



Unibrain FireAPI™ Documentation

1394 Class Driver User Mode Interface

Release 5.60 (Dec 2008)

*A detailed list of features added to each version of FireAPI
can be found at the end of this document.*

Part I - Technical Overview	1
Introduction	2
References from the IEEE-1394 Standard.....	2
Architecture.....	3
Conventions.....	3
Before You Start	4
Initializing with FireAPI.....	5
Initializing and Terminating 1394 Support.....	5
Multiple Adapter Support	6
Enumerating the installed adapters.....	6
Opening a handle to an adapter.....	7
Opening an adapter by GUID.....	8
Receiving notifications about the dynamic addition of 1394 adapters.....	9
Asynchronous Operations	10
Initiating Transaction Requests	10
Maximum Transmission Speed Per Destination NodeID	10
Maximum Asynchronous Packet Size.....	11
Bandwidth Consumption	13
Broadcast Requests.....	13
Software Loopback	13
Transaction Management	14
Transaction Label Management for Outgoing Transaction Requests	14
Transmit Order.....	15
NodeID & Device Functions	16
Bus Resets & Asynchronous Transactions	18
Bus Reset Exceptions (obsolete)	19
Reading the GUID of bus nodes	20
Reading the GUID of bus nodes – Transaction Failures.....	21
Reading the GUID of bus nodes – Transaction Timeouts.....	22
Reading the GUID of bus nodes – Various Notes.....	23
Determining the nodes connected to the 1394 Bus	24
Enumerating the Devices on the 1394 Bus.....	26
Non-blocking Calls	28
Write Transaction Requests.....	30
Lock Transaction Requests.....	31
Handle-Based Functions.....	33
Using Device Handles.....	34
Retrying transactions with <i>C1394Retry</i> functions.....	37
Performing Asynchronous Streaming transactions	38
Accepting Transactions from Remote Nodes	41
The 1394 Address Space	41
Address Ranges in the 1394 Address Space.....	42
Allocating and Freeing an Address Range.....	43
Incoming Transaction Request Processing.....	45
Application Control Flow	47
Simple CSR Server Sample.....	47
Mapping an Address Range to more than one adapters.....	49

Performance Optimization for Incoming Requests.....	49
Requests Spanning Address Ranges	51
Receiving Asynchronous Streaming transactions.....	51
Advantages of FireAPI Incoming Transaction Request Processing	51
Summary of Class Driver Transaction Processing functions.....	51
Common Errors in Transaction Processing.....	52
Event Notifications.....	53
Registering a Bus Reset Notification.....	54
Using a separate thread for events	56
Notes on Bus Reset Processing	58
Isochronous Operations.....	59
Adapter Channels & DMA Channels	59
DMA Multiplexing	61
DMA Multiplexing Modes	62
Opening an Adapter Channel.....	64
Enabling stream channel numbers for an Adapter Channel.....	65
Adapter Channel Operating Models	68
Queued-Completion Model	70
Instant Completion Notification	71
Isochronous Request Types	72
Why use 'Packet' or 'Fixed' operations?	74
Design Examples	75
Processing Isochronous Requests	77
Outline of Isochronous Processing Loop	77
Queueing Isochronous Requests.....	78
Retrieving Complete Isochronous Requests.....	79
How many requests to queue?	80
Queueing 'Small' Requests.....	81
Isochronous Options	82
Common Mistakes in Isochronous Processing	83
Isochronous Operation Limits	87
Isochronous Resource Allocation	88
Isochronous Timing.....	88
The Protocol.....	88
Bus Reset & Isochronous Resources	90
Identifying the IRM.....	90
Allocating a channel number using compare swap.....	90
Allocating bandwidth using compare swap	92
Freeing Isochronous Resources	92
VersaPHY Operations	93
VersaPHY Basics.....	93
VersaPHY Functions & Profiles	93
VersaPHY Transactions.....	94
VersaPHY API Overview	94
VersaPHY PhyID functions	94
VersaPHY Label functions	94
VersaPHY Packet Structures.....	95
VersaPHY Packet Initialization/Handling	96

VersaPHY Transaction Serialization	96
VersaPHY Transaction Timeout.....	97
Miscellaneous Topics	98
Endianness Considerations	98
Endianness Swapping	98
Utility String Functions	99
64-bit Integer Arithmetic.....	100
Path Speed Information	101
Bus Topology Information	103
C1394_NODE_INFO	103
SelfID Analysis Error Codes.....	106
Topology Analysis Error Codes.....	108
Manipulating CYCLE_TIME timestamps.....	109
Application Reaction Time	112
Accessing the Link Layer Registers	115
Changing the FIFO settings	116
Part II FireAPI Function Reference	120
Initialization Functions	121
C1394Initialize.....	122
C1394Terminate.....	123
C1394GetAdapters.....	124
C1394OpenAdapter	125
C1394CloseAdapter	126
Outgoing Asynchronous Transactions.....	127
C1394ReadNode.....	128
C1394WriteNode.....	131
C1394LockNode.....	134
C1394CompareSwapNode.....	139
C1394TransmitRaw.....	141
C1394ReadNodeAsynch	143
C1394WriteNodeAsynch	146
C1394LockNodeAsynch.....	149
C1394PingNode	153
C1394ReadPHYRegister	154
C1394QueryPhyBaseRegs	156
C1394QueryPhyPagedRegs	158
C1394TransmitPackets	160
FIREAPI_TRANSACTION	164
C1394AcknowledgeBusReset.....	166
C1394CompleteAsynch.....	167
Incoming Asynchronous Transactions.....	168
C1394MapAddressRange	169
C1394UnmapAddressRange.....	173
C1394GetNextRequest	174
C1394SendErrorResponse	176
C1394SendResponse	177

C1394ServiceTransactionRequest.....	178
C1394CompletePacket	179
C1394CompletePackets.....	179
Device Handle Functions	180
C1394OpenDevice	181
C1394CloseDevice.....	182
C1394ReadDevice	183
C1394WriteDevice	186
C1394GetDeviceNodeID	189
Retry Functions.....	190
C1394MayRetryTransaction.....	191
C1394RetryReadNodeInQuads.....	192
C1394RetryReadNodeExInQuads	193
C1394RetryReadDeviceInQuads	194
C1394RetryWriteDeviceInQuads	195
Isochronous Processing	196
C1394OpenAdapterChannel	197
C1394CloseAdapterChannel.....	200
C1394IsochQueue	201
C1394GetNextCompleteRequest.....	203
C1394IsochCancel	204
VersaPHY Functions.....	205
C1394VPReadNode.....	206
C1394VPWriteNode	209
C1394VPSendPacket.....	212
C1394VPChannelOpen	213
C1394VPChannelClose	216
C1394VPChannelGetNextPacket.....	217
C1394VPChannelRead	218
C1394VPChannelWrite	220
Control & Information Functions.....	221
C1394BusReset	222
C1394IsBusResetInProgress	223
C1394GetBusResetCount	224
C1394QueryInformation	225
C1394QueryBooleanInformation	238
C1394QueryULONGInformation	239
C1394SetInformation	240
C1394GetAdapterMaxRec	244
C1394GetAdapterGUID	245
C1394GetAdapterNodeID	246
C1394GetAdapterSpeed	247
C1394GetCycleTime	248
C1394GetIRMNodeID	249
C1394GetRootNodeID	250
C1394GetMaxPayloadForMaxRec.....	251

C1394GetMaxPayloadForSpeed.....	252
C1394GetMaxSpeedBetweenNodes.....	253
C1394GetMaxSpeedToNode	254
C1394GetNodeSpeed	255
C1394GetPhyPacketType	256
C1394GetTransactionType	257
C1394GetExpectedResponseCode	258
C1394IsResponseCodeLegal.....	259
C1394IsTransactionCodeLegal	260
Event Notification Functions	261
C1394RegisterNotification.....	262
C1394UnregisterNotification.....	266
C1394GetAsynchEvent	267
C1394GetAsynchEventHandle.....	268
C1394GetAddAdapterEventHandle.....	269
Miscellaneous Functions	270
C1394AddBigEndian32	271
C1394AddBigEndian64	271
C1394CalculateCRC16	272
C1394CalculateCRC8	273
C1394CalculateLinkCRC	274
C1394DebugPrint.....	275
Part III FireAPI Structures & Macros Reference	276
C1394_PACKET_HEADER.....	277
Incoming Packets & C1394_PACKET_HEADER.....	277
Outgoing Packets & C1394_PACKET_HEADER.....	279
FIREAPI_ISOCH_REQUEST	282
Request Index.....	285
Request Timeouts.....	285
Bus Reset Handling	287
Other Options.....	288
Isochronous Completion Status	289
Isochronous Operation Parameters.....	291
C1394_ISOCH_RCV_FIXED_PKTS.....	291
C1394_ISOCH_RCV_FIXED_DATA	293
C1394_ISOCH_RCV_FIXED_DATA_NH	295
C1394_ISOCH_XMIT_PKTS	296
C1394_ISOCH_XMIT_FIXED_PKTS.....	298
C1394_ISOCH_XMIT_DATA.....	300
C1394_ISOCH_IDLE_CYCLES.....	301
Isochronous Packet Header Structures & Macros.....	302
C1394_STREAM_PACKET_HEADER	302
MAKE_ISOCH_HEADER	302
PHY Packet Structures & Macros.....	303
C1394_PHY_PACKET_GENERIC	303
C1394_PHY_PACKET_SELF_ID_0	303
C1394_PHY_PACKET_SELF_ID_N.....	304
C1394_PHY_PACKET_SELF_ID_1	305

C1394_PHY_PACKET_SELF_ID_2	306
C1394_PHY_PACKET_SELF_ID_3	307
C1394_PHY_PACKET_LINK_ON	307
C1394_PHY_PACKET_CONFIGURATION.....	308
C1394_PHY_PACKET_EXTENDED	308
C1394_PHY_PACKET_PING	309
C1394_PHY_PACKET_REMOTE_ACCESS.....	309
C1394_PHY_PACKET_REMOTE_REPLY.....	310
C1394_PHY_PACKET_REMOTE_COMMAND.....	311
C1394_PHY_PACKET_REMOTE_CONFIRMATION.....	312
C1394_PHY_PACKET_RESUME	313
C1394_PHY_PACKET.....	313
Status Codes Reference (<i>alphabetical listing</i>).....	315
Change History (<i>reverse chronological order</i>).....	318

Part I - Technical Overview

Introduction

This document contains a description of the 1394 features provided by FireAPI. It is not intended as a general purpose 1394 tutorial. Please refer to the IEEE 1394 standard or to other provided documentation for a general description of the operations of the 1394 protocol.

