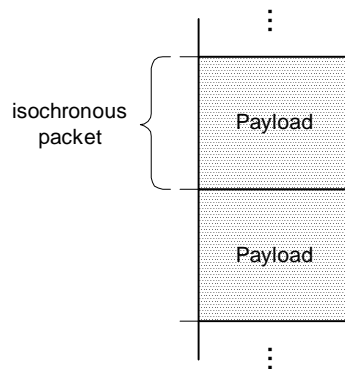


Figure 13: Packet format in talk mode

The streaming buffer contains the payload data of the packets to be transmitted. The isochronous packet header will automatically be added by the driver. The application is able to define the values of the `SY` and `TAG` fields presented in the header.