This documentation assumes that you are familiar with the following core 1394 concepts:

- 1394 Address Space & Topology
- Bus & Physical Identifiers
- 1394 Bus Reset procedures
- 1394 Transmission Rates
- Control & Status Registers (CSRs)
- Transaction Requests & Responses
- Split & Unified Transactions
- Transaction Labels
- Packet Acknowledges
- Asynchronous & Isochronous transmission mode
- 1394 Globally Unique Identifiers (GUIDs)

You may get information about the above concepts by referring to the following paragraphs of the IEEE 1394-1995 and IEEE 1394a-2000 standards:

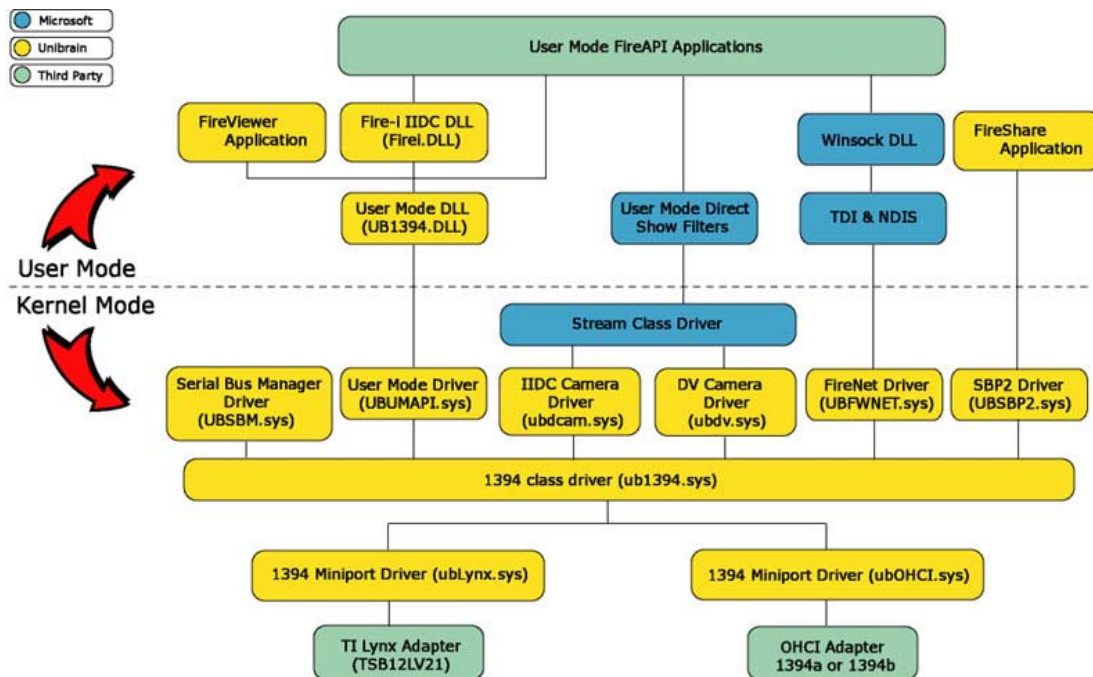
References from the IEEE-1394 Standard			
Ch.2	Definitions & Abbreviations	4.3.4.1	SelfID Packet
3.2.1	Cable Environment	4.3.4.2	LinkOn Packet
3.3	Addressing	4.3.4.3	PHY Configuration Packet
3.4	Protocol Architecture	4.3.6	Gap Timing
3.4.1	Data Transfer Services	6.2 up to	Packet Formats
3.5	Transaction Layer	6.2.5.2.2	
3.5.1	Transaction Layer Services	7.3.2 up to	Transaction Descriptions
3.5.2	Lock Subcommands	7.3.2.5	
3.6	Link Layer	8.1	Serial Bus Mgmt Summary
3.6.1	Link Layer Services	8.1.1	Node Control
3.6.2	Link and Transaction Layer Interactions	8.1.2	Isochronous Resource Manager
3.6.2.1	Unified Transactions	8.1.4	Bus Manager
3.6.2.2	Split Transactions	8.3.1	Node Capabilities Taxonomy
3.6.2.3	Subaction Concatenation	8.3.2	Command & Status Registers
3.6.3	Asynchronous Arbitration	8.3.2.3	Serial Bus Dependent Registers
3.6.4	Isochronous Arbitration	8.4.2	Bus Configuration Procedures
3.7.2	Fair Arbitration	8.4.3	Isochronous Management
3.7.3.1	Cable Configuration	8.4.5	Speed Management
3.7.3.1.1	Bus Initialize	8.4.5.1	Accessing the Speed Map
3.7.3.1.2	Tree Identify	8.4.6	Topology Management
3.7.3.1.3	Self Identify	8.4.6.1	Accessing the Topology Map
3.7.3.2	Normal Arbitration	Annex E.3	Ph.Layer Configuration Example
3.8	Bus Management	Annex H	Bus Configuration Example

Architecture

Unibrain’s 1394 stack is formed by a set of drivers layered on top of each other. The heart of this stack is the driver that implements the 1394 protocol (the transaction layer) and provides all the services that are needed by client drivers and applications. This is called the 1394 Class Driver (**UB1394.SYS**), often simply referred to as the *Class Driver*.

Beneath the 1394 Class Driver lie one or more 1394 *Miniport Drivers* (for example **UBLYNX.SYS** or **UBOHCI.SYS**). A 1394 miniport driver is assigned very specific tasks that have to do with directly accessing the adapter hardware. The 1394 Class Driver never directly accesses the adapter hardware, but uses the services of a 1394 Miniport Driver in order to do so.

Above the 1394 Class Driver lay one or more *Client Drivers*. 1394 Client Drivers use the services provided by the class driver in order to communicate over the 1394 physical medium. FireNet (**UBFWNET.SYS**) and the Serial Bus Manager are 1394 Client Drivers.



Components like FireNet™ can only be implemented in kernel mode, since FireNet™ also is a network driver, and network drivers necessarily run in kernel mode. The Serial Bus Manager could possibly run as a user mode service, but it was implemented as a kernel driver in order to ensure that it would be able to respond as fast as possible to the various bus events.

In general, any component that is speed-critical, and/or provides some form of programming interface to higher level components (applications or higher level kernel mode components), should be implemented in kernel mode.

Otherwise a user mode application or service (demon) should be all that is required to do whatever is necessary.

Conventions

- All FireAPI functions are prefixed with the **C1394** prefix. The term *C1394XXX* will be used when referring to any of the functions of FireAPI.
- Sample code will be typed in a fixed pitch font.
- Color-coding will be used in the sample code for easier reading. This will be applied to *comments*, *keywords*, *numbers* and *strings*. Additionally *C1394XXX* function calls will be in bold text.
- A variant of Hungarian naming is used for variables. Each variable name is prefixed by one or more characters that identify its type. Additionally, all global variables are prefixed with a ‘g_’ and

all function arguments with a 'a_'. For example:
 uIndex is a local variable of type unsigned long.
 g_uOperations is a global variable of type unsigned long.
 a_puIndex is a parameter that is a pointer to an unsigned long.
 szName is a local string variable.

- When the text refers to function, type and constant names they will appear in this font: **C1394Initialize**.
- When the text refers to file names they will appear in this font: **FireAPI.h**
- When the text refers to structure fields and parameters their names will appear *italicised*.
- Most *C1394XXX* functions need to return error information to the caller. For this reason FireAPI uses the **STATUS_1394** type, which includes several *status codes*. A listing of these codes can be found at the end of this document.
STATUS_1394_SUCCESS is the status code that is used to indicate that an operation is successful. Almost all other status codes are describing an error situation.
- Sample code fragments will usually not include complete error-handling code in order to keep the text simple and small.

Before You Start

- Each source file that uses FireAPI functions must include the header file **FireAPI.h**. All other necessary header files are pulled in by this file.
- You must add the directory where the FireAPI header files reside to your project's include directories.
- You must add the directory where the FireAPI library files reside to your project's library directories.
- You must add **UB1394.LIB** to the link libraries of your project.
- **C1394INITIALIZE.OBJ** is for kernel mode FireAPI clients only. Do not use with applications.
- You must have **UB1394.DLL** installed in a directory where it can be located by your executable. It is suggested that you either copy **UB1394.DLL** to a directory that is in your system's PATH, or that you add the directory where **UB1394.DLL** is installed to your system's PATH.
- The installation program of the ubCore 1394 runtime provided with FireAPI installs the retail version of the drivers (also called *free* drivers). It is strongly suggested that you install the debug drivers and DLL¹ that can be found in the FireAPI Drivers directory.
 The debug binaries provide warning and error messages for **ALL** failed FireAPI operations (invalid parameters, unsuccessful operations etc). This will provide invaluable help during all stages of your development process.
 In order to be effective on this you have to install all the debug binaries (SYS and DLLs). For example, if an operation fails inside **UB1394.DLL** but you only have the debug SYS files installed, then you will not see the related debug messages.
Unibrain will only provide FireAPI developer support if the reported problems are accompanied by the messages produced by the debug binaries, or if the problem only occurs on free drivers but not on debug drivers.

¹ See the FireAPI Installation Guide for information on how to install the debug binaries, and how to view the kernel debugger messages.

Initializing with FireAPI

Initializing and Terminating 1394 Support

The very first thing that a FireAPI application has to do is to call **C1394Initialize**, which performs all the necessary actions to initialize support for 1394. This is the function that will check whether the FireAPI driver stack is installed properly and the appropriate drivers started, and initialize the internal structures that the stack will need.

This call is coupled with a call to **C1394Terminate**, usually when the application exits.

If any *C1394xxx* function is called before **C1394Initialize** or after **C1394Terminate**, then the call will fail.

The sample code below demonstrates the usage of these functions.

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    STATUS_1394 Status1394;

    Status1394 = C1394Initialize();

    if (STATUS_1394_SUCCESS != Status1394)
    {
        printf( "C1394Initialize failed with status %s\n",
               C1394StatusString(Status1394) );
        return -1;
    }

    // Do other processing.
    puts("Processing...");

    C1394Terminate();
    return 0;
}
```

A well behaved application should always check the return value of **C1394Initialize** and handle the failure as appropriate. See the comments section of **C1394Initialize** for more information on the reasons that could make this function fail.

It is not mandatory to call **C1394Terminate** when the application is exiting. For example if an application crashes, then the 1394 stack will automatically perform the operations performed by **C1394Terminate**.

This means that application writers need not worry about calling this function from inside fatal error or abnormal termination handlers.

However it is suggested that developers call this function upon normal application termination.

Multiple Adapter Support

FireAPI supports multiple adapters in the same PC and implements a separate 1394 address space for each adapter.

It would not be correct to implement a single 1394 address space on the PC, because the host computer cannot be considered to be the "1394 Node".

This is because there are data in 1394 Control and Status Registers (CSRs) that logically and technically belong to a specified bus or adapter (BUS_MANAGER_ID register, Configuration ROM information, TOPOLOGY_MAP registers etc).

Since each adapter must have different information on these registers, whose offsets are fixed, it is necessary that the 1394 stack implements a different 1394 address space for each adapter.

Applications and client drivers running on a PC, must open one or more handles to the adapters installed on the PC, in order to be able to perform any 1394 operations through these adapters. In almost all FireAPI calls, the first parameter is the handle that identifies the adapter to the 1394 stack.

Unless an application has specific knowledge that it will execute in a system with 1 adapter, it should be designed to expect more than one installed adapter on a PC. If the application can only work through one adapter at a time, then it should provide the ability to let the user specify over which adapter it wants to run.

Enumerating the installed adapters

The sample code below demonstrates how an application enumerates the adapters installed on the local host:

```
#include <stdio.h>
#include <FireAPI.h>

#define MAX_ADAPTERS 4

main(void)
{
    C1394_GUID AdapterGuid[MAX_ADAPTERS];
    unsigned long I, uAdapters;

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    uAdapters = C1394GetAdapters( AdapterGuid, MAX_ADAPTERS );

    printf( "%u adapters are locally installed.\n", uAdapters );

    for (I=0; I<uAdapters; I++)
    {
        printf("Adapter No.%u GUID : %02X %02X %02X %02X %02X %02X %02X %02X",
            I,
            (unsigned long) AdapterGuid[I].Bytes[0],
            (unsigned long) AdapterGuid[I].Bytes[1],
            (unsigned long) AdapterGuid[I].Bytes[2],
            (unsigned long) AdapterGuid[I].Bytes[3],
            (unsigned long) AdapterGuid[I].Bytes[4],
            (unsigned long) AdapterGuid[I].Bytes[5],
            (unsigned long) AdapterGuid[I].Bytes[6],
            (unsigned long) AdapterGuid[I].Bytes[7]
        );
    }

    C1394Terminate();
    return 0;
}
```

Applications need not worry about what will happen if there are more adapters installed than what they can handle. For example if MAX_ADAPTERS in the code fragment above was 2, and there were 3 installed boards, then C1394GetAdapters would return 2.

This means that an application that is written to work with only one adapter can simply go about as shown below:

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_GUID AdapterGuid;

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    C1394GetAdapters( &AdapterGuid, 1 );

    printf("Adapter GUID : %08X-%08X\n",
        SwapEndian32(*((unsigned long*) AdapterGuid.Bytes)),
        SwapEndian32(*((unsigned long*) &AdapterGuid.Bytes[4]))
    );

    C1394Terminate();
    return 0;
}
```

This sample also demonstrates the use of FireAPI function **SwapEndian32**, that can be used to byte swap a 32-bit value.

Opening a handle to an adapter

The following sample demonstrates the simplest way to open a handle to an adapter, which is to open a handle to the *default* adapter. If there is only one adapter installed, then this is also the *default* adapter. If there are more than one adapters installed, then the default adapter is the one that was enumerated first by the 1394 stack².

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE C1394AdapterHandle;
    STATUS_1394          Status1394;

    C1394AdapterHandle = NULL;

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    Status1394 = C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

    if (STATUS_1394_SUCCESS != Status1394)
        return -2;

    // Do the rest of the processing.
    // ...

    C1394CloseAdapter( C1394AdapterHandle );
    C1394Terminate();
    return 0;
}
```

As is the case with **C1394Terminate**, it is not mandatory to call **C1394CloseAdapter** when the application is exiting. For example if an application crashes, then the 1394 stack will automatically perform the operations performed by **C1394CloseAdapter**. This means that application writers need not worry about calling this function from inside fatal error or abnormal termination handlers. However it is suggested that developers call this function upon normal application termination.

² In future versions of FireAPI, when there are multiple adapters installed it will be possible to specify which adapter will be the *default* through registry settings.

Opening an adapter by GUID

The sample below demonstrates how to open a handle to all the adapters that are installed on the local workstation.

```
#include <stdio.h>
#include <FireAPI.h>

#define MAX_ADAPTERS 4

main(void)
{
    C1394_GUID          AdapterGuid[MAX_ADAPTERS];
    C1394_ADAPTER_HANDLE AdapterHandle[MAX_ADAPTERS];
    STATUS_1394        Status1394;
    unsigned long I, uAdapters;

    // Set everything to NULL.
    memset( AdapterGuid, sizeof(AdapterGuid), 0);
    memset( AdapterHandle, sizeof(AdapterHandle), 0);

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    uAdapters = C1394GetAdapters( AdapterGuid, MAX_ADAPTERS );

    for (I=0; I<uAdapters; I++)
    {
        Status1394 = C1394OpenAdapter( &AdapterGuid[I], NULL, &AdapterHandle[I]);

        if (STATUS_1394_SUCCESS != Status1394)
        {
            printf("C1394OpenAdapter returned %s\n", C1394StatusString(Status1394));
            goto Cleanup;
        }
    }

    ////////////
Cleanup:;
    for (I=0; I<uAdapters; I++)
        if (NULL != AdapterHandle[I])
            C1394CloseAdapter( AdapterHandle[I] );

    C1394Terminate();
    return 0;
}
```

For more information on **C1394OpenAdapter** see the description of this function.

Receiving notifications about the dynamic addition of 1394 adapters

The sample below demonstrates how to receive notifications about the addition of 1394 adapters in the system. This process happens through the Plug-n-Play functionality of the operating system. A disabled 1394 adapter might be enabled by the user through the Device Manager, or a PCMCIA 1394 adapter might be added, in the case of a laptop.

```
#include <stdio.h>
#include <FireAPI.h>

#define MAX_ADAPTERS 4

main(void)
{
    HANDLE                hWaitEvent;
    C1394_GUID            AdapterGuid[MAX_ADAPTERS];
    C1394_ADAPTER_HANDLE  AdapterHandle[MAX_ADAPTERS];
    DWORD                WaitStatus;
    STATUS_1394          Status1394;
    unsigned long        I, uAdapters;

    // Set everything to NULL.
    memset(AdapterGuid, sizeof(AdapterGuid), 0);
    memset(AdapterHandle, sizeof(AdapterHandle), 0);

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    hWaitEvent = C1394GetAddAdapterEventHandle();
    WaitStatus = WaitForSingleObject( hWaitEvent, FALSE, INFINITE );

    if(WAIT_OBJECT_0 == WaitStatus)
    {
        uAdapters = C1394GetAdapters( AdapterGuid, MAX_ADAPTERS );

        for (I=0; I<uAdapters; I++)
        {
            Status1394 = C1394OpenAdapter( &AdapterGuid[I], NULL, &AdapterHandle[I]);

            if (STATUS_1394_SUCCESS != Status1394)
            {
                printf("C1394OpenAdapter returned %s\n",
                    C1394StatusString(Status1394));
                goto Cleanup;
            }
        }
    }

    ////////////
Cleanup:;
    for (I=0; I<uAdapters; I++)
        if (NULL != AdapterHandle[I])
            C1394CloseAdapter( AdapterHandle[I] );

    C1394Terminate();
    return 0;
}
```

Note that the AddAdapter event handle is not initially signalled, so the code above will wait infinitely if there is no new adapter added to the system. An application typically opens any adapters present when it is starting up and then monitors the event for the addition of new adapters. When new adapters are added the application should retrieve the GUIDs of all local adapters, compare them with those it has already opened and locate itself the GUID of the new adapter.

For more information on **C1394GetAddAdapterEventHandle** see the description of this function.

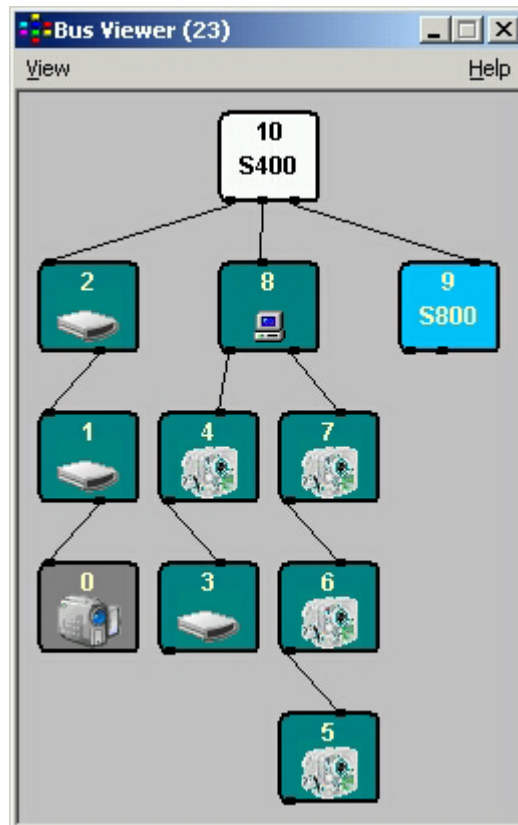
Asynchronous Operations

Initiating Transaction Requests

Maximum Transmission Speed Per Destination NodeID

One of the fundamental characteristics of 1394 is that the PHY chips automatically act as physical repeaters, even if the node is unpowered or has its link layer turned off.

However due to the technical characteristics of 1394, nodes that operate at lower speeds cannot repeat higher speed transmissions. This way they act as a *traffic bottleneck*.



According to the original 1394 standard (IEEE1394-1995), after a bus reset takes place the Serial Bus Manager analyzes the configuration of the bus, calculates a speed map which contains the maximum speed of communication between any two nodes on the bus, and publishes this information in the SPEED_MAP registers.

The rest of the nodes on the bus can then read these values and find out what is the highest possible transmission rate at which they can communicate with each other.

The SPEED_MAP registers have been obsoleted by the IEEE1394a-2000 standard, so this option is no longer available to applications.

In 1394a and later, it is possible for each port on an adapter to operate on a different speed than the link chip. The 1394a standard provides the means, by remote PHY register reads, to discover the speed that each port runs at, and using topology information make it possible for a node to build a speed map for the current topology of the 1394 bus.

The 1394 Class Driver at every node performs this analysis and can readily provide path-speed information to any application running on the PC.

This information is available to any application as soon as a bus reset is complete through the **C1394GetMaxSpeedToNode** and **C1394GetMaxSpeedBetweenNodes** functions.

Broadcast packets should be transmitted at the speed of the slowest device on the bus, so that all devices can receive the packet.

All transmission requests made to the 1394 Class Driver can contain the speed at which the transmission should be made. However most high-level APIs do not take a speed parameter but internally automatically select the maximum appropriate speed for the destination node.

Maximum Asynchronous Packet Size

The maximum size of outgoing asynchronous packets towards some node is dependent not only on the maximum transmission speed on the path to that node, but also on the value of the *max_rec* field in the destination node's *Bus_Info_Block*.

Note that this limits the size of block write transactions that the target node can receive, and the size of block read responses that the target node can send. It does not necessarily limit the size of block write transactions that node can transmit.

The class driver provides the **C1394GetMaxPayloadForSpeed** and **C1394GetMaxPayloadForMaxRec** functions in order to perform these calculations.

Currently the class driver does not take the *max_rec* field of the target node into consideration for limiting the size of outgoing requests, but only uses the speed of the path to the target node. This means for example that the class driver will fail³ an asynchronous block write of 1600 bytes or an asynchronous block read of 1600 bytes to a destination that can be reached only at S200.

In future versions the class driver very shortly after a bus reset will read the *max_rec* field of all nodes, and then combine this information with the path-speed information in order to calculate the maximum asynchronous packet size for that node.

The tables below are taken from the 1394 standards, and can be used as references for the maximum asynchronous block size depending on the speed, and *max_rec*.

Data Rate	Maximum Data Payload (bytes)
S100	512
S200	1024
S400	2048
S800	4096

Table 1. Maximum payload size for primary packets with data block payload

The table above applies to asynchronous packets with data block payload. The limit for isochronous packets is different (the double).

³ This means that the packet will not be transmitted at all, and a status code of **STATUS_1394_SIZE_LIMITATION** will be returned.

The encoding of the *max_rec* field is shown in the table below.

Code	Maximum Data Payload (bytes)
0	Not specified
1	4
2	8
3	16
4	32
5	64
6	128
7	256
8	512
9	1024
A ₁₆	2048
B ₁₆	4096
C ₁₆	8192
D ₁₆	16384
E ₁₆	Reserved
F ₁₆	Reserved

Table 2. Encoding of max_rec field

Bandwidth Consumption

The bandwidth that is necessary to transmit a packet, does not only depend on the size of the packet, but also on the speed on which it is transmitted. As the transmission rate increases, the required bandwidth is halved. This is true both for asynchronous and isochronous transmissions.

For example it takes the double bandwidth to transmit a 512-byte packet at S100, than it takes to transmit it at S200, and it takes double the bandwidth to transmit it at S200 than it takes at S400.

This means that transmitting a 512-byte packet at S100, consumes the same bandwidth⁴ as a 1024-byte packet at S200 or a 2048-byte packet as S400.

It is obvious that transmissions at a rate lower than the required one are a waste of bandwidth and applications should avoid them.

Broadcast Requests

Broadcast requests have a destination node ID equal to 63_{10} ($3F_{16}$). Only write transactions (block or quadlet) can be sent as broadcasts.

The class driver guarantees that a broadcast transaction uses a transaction label that is not in use at the moment of transmission for all nodes on the bus. This is a very subtle detail of the 1394 protocol. It is a requirement that a node must not have at the same time pending to a node on the bus more than one transaction with the same transaction label. On a bus with heavy asynchronous traffic this phenomenon will certainly manifest and it would be practically impossible to diagnose or deal with. The class driver efficiently deals with this issue and relieves all applications from the possibility of such an error. If that was not the case, then there would always be the possibility that a node rejected the broadcast packet, and an application would have to design complex recovery procedures to deal with such an error.

Broadcasts are by default sent at the *broadcast speed*, which is the speed of the slowest device on the bus. This is done so that all devices can receive the transmission. However it is possible for an application to send a broadcast at a speed higher than the *broadcast speed*. In this case the application should know the bus topology, and make sure that it does not care for the nodes that will miss the broadcast. Given this fact, there are two reasons why an application would want to broadcast at a higher rate:

- Send a packet that is bigger than what can be transmitted at the *broadcast speed*.
- Consume less 1394 bandwidth.

Software Loopback

The miniport drivers provide completely transparent loopback functionality, and of course the class driver supports it as well.

This means that an application can access 1394 registers owned by other applications or drivers on the same machine. This is done through 1394 transactions, as if the target register lies on a different node.

This feature is very useful for applications, because it allows applications that communicate over 1394 to be developed and tested on a single PC before they are tested on the network, and of course allows such applications to run on the same PC even in production environments.

Broadcasts are also loopbacked, but they must be specifically permitted for when a 1394 address range is created. For more information refer to the discussion of the **ACCESS_BROADCAST_LOOPBACK** flag in the description of function **C1394MapAddressRange**.

Raw transmissions (function **C1394TransmitRaw**) are not loopbacked.

⁴ Actually even more, because the 20 byte header should also be taken into consideration.

Transaction Management

Transaction management is involved with implementing the 1394 transaction layer according to the rules specified in the 1394 standards. The rules that must be followed are listed below:

- There are 64 transaction labels.
- A node should not have at any moment two pending split transaction requests with the same transaction label transmitted to the same node.
- A node should not accept at any moment two pending split transactions from the same node with the same transaction label.
- A response to a split transaction request should be received within the time interval described by the value of the SPLIT_TIMEOUT register. Otherwise, the split transaction should be declared *timed out*, and a late response should be discarded.
- When a split transaction is timed out, the transaction label should not be reused for a transaction to the same node until twice the amount of time specified by the SPLIT_TIMEOUT register has elapsed.
- A quadlet transaction is only allowed on a quadlet aligned address.
- The source NodeID in a transaction response should be exactly the same as the destination NodeID in the corresponding transaction request. The class driver provides a configuration option that relaxes this requirement and allows 3FF₁₆ and the current bus ID of the local bus to be used interchangeably. This is meant for use with hardware devices that always respond using 3FF₁₆ as the bus ID no matter what was the bus ID in the request packet.
- The data length of a transaction response should be exactly the same as the data length specified in the corresponding transaction request.
- A block write response with data length equal to 4 can only be sent as an answer to a block write request with data length equal to 4.
- A quadlet write response can only be sent for a quadlet write request.
- The above imply that the transaction label used in a broadcast transaction (destination NodeID==63) must not be pending on any node.

An additional rule that will be added to prevent abnormal conditions is that if a read request is received with a *data_length* field greater than what the node can transmit to the source node (due to packet size limitations to that node), the transaction will be considered invalid and will be completed with a *resp_type_error*.

Transaction Label Management for Outgoing Transaction Requests

The transaction label field in a packet header uses 6 bits. This means that a node can have a maximum of 64 split transaction requests pending to a given node. If clients make more than 64 concurrent transmission requests to a given destination node, then some of them will fail to allocate a transaction label.

This is an error condition that might occur frequently on a bus with heavy asynchronous traffic, and will only last for a short time.

The FireAPI class driver makes transaction label management completely transparent to applications. An application never specifies the transaction label to be used on a transaction request.

In order to achieve this, the class driver maintains a queue of pending transmit requests for the destination node that need to allocate a transaction label. Whenever transaction labels are freed, queued transmit requests can be transmitted.

So applications never need to worry about transaction labels.

Transmit Order

In general, whenever a client wants to be sure of the processing order of two transactions, it should transmit the first, get an acknowledgement and a response and then transmit the second.

Additionally the following are performed by the class driver:

- Raw transmit requests are performed immediately, without regards to any other transmit requests that might be queued in various places within the class driver.
- NodeID-based transaction request packets that use a pre-allocated transaction label⁵ are transmitted immediately. This means that they might be transmitted before any older NodeID-based transaction request packets that are still waiting for a transaction label at the destination node's transaction label queue.
- An urgent⁶ NodeID-based transaction request packet for node X that must allocate a transaction label is always transmitted before any non-urgent packets for node X that were already queued in node X transaction label queue when the urgent request was submitted.
- NodeID-based transmits for different destination node IDs transmit independently of each other. If a client submits two packets to the class driver, the first for node W and the second for node Y, then the following transmit-order scenarios are possible:
 1. Both destination nodes' transaction label queues are empty and a transaction label can be allocated for both packets. Then the packet for W is transmitted first and the packet for Y is transmitted second.
 2. One of the two queues is busy while the other is not. The packet for the destination node with the non-busy queue is transmitted first.
 3. Both destination node queues are busy. Then the order of transmission is random. The packet that will reach the top of its transaction label queue first, will transmit first.
- Transaction response packets for a node are in principle independent of transaction requests for the same node. A response packet for destination node X, may transmit before one or more older transaction request packets that were submitted for node X, because these packets might be waiting in the transaction label queue for node X.
- All handle-based transaction request packets from a given handle, always maintain their relative transmit order, even if a bus reset takes place.
- Transmit requests from different handles are completely independent of each other, even if both handles refer to the same destination node. In general if handles H1 and H2 refer to the same node, then transmits through these handles will be ordered by time of submission, but this might change when a bus reset takes place, because these requests will be requeued on a handle basis.
- In a transmit call that specifies more than one packets, the packets are transmitted in the order they are found inside the packet array, with the exceptions described above.

⁵ Pre-allocated transaction label are only available for kernel mode FireAPI clients.

⁶ Urgent transaction requests are only available for kernel mode FireAPI clients.

NodeID & Device Functions

The 1394-1995 standard requires that every node has a 64-bit unique node identifier, call *Globally Unique Identifier* or GUID⁷ in their *Bus_Info_Block*.

One of the basic characteristics of 1394 is the fact that it supports plug-n-play by automatically reconfiguring physical device addresses when one or more devices are added or removed from the 1394 bus.

However this results into a reassignment of the physical IDs to the devices on the bus. A device can only be *permanently* identified by its GUID, which is constant through the device's lifetime.

After a bus reset, software should in principle reidentify the device, by enumerating the active nodes on the bus and locating the device it was working with before the bus reset.

When one node wants to transmit a packet to another node, in the lowest level the drivers must use the target node's 16-bit NodeID.

The 16-bit NodeIDs of 1394 are dynamically configurable and can change any time a bus reset occurs. If the bus reset is software initiated and a *forceroot* PHY configuration packet has not been sent specifying for root a node other than the current root, then the bus configuration stay the same and the NodeIDs will not change. However, if the root changes or the bus configuration changes (by adding or removing devices from the bus) then the NodeIDs of most devices will change.

Dealing correctly with this dynamic reconfiguration of the bus is of primary importance to the correct operation of applications. The easiest way for applications to deal with this problem is to bypass it, by using *Device Handles*. A device handle is a way to identify a 1394 device by its GUID.

The user mode interface of FireAPI implements two types of functions that developers can use to access a remote node:

- The *NodeID* functions that accept the 16-bit NodeID of the target device.
- The *Device* functions that accept a handle to the target device. These are only available in user mode (sample source code is provided in this document).

In the lowest level *Device* functions resolve to the appropriate *NodeID* function call so we will have to first analyze the operation of *NodeID* functions. The *NodeID* functions are listed in the table below.

Function Name	Description
C1394ReadNode C1394WriteNode C1394LockNode	Blocking calls that perform a single read, write or lock transaction with a destination node.
C1394ReadNodeAsynch C1394WriteNodeAsynch C1394LockNodeAsynch	Non-blocking calls that perform a single read, write or lock transactions with a destination node.
C1394TransmitPackets	Non-blocking call that can send multiple transaction requests to various nodes.
C1394ReadNodeEx C1394WriteNodeEx C1394LockNodeEx	Blocking calls that perform a single read, write or lock transaction with a destination node.
C1394ReadNodeAsynchEx C1394WriteNodeAsynchEx C1394LockNodeAsynchEx	Non-blocking calls that perform a single read, write or lock transactions with a destination node.
C1394TransmitPacketsEx	Non-blocking call that can send multiple transaction requests to various nodes.

⁷ Also known as EUI-64 (Extended Unique Identifier, 64 bits).

The Device functions are listed in the table below.

Function Name	Description
C1394OpenDevice C1394CloseDevice	Open/Close a handle to a device.
C1394ReadDevice C1394WriteDevice	Blocking calls that perform a single read or write transaction with a destination device. Lock transactions are NOT currently supported by the API.
C1394GetDeviceNodeid	Returns the current Node Id of the particular device.

Bus Resets & Asynchronous Transactions

The 1394 stack maintains the *System Bus Reset Count*, which indicates the number of bus resets that have occurred since the 1394 stack loaded. The value of the *System Bus Reset Count* is returned by calling **C1394GetBusResetCount**.

FireAPI requires that each *NodeID* packet transmission is associated with a *Bus Reset Count*.

If the *Bus Reset Count* associated with a packet transmission is not the same as the *System Bus Reset Count* then the 1394 stack will fail the transmission with **STATUS_1394_BUS_RESET**.

Transaction Requests are given their *Bus Reset Count* by the application, while *Transaction Responses* should use the *Bus Reset Count* of the corresponding *Transaction Request* packet.

This in general means that applications should maintain some kind of *Application Bus Reset Count* that they should use as the *Bus Reset Count* of any packet they are trying to transmit.

Kernel mode FireAPI clients are required to maintain such a *Bus Reset Count*, since this is a parameter to the kernel mode **C1394ReadNode** and **C1394WriteNode** functions⁸.

Because user mode FireAPI applications were not originally required to maintain such a *Bus Reset Count*, the 1394 stack by default internally maintains and automatically updates such a count for each application.

If an application wishes to maintain and update the bus reset count by itself, it should call **C1394AcknowledgeBusReset** for the adapter as soon as it opens the adapter with **C1394OpenAdapter**. From that point on, the 1394 stack will **NOT** update automatically the internal *Bus Reset Count* that it maintains for the application.

When a bus reset occurs then the application's *Bus Reset Count* will become different from the *System Bus Reset Count*. The next time the application tries to transmit a packet using any of the standard transaction request function (**C1394ReadNode**, **C1394WriteNode**, **C1394LockNode**, **C1394ReadNodeAsynch**, **C1394WriteNodeAsynch**, **C1394LockNodeAsynch** and **C1394TransmitPackets**) the function will return **STATUS_1394_BUS_RESET**.

The application must call **C1394AcknowledgeBusReset** in order to be able to transmit a transaction request again. This will synchronize the application's *Bus Reset Count* to the *System Bus Reset Count*.

This way the application can realize that a bus reset has occurred since the last time it sent a transaction request, and it will be prevented from transmitting any new requests until it calls **C1394AcknowledgeBusReset**. With this call the application can be considered to inform the 1394 stack that it has taken all necessary actions to update its internal structures based on the new bus topology.

Bus Reset Exceptions (obsolete)

An optional facility that used to be available to applications prior to ubCore 5.50 was the ability of **UB1394.DLL** to raise a SEH exception instead of returning **STATUS_1394_BUS_RESET**.

This was supposed to make the application code much easier to write and understand because the ‘exceptional’ event is handled outside the main execution flow of the program.

This was controlled on a per-process basis, through the use of **C1394SetInformation** and the **OID_BUS_RESET_EXCEPTIONS** identifier. An exception would only be raised by **C1394ReadNode**, **C1394WriteNode** and **C1394LockNode**. The non-blocking versions of these functions would not raise an exception.

The exception code that was raised was configurable by the application, and is specified in the call to **C1394SetInformation**. The exception raised by UB1394.DLL was *continuable*. This means that the application’s exception filter could do all the required processing to update the state of the application and then return **EXCEPTION_CONTINUE_EXECUTION**. As a consequence the function that raised the exception would continue normally its operation and return **STATUS_1394_BUS_RESET**.

For more information on exceptions see the documentation of the *Microsoft Platform SDK* under *Base Services – Debugging & Error Handling – Structured Exception Handling*.

The support for Bus Reset Exceptions was discontinued in version 5.50 of ubCore, since it was proved that:

- Practically noone used it.
- It can cause much more harm than benefit to an application developer. Since the setting is process-wide, if a third party Firewire component (DLL) gets loaded into a process and enables Bus Reset Exceptions then it will break all other code that was written to expect return error codes and does not have exception handlers.
- It greatly complicates development for both Unibrain and partners. It is extremely difficult to write code for a component of any sort that will run correctly on both the *return error code* and *bus reset exception* models. The code will either have to be written twice (leading to a maintenance nightmare), or it will most likely contain errors.

Reading the GUID of bus nodes

The following sample demonstrates the usage of **C1394ReadNode**. The code tries to read the GUID of each node on the bus.

```

#include <stdio.h>
#include <FireAPI.h>

void ReadNodeGuid( C1394_ADAPTER_HANDLE  a_C1394AdapterHandle,
                  C1394_NODE_ID        a_NodeID )
{
    STATUS_1394          Status1394;
    C1394_GUID           NodeGuid;
    C1394_RESPONSE_CODE ResponseCode;
    C1394_ACK_CODE       AcknowledgeCode;

    Status1394 = C1394ReadNode( a_C1394AdapterHandle,
                               a_NodeID,
                               CSR_BUS_INFO_BLOCK+8,
                               8,
                               &NodeGuid,
                               &ResponseCode,
                               &AcknowledgeCode );

    switch (Status1394)
    {
        //-----
        case STATUS_1394_SUCCESS:
            printf( "GUID of Node %u.%u: %08X-%08X\n",
                   (ULONG) a_NodeID.BusID,
                   (ULONG) a_NodeID.PhysicalID,
                   SwapEndian32(*( (ULONG*) NodeGuid.Bytes)),
                   SwapEndian32(*( (ULONG*) &NodeGuid.Bytes[4])) );

            break;

        //-----
        case STATUS_1394_TIMEOUT:
            printf( "Block Read Request timeout for Node %u.%u\n",
                   (ULONG) a_NodeID.BusID,
                   (ULONG) a_NodeID.PhysicalID );

            break;

        //-----
        case STATUS_1394_NOT_FOUND:
            printf( "No acknowledge from Node %u.%u\n",
                   (ULONG) a_NodeID.BusID,
                   (ULONG) a_NodeID.PhysicalID );

            break;

        //-----
        case STATUS_1394_TRANSACTION_FAILED:
            printf( "Transaction FAILED for Node %u.%u. Response code is %s\n",
                   (ULONG) a_NodeID.BusID,
                   (ULONG) a_NodeID.PhysicalID,
                   C1394RespCodeString(ResponseCode) );

            break;

        //-----
        default:
            printf( "Block Read FAILED for Node %u.%u with status %s\n",
                   (ULONG) a_NodeID.BusID,
                   (ULONG) a_NodeID.PhysicalID,
                   C1394StatusString(Status1394) );
    }
}

```

```

/*****
main(void)
{
    C1394_ADAPTER_HANDLE  C1394AdapterHandle;
    C1394_NODE_ID         NodeID;
    ULONG                 I;

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

    NodeID.BusID = LOCAL_1394_BUS_ID;

    for (I=0; I<63; I++)
    {
        NodeID.PhysicalID = (C1394_PHYSICAL_ID) I;
        ReadNodeGuid(C1394AdapterHandle, NodeID);
    }

    C1394CloseAdapter( C1394AdapterHandle );
    C1394Terminate();
    return 0;
}

```

The usage of **C1394ReadNode** itself is quite simple. You must specify the adapter handle, the destination NodeID, the 1394 address space offset, the number of bytes to read and the buffer into which to read the data.

The function returns a status code that describes the result of the transaction, and optionally the response code and acknowledge code (in the cases where the transaction has not completed successfully).

Reading the GUID of bus nodes – Transaction Failures

With regards to the previous sample, there is more to be said about the handling of the unsuccessful return values, rather than **C1394ReadNode** itself. There are a lot of error return-codes for all *asynchronous transaction request* functions because there are a lot of things that can possibly go wrong with a 1394 transaction request:

- The destination node is turned off and did not even acknowledge the request.
- The destination node is not present on the bus.
- The destination node acknowledges the request with `ack_pending` but never sends a response packet. The transaction request will timeout.
- The destination node acknowledges the request but sends a response packet with a response code other than `resp_complete`. This means that the destination node did not accept the transaction.
- The destination node acknowledges the request with `ack_pending` but a bus reset occurred before the response packet was delivered.
- A bus reset occurred at the time that the transmission of the request packet was about to be executed.
- A transmission error occurred.

Although it might at first appear a little bit puzzling that the code has to check for and handle all those different error codes, things are not difficult in practice. Usually the application knows which error codes it has to expect from the node it communicates with and need only handle separately these codes, and let the rest fall into the default case.

For example, it is quite unusual for a transaction to fail in a real application, or at least fail unpredictably. Some reasons why a read transaction could fail are:

- An invalid/inaccessible 1394 offset was specified. This should cause a read response with the `resp_address_error` response code.
- The target offset is valid, but does not support the requested transaction type. For example, the register only supports quadlet reads and the application attempted a block read. Depending on the type of the target device, it might either respond back with a response packet that contains the response code `resp_type_error`, or directly acknowledge the transaction request with `ack_type_error`.

However this is an expected type of failure, so the code could be written to check for the returned response code (`resp_type_error`) and retry the GUID read using 2 quadlet reads instead. OHCI 1394 host adapters as well as some other 1394 devices allow only quadlet reads in the configuration ROM so this failure is not at all uncommon.

The program listed above will perform its intended operation only if no bus reset occurs during the read loop. This is because the program does not call **C1394AcknowledgeBusReset** for the adapter it opens. The sample code in section [Enumerating the Devices on the 1394 Bus](#) demonstrates how to enumerate the bus correctly with regards to bus resets.

Reading the GUID of bus nodes – Transaction Timeouts

The above are strictly-speaking *transaction failures*. A timeout is another type of failure that is not strictly speaking a *failed transaction* but an *incomplete transaction*. A transaction that failed for the reasons listed above, will fail again and again no matter how many times you retry it. A transaction that timed out will probably succeed if retried by the software.

The minimum timeout for 1394 transactions is 100 msec according to 1394a-2000. This is the default value used by the Unibrain drivers. The Serial Bus Manager driver makes sure that all nodes on the bus have the same value for their split transaction timeout (SPLIT_TIMEOUT register of 1394). This is performed by default in a “passive” manner: if the SPLIT_TIMEOUT register is written on the node where the currently active (elected) Bus Manager resides, then the Serial Bus Manager will immediately proceed to write the SPLIT_TIMEOUT register of all nodes on the bus. However immediately after a bus reset takes place the Serial Bus Manager will not try to rewrite the SPLIT_TIMEOUT register of all nodes. To enable this behaviour you have to use the *SplitTimeoutOnBusReset* registry setting.

Timeouts are not occurring often in practice and when they happen they are more likely to happen on block read transactions. Applications that have to transfer big amounts of data usually perform *unified block writes*, which are block write transactions that are directly acknowledged by the hardware with *ack_complete* and have no response packet.

When communicating with a ‘simple’ device (a camera or some other embedded device that does not have serious computational power) a timeout is usually an indication that the target node never actually sent a response, rather than a response was sent late.

For example some digital 1394 cameras often fail to respond to configuration ROM reads when they are running at resolution 640x480, YUV 4:2:2, at 30 fps. When they are running at lower resolutions or lower frame rates this never occurs. At this rate however, the device transmits 18MB/sec, and it is so busy performing its camera-tasks that it often fails to respond to control requests like configuration ROM reads which are treated as ‘low-priority’ by the device.

Moreover such devices often get confused if they receive more than one transaction requests at the same time, and end up not responding to any of the two. This should be taken into account especially for post bus-reset processing, when many nodes may try to enumerate the bus at the same time.

For example the Class Driver uses a special policy when enumerating the bus after the bus reset in order to prevent device saturation. Instead of starting from NodeID 0 and moving upwards, it starts from its own NodeID, moves upwards towards the highest numbered present node, and then starts over from zero moving up to its own NodeID. This way if more than one PCs are present on the bus, chances are significantly reduced that they will be enumerating the same device at the same moment.

When communicating between PCs a timeout may occur if the target system is so busy with high priority tasks, that the responding application did not get the chance to process its transaction requests in time. However this is not very likely to happen in most systems.

There are 2 ways that application designers can deal with timeouts, if they found out that they indeed occur in their ‘system’:

- Increase the SPLIT_TIMEOUT value, so that timeouts occur less often.
- Write code to retry transactions that were timed out.

The first solution is better in the case of a heavily-loaded system that might need a bigger timeout value in order to be able to successfully complete transactions in the first place. For example if the timeout is 100msec (the minimum allowed) a system with heavy CPU load may not be able to complete read transactions in time at all.

The drawback of increasing the timeout value is that if timeouts still occur, then the application has to wait longer for each operation that times out, and it might appear non-responsive to the user.

In general, when developing a system with FireAPI, the designers will find out whether there is any issue with transaction timeouts, and if so decide which actions to take around their problem.

Reading the GUID of bus nodes – Various Notes

The sample program also implicitly demonstrates the loopback functionality provided by the API. The code does not perform any checks so that it does not send a transaction to the local node. It simply proceeds to read from the local node⁹ in the same way it does for remote nodes.

When a node sends a transaction to a non-existent node, then the transaction request is not acknowledged at all. In 1394 terminology the “*acknowledge is missing*” situation is often described as “*ack_missing was received*” or “*the request was acknowledged with ack_missing*”.

Transaction requests are acknowledged with `ack_missing` not only for transactions sent to non-existent nodes, but also for existent nodes whose link layer is not active.

The return of information for the response code and the acknowledge code is optional. In most cases applications specify NULL in the last two parameters to **C1394ReadNode**.

Another point to keep in mind is that the previous sample obviously does not demonstrate the best way to enumerate the GUID of bus nodes. In most cases the bus will have much less than 63 nodes present, so attempting read transaction for all 63 nodes produces unnecessary traffic on the bus.

It might appear that a couple of dozens of extra requests are simply no big deal since the extra traffic load they cause is practically zero compared to the capabilities of 1394. While this is certainly true, there is another reason why unnecessary traffic should be avoided, that escapes new 1394 developers.

This has to do with the logical analysis of the output of 1394 bus analyzers. If each node performs several unnecessary transactions then many more packets appear in the analyzer’s capture log, and it is not long before it starts getting more and more difficult looking through these logs for the explanation of why your system is not logically functioning as expected.

⁹ FireAPI also provides an alternative way to get the local node GUID, through the **C1394QueryInformation** function.

Determining the nodes connected to the 1394 Bus

The 1394 specification requires that each device connected to the 1394 bus should transmit its SelfID packet after a bus reset in order to identify itself to the other nodes on the bus. The format of SelfID packets is described in paragraph 4.3.4 of 1394-1995.

The information provided in the SelfID packet is quite limited, but is enough for many purposes. Each SelfID packet among others contains the following:

- The physical ID of the node.
- The *gap_count* of the node.
- The *LinkOn* bit which indicates whether the node has its link layer activated (which means that it can accept transactions).
- The *Contender* bit which indicates whether the node is a contender for the role of Isochronous Resource Manager (IRM).
- The speed of the node.
- The state of each of its ports (disconnected, connected to parent, connected to child, non present).

FireAPI presents this information to applications in various ways.

For example a 64-bit mask is built, which has a bit set for each node physically present on the bus¹⁰. If bit 7 is set, then there is a node on the bus whose physical ID (PhyID) is seven¹¹. This mask is available through function **C1394QueryInformation** with the **OID_PHYSICAL_NODES** identifier.

Similarly, another 64-bit mask is being built which has a bit set for each node on the bus that has the *LinkOn* bit set in its SelfID packet. This mask is available through function **C1394QueryInformation** with the **OID_LINK_ON_NODES** identifier.

The speed of a node can be determined by using the **C1394GetNodeSpeed** function.

The status of its ports can be determined through **C1394QueryInformation** with the **OID_BUS_TOPOLOGY** identifier.

FireAPI includes other calls through which an application can find out how many nodes are on the bus, and which of those have their link layers active, so that it can proceed to read only from them. Such a method is demonstrated in the following sample:

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE    C1394AdapterHandle;
    ULONGLONG               UPhysicalNodes;
    ULONGLONG               ULinkOnNodes;
    ULONGLONG               UContenderNodes;
    ULONG                   I;

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );
```

¹⁰ These are the nodes that transmitted a selfID packet. A device that cannot not power its PHY chip from the 1394 bus will not cause a bus reset when connected to the bus and will not transmit SelfID packets after subsequent bus resets, so will not appear to be connected to the bus.

¹¹ In 1394-1995 PhyIDs are contiguous, which means that if a node exists with PhyID *x*, then nodes with PhyIDs from 0 to *x-1* also exist on the bus. However there is a remote possibility that this might change in the future when the power management procedures allow the creation of *suspended domains*. For this reason FireAPI provides this information as a general purpose 64-bit mask, instead of returning the number of nodes on the bus and implying the physical IDs.

```

// Get information about the nodes physically present on the local 1394 bus.
*((PC1394_BUS_ID)&UPhysicalNodes) = LOCAL_1394_BUS_ID;
C1394QueryInformation( C1394AdapterHandle,
                      OID_PHYSICAL_NODES,
                      &UPhysicalNodes,
                      sizeof(UPhysicalNodes),
                      NULL,
                      NULL );

printf("Physically Present:");
for (I=0; I<63; I++)
  if ( (((ULONGLONG)1)<<I) & UPhysicalNodes )
    printf("%3u", I);

printf("\n");

// Which nodes on the local 1394 bus have the LinkOn bit set?
*((PC1394_BUS_ID)&ULinkOnNodes) = LOCAL_1394_BUS_ID;
C1394QueryInformation( C1394AdapterHandle,
                      OID_LINK_ON_NODES,
                      &ULinkOnNodes,
                      sizeof(ULinkOnNodes),
                      NULL,
                      NULL );

printf("Link On Nodes      :");
for (I=0; I<63; I++)
  if ( (((ULONGLONG)1)<<I) & ULinkOnNodes )
    printf("%3u", I);

printf("\n");

// Which nodes on the local 1394 bus have the Contender bit set?
*((PC1394_BUS_ID)&UContenderNodes) = LOCAL_1394_BUS_ID;
C1394QueryInformation( C1394AdapterHandle,
                      OID_CONTENDER_NODES,
                      &UContenderNodes,
                      sizeof(UContenderNodes),
                      NULL,
                      NULL );

printf("Contender Nodes    :");
for (I=0; I<63; I++)
  if ( (((ULONGLONG)1)<<I) & UContenderNodes )
    printf("%3u", I);

printf("\n");

C1394CloseAdapter( C1394AdapterHandle );
C1394Terminate();
return 0;
}

```

Keep in mind that depending on the power-up state of a 1394 device, it might be the case that the device has its *LinkOn* bit set to one but not actually have its link layer active. This means that any transaction requests sent to them will not be acknowledged (or in FireAPI terminology: *acknowledged with ack_missing*).

For example consider a PC with a 1394 adapter that is connected to the 1394 bus, but not turned on. The 1394 adapter powers its PHY chip from the 1394 bus, but not its link layer. Typically when a bus reset occurs the *LinkOn* bit in the selfID packet of the adapter should be zero, but some adapters may have it set to one. This might also occur with any other self-powered device that might be unpowered but still connected to the 1394 bus.

Technically this is a violation of the 1394 specification, but developers should be prepared to meet this condition in practice.

Enumerating the Devices on the 1394 Bus

The sample code below demonstrates how to enumerate the devices on the bus in a *bus-reset safe* manner. The delays in the execution are intentionally put into the code so that you can try the behaviour of this program. Open up a window with CMD1394, start this program and while it is executing the inner loop initiate a bus reset.

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE    C1394AdapterHandle;
    ULONGLONG               UPhysicalNodes;
    ULONGLONG               ULinkOnNodes;
    ULONGLONG               UContenderNodes;
    C1394_GUID              DeviceGuidArray[63];
    C1394_NODE_ID           NodeID;
    STATUS_1394             Status1394;
    ULONG                   uDevicesFound, I;
    BOOLEAN                 bRestartLoop;

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );
    C1394AcknowledgeBusReset(C1394AdapterHandle);

    NodeID.BusID = LOCAL_1394_BUS_ID;

    do {
        bRestartLoop = FALSE;
        uDevicesFound = 0;

        *((PC1394_BUS_ID)&ULinkOnNodes) = LOCAL_1394_BUS_ID;
        C1394QueryInformation( C1394AdapterHandle,
                               OID_LINK_ON_NODES,
                               &ULinkOnNodes,
                               sizeof(ULinkOnNodes),
                               NULL,
                               NULL );

        for (I=0; I<63; I++)
            if ( (((ULONGLONG)1)<<I) & ULinkOnNodes )
                {
                    printf("-L-");
                    Sleep(500);

                    NodeID.PhysicalID = (C1394_PHYSICAL_ID) I;

                    Status1394 = C1394ReadNode( C1394AdapterHandle,
                                                NodeID,
                                                CSR_BUS_INFO_BLOCK+8,
                                                8,
                                                &DeviceGuidArray[uDevicesFound],
                                                NULL,
                                                NULL );

                    switch (Status1394)
                    {
                        //-----
                        case STATUS_1394_SUCCESS:
                            uDevicesFound++;
                            break;
                        //-----
                        case STATUS_1394_BUS_RESET:
                            printf("<BR>");
                            C1394AcknowledgeBusReset(C1394AdapterHandle);
                            bRestartLoop = TRUE;
                            break;
                    }

                    if (bRestartLoop)
                        break;
                }
    }
    while (bRestartLoop);
}
```

```
puts("");

for (I=0; I<uDevicesFound; I++)
{
    printf("Device found with GUID %08X-%08X\n",
        SwapEndian32(*(ULONG*) DeviceGuidArray[I].Bytes),
        SwapEndian32(*(ULONG*)&DeviceGuidArray[I].Bytes[4]) );
}

C1394CloseAdapter( C1394AdapterHandle );
C1394Terminate();
return 0;
}
```

No matter when a bus reset will occur, the code will identify it and restart the enumeration loop.

Non-blocking Calls

The call to **C1394ReadNode** is *blocking*, which means that the call returns only after the read transaction has completed in one way or the other. However this involves many steps (sending a request packet, checking the acknowledge, waiting for the response or timeout, etc), and an application might have other important tasks to perform in the meanwhile.

For this reason FireAPI includes a *non-blocking*¹² counterpart to each **C1394xxxNode** function, called **C1394xxxNodeAsynch**.

The operating model of all non-blocking transaction request calls is as follows:

1. Make the non-blocking call, passing among the parameters a handle to an event object, and a context value that means *something* to the application.
2. The call will return a handle to a *non-blocking asynchronous transaction*.
3. When the event is set make a call to **C1394CompleteAsynch** that will complete the operation and return context information to the caller.

The sample function below demonstrates the usage of **C1394ReadNodeAsynch**.

```
void ReadNodeGuidAsynch( C1394_ADAPTER_HANDLE  a_C1394AdapterHandle,
                        C1394_NODE_ID        a_NodeID )
{
    STATUS_1394          Status1394;
    C1394_GUID           NodeGuid;
    C1394_ASYNC_HANDLE  AsynchHandle;
    HANDLE               hEvent;
    void                 *Context;

    hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    AsynchHandle = C1394ReadNodeAsynch( a_C1394AdapterHandle,
                                       a_NodeID,
                                       CSR_BUS_INFO_BLOCK + 8,
                                       8,
                                       &NodeGuid,
                                       &Status1394,
                                       NULL,
                                       NULL,
                                       (void*)0xABCDEF12,
                                       hEvent );

    if (NULL == AsynchHandle)
    {
        printf( "The operation failed with status %s\n",
               C1394StatusString(Status1394) );
        return;
    }

    printf( "Non-blocking execution (status is %s).\n",
           C1394StatusString(Status1394) );

    do
    {
        // Do some other processing.
        printf("***");
    }
    while ( WAIT_TIMEOUT == WaitForSingleObject(hEvent, 0) );

    printf("\n");

    // Complete the operation.
    Context = C1394CompleteAsynch( AsynchHandle );

    // Context returned is the context passed to the call.
    printf( "INFO - Returned context is %X\n", Context );
}
```

¹² *Blocking* and *Non-Blocking* are also known as Synchronous and Asynchronous, but using these terms would certainly cause confusion with the 1394 meanings of Isochronous and Asynchronous.

```

// What happened with the operation?
switch (Status1394)
{
    //-----
    case STATUS_1394_SUCCESS:
        printf( "GUID of Node %u.%u: %08X-%08X\n",
            (ULONG) a_NodeID.BusID,
            (ULONG) a_NodeID.PhysicalID,
            SwapEndian32(*((ULONG*) NodeGuid.Bytes)),
            SwapEndian32(*((ULONG*) &NodeGuid.Bytes[4])) );
        break;

    //-----
    default:
        printf( "Block Read Request for Node %u.%u failed with status %s\n",
            (ULONG) a_NodeID.BusID,
            (ULONG) a_NodeID.PhysicalID,
            C1394StatusString(Status1394) );
}
}

```

In the call to **C1394ReadNodeAsynch** the caller can optionally specify a context value that will be returned by **C1394CompleteAsynch** when the operation is completed. This can be either a value of some sort or a pointer to a structure that contains information the caller maintains about the operation. If there is nothing that the client wants to use as context information, then it can simply specify NULL and ignore the return value of **C1394CompleteAsynch**.

Note, that if the do-while loop was replaced with this one:

```

do
{
    // Do some other processing.
    printf( "*" );
}
while ( Status1394 == STATUS_1394_PENDING );

```

then we would have an infinite loop.

*Status information about asynchronous command completion is only returned to user mode when **C1394CompleteAsynch** is called.*

Write Transaction Requests

Write transaction requests through functions **C1394WriteNode** and **C1394WriteNodeAsynch** are handled similarly to read requests. The sample below demonstrates how to send a broadcast write transaction. Broadcast writes do not receive a response packet, neither get acknowledged. A successful broadcast write is indicated by FireAPI with a return status of `STATUS_1394_SUCCESS`. In the case of broadcast writes FireAPI returns the artificial acknowledge code `ack_none`.

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE  C1394AdapterHandle;
    C1394_NODE_ID         NodeID;
    STATUS_1394           Status1394;
    C1394_RESPONSE_CODE  ResponseCode;
    C1394_ACK_CODE       AcknowledgeCode;
    UCHAR                WriteBuffer[4] = {0x15, 0xF3, 0x72, 0x99};

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

    NodeID.BusID = LOCAL_1394_BUS_ID;
    NodeID.PhysicalID = 63;

    Status1394 = C1394WriteNode( C1394AdapterHandle,
                                NodeID,
                                CSR_MAINT_UTILITY,
                                4,
                                &WriteBuffer,
                                &ResponseCode,
                                &AcknowledgeCode );

    switch (Status1394)
    {
        case STATUS_1394_SUCCESS:
            printf( "Broadcast write succeeded. Acknowledge is %s\n",
                   C1394AckCodeString(AcknowledgeCode) );
            break;

        default:
            printf( "C1394WriteNode failed with status %s\n",
                   C1394StatusString(Status1394) );
    }

    C1394CloseAdapter( C1394AdapterHandle );
    C1394Terminate();
    return 0;
}
```

Lock Transaction Requests

Lock transactions are a little different than reads or writes because there are 6 different types of *lock functions* that are implemented through a lock transaction. The transaction code is 'Lock', and the extended transaction code identifies the function to be used.

Lock functions can be performed on 32-bit or 64-bit values, and take one or two 32-bit or 64-bit arguments. These are called the *arg_value* and the *data_value*. In functions that take one argument, only *data_value* is used.

The value at the destination 1394 offset immediately prior to performing the lock function is always returned in the data of the lock response packet.

The list of lock functions, and their specification is shown in the table below.

Lock Function	Args	Update Action
mask_swap	2	$\text{new_value} = (\text{data_value} \& \text{arg_value}) \mid (\text{old_value} \& \sim \text{arg_value});$
compare_swap	2	$\text{if } (\text{old_value} == \text{arg_value})$ $\text{new_value} = \text{data_value};$
fetch_add	1	$\text{New_value} = \text{old_value} + \text{data_value};$
little_add	1	$(\text{little}) \text{ new_value} =$ $(\text{little}) \text{ old_value} + (\text{little}) \text{ data_value};$
bounded_add	2	$\text{if } (\text{old_value} != \text{arg_value})$ $\text{new_value} = \text{old_value} + \text{data_value};$
wrap_add	2	$\text{new_value} = (\text{old_value} != \text{arg_value}) ?$ $(\text{old_value} + \text{data_value}) : \text{data_value};$

Table 3. Lock Transaction Functions

Let us describe the operation of `compare_swap` which is the most commonly used lock function. As its name indicates this is a function used to perform atomic compare & swap operations.

The sender transmits a lock transaction with the offset of a 1394 register that it wants to update atomically. In this lock transaction packet, the sender includes the *arg_value*, which is the most recent value that the sender knows that the target register has, and the *data_value*, which is the new value that the sender wishes to store in the register.

When the target node receives the lock request it will prepare a lock response packet with the `resp_complete` response code that will contain the current value of the target register (referred to as *old_value*).

Then it will compare the *arg_value* in the lock request packet against the current value of the register, and if the values are equal, then the target node will store *data_value* as the new value of the register. If they are not equal, no more actions will be taken.

When the sender of the lock request receives the lock response packet, it will compare the value returned to him (*old_value*) against the *arg_value* that he sent in order to find out if the lock function failed.

A common source of confusion is what is meant when we say that “*a lock transaction failed*”. Most people take this to mean the same as the “*lock function failed*”, but we must be careful with this. The description of `compare_swap` that was given above is deliberately incomplete in order to be able to emphasize on this kind of error.

The first thing that the receiver of the lock response must check is whether the response code is `resp_complete`. If it is not then the *lock transaction* has failed. If it is `resp_complete`, then the lock transaction has succeeded but we don't yet know whether the *lock function* has succeeded. In order to check for this, the comparison between *old_value* and *data_value* must be made.

The lock functions, **C1394LockNode** and **C1394LockNodeAsynch**, return information about what happened with the lock transaction. The result of the *lock function* itself should be checked by the

caller. However, because compare-swap is a very common operation, FireAPI also includes for convenience **C1394CompareSwapNode**, a wrapper function around **C1394LockNode**, so that it is easier to perform this specific lock function.

The sample fragment code below demonstrates a possible implementation of **C1394CompareSwapNode**.

```

STATUS_1394 C1394CompareSwapNode(
    IN    C1394_ADAPTER_HANDLE  a_C1394AdapterHandle,
    IN    C1394_NODE_ID        a_NodeID,
    IN    C1394_OFFSET         a_Offset,
    IN    BOOLEAN               a_32BitLock,
    IN    ULONGLONG            a_UArgValue,
    IN    ULONGLONG            a_UDataValue
)
{
    STATUS_1394    Status1394;
    UCHAR          OldValue[8];

    Status1394 = C1394LockNode( a_C1394AdapterHandle,
                               a_NodeID,
                               a_Offset,
                               COMPARE_SWAP,
                               a_32BitLock ? 4 : 8,
                               a_UArgValue,
                               a_32BitLock ? 4 : 8,
                               a_UDataValue,
                               OldValue,
                               NULL,
                               NULL );

    // Did the lock-transaction fail in the first place?
    if (STATUS_1394_SUCCESS != Status1394)
        return Status1394;

    // Let us check whether the compare-swap succeeded as well.
    if (a_32BitLock)
    {
        if ( (*(ULONG*)OldValue) != (ULONG)a_UArgValue )
            return STATUS_1394_LOCK_FAILED;
    }
    else
    {
        if ( (*(ULONGLONG*)OldValue) != a_UArgValue )
            return STATUS_1394_LOCK_FAILED;
    }

    // The compare swap succeeded.
    return STATUS_1394_SUCCESS;
}

```

As it can be seen, depending on whether this is a 32-bit or 64-bit lock, in the comparison after the lock transaction completes, the *a_UArgValue* parameter is either casted to ULONG or simply treated as a ULONGLONG. Similarly depending on the size of the lock, both **C1394LockNode** and the check code treat the *OldValue* variable as a pointer to a ULONG or to a ULONGLONG.

It is important to remember several things for **C1394LockNode**, **C1394LockNodeAsynch** and **C1394CompareSwapNode**:

1. They always treat the target CSR as big endian.
2. They are the only functions that treat their input data in the native format of the host CPU, by internally doing little-to-big endian conversions as necessary.
3. A failed lock also acts like a read operation. You don't have to do a read after a failed lock, but simply update the *arg_value* and *data_value* parameters and retry the call.

Handle-Based Functions

The importance of device handles was discussed earlier in this document. Before ubCore 5.50, Device Handles were implemented as a separate DLL (UB1394DH.DLL) in ubCore and a separate import library (UB1394DH.LIB) in FireAPI.

As of ubCore/FireAPI 5.50 all the Device Handle functions have been moved into UB1394.DLL and the import library that contains them is UB1394.LIB.

UB1394DH.DLL is still part of ubCore for backwards compatibility reasons. This DLL is a *Forwarder DLL* that forwards all function calls to UB1394.DLL. The UB1394DH.LIB import library has been removed from FireAPI 5.50 or later.

An application will be able to use **C1394OpenDevice** to open a handle to the nodes that it wants to work with. Using this handle, the application can communicate with the node regardless of the current physical ID that the node is using.

The class driver maintains a mapping between a device handle and the NodeID. Whenever a bus reset occurs, the driver will “freeze” the handle (hold outgoing requests in a queue) and, once the bus reset is completed, take actions to reestablish the mappings so that the application can continue its operation using the same handles, completely unaware of the bus reset.

The class driver provides a **completely transparent handling of bus resets**. This includes the automatic retransmission of pending requests that were queued for transmission but were cancelled when the bus reset took place, and transaction requests that had been sent and acknowledged but a bus reset occurred before the response had arrived.

Simply put, when an application uses a handle to a device it will never get a transaction aborted due to a bus reset. This greatly simplifies the program code, because handling with such an error requires complex procedures and involves many subtle issues.

If the device has been removed from the bus, then an attempt to send a transaction request to the device returns **STATUS_1394_DEVICE_NOT_FOUND**. The device handle is not invalidated in this case. If the device is reconnected to the bus then the application will be able to continue working with it.

The class driver internally maintains a map between GUIDs and NodeIDs that it updates by snooping on read response packets that read the GUID portion of a device’s Configuration ROM. By maintaining a cache with this data, the class driver significantly reduces bus traffic after a bus reset. No matter how many applications run on the node, and how many device handles are open, each FireAPI node will only read the configuration ROM of a given node once.

Using Device Handles

The sample code below shows how to use device handles.

A couple of important items to mention prior to using device handle functionality:

- Each file that uses Device Handles must include the header file **UB1394DH.H**.
- You must add the directory where the above header file resides to your project's *include directories*.
- You must add the directory where **UB1394.LIB** file reside to your project's library directories (as appropriate for x86/x64 builds).
- You must add **UB1394.LIB** to the link libraries of your project (as appropriate for x86/x64 builds).

Here is a sample code using Device Handles to read the configuration rom of a node.

```
#include <stdio.h>
#include "FireAPI.h"
#include "UB1394DH.h"

int main()
{
    STATUS_1394           Status1394;
    C1394_ADAPTER_HANDLE AdapterHandle;
    C1394_NODE_ID        NodeID;
    C1394_GUID           DeviceGuid;
    DEVICE_HANDLE        DeviceHandle;

    Status1394 = C1394Initialize();
    if (Status1394 != STATUS_1394_SUCCESS)
    {
        printf("Error initializing 1394 Stack. Error code:%s.\n",
            C1394StatusString(Status1394));
        return -1;
    }

    Status1394 = C1394OpenAdapter( NULL,
        (CLIENT_ADAPTER_HANDLE) AdapterHandle,
        &AdapterHandle);

    if (Status1394 != STATUS_1394_SUCCESS)
    {
        printf("Error opening local adapter. Error code:%s.\n",
            C1394StatusString(Status1394));
        return -1;
    }

    // Get the GUID of node N=0 on the local bus.
    // We will assume it has LinkOn==1 so that it will respond to ROM reads.
    // Additionally we will assume it correctly implements the specs and can handle
    // an 8-byte block read in the Configuration ROM header area.
    NodeID.BusID = LOCAL_1394_BUS_ID;
    NodeID.PhysicalID = (C1394_PHYSICAL_ID)0;
    Status1394 = C1394ReadNode( AdapterHandle,
        NodeID,
        CSR_CONFIG_ROM+0xC,
        0x8,
        &DeviceGuid,
        NULL,
        NULL);

    if (Status1394 != STATUS_1394_SUCCESS)
    {
        printf("Error getting Device GUID. Error code:%s.\n",
            C1394StatusString(Status1394));
        return -1;
    }

    // Open a Device handle for that Guid
    Status1394 = C1394OpenDevice( AdapterHandle,
        &DeviceGuid,
        &DeviceHandle);
}
```

```

if (Status1394 != STATUS_1394_SUCCESS)
{
    printf("Error opening handle for the device. Error code:%s.\n",
        C1394StatusString(Status1394));
    return -1;
}

// Read using the newly created device handle.
Status1394 = C1394ReadDevice( DeviceHandle,
                             CSR_CONFIG_ROM,
                             24,
                             Buffer,
                             NULL,
                             NULL);

if (Status1394 != STATUS_1394_SUCCESS)
{
    printf("Error reading device. Error code:%s.\n",
        C1394StatusString(Status1394));
    return -1;
}

C1394CloseDevice(DeviceHandle);
C1394CloseAdapter(AdapterHandle);
C1394Terminate();
return 0;
}

```

- The code inside the UB1394.DLL tries 3 times to read the nodes' GUID after a bus reset. The reason for this is the following: When a bus reset occurs, all devices have to perform some kind of post-bus-reset processing. This usually takes different amounts of time on each device, depending on how fast it is, how long did it took to raise the bus reset interrupt, etc. This in turn means that it might be possible that the application's node has completed its bus reset processing and started executing the relocation code before one or more other nodes on the bus. This will result in the read transaction request being sent before the target node being ready to process it. As a result the transaction might be acknowledged by the adapter hardware with `ack_pending` but not get actually processed so response is ever sent and the request times out. For this reason, it is in general suggested that either 1394 applications and drivers wait 10-20 msec after a bus reset before they start sending any transactions or they are prepared to handle timeouts on the first transaction(s) they send to a node¹³.
- Invalid bus topology.** This is a very important issue. You might get surprised to find out that there are occasions when the hardware does not work according to the standards, even in basic things like sending the SelfID packets when a bus reset occurs. Two things can possibly go wrong when plugging or unplugging a device from the 1394 bus¹⁴:

 - One or more adapters did not send their SelfID packets.
 - A *nested bus reset* occurred and this causes the adapters to think that they have received no SelfID packets at all.

In both cases the 1394 class driver (**UB1394.SYS**) does not have valid information about the bus topology. Usually these conditions get corrected if an additional, software generated bus reset is initiated. Unibrain's Serial Bus Manager driver (**UBSBM.SYS**) checks for these conditions and automatically initiates a bus reset to correct them.

¹³ This kind of problem is more likely to occur in kernel mode drivers that use FireAPI, because they get a notification callback called immediately when bus reset is complete. User mode *Bus Reset Complete* notifications delay a little more because (a) all the kernel mode *bus reset complete* processing code has to execute (b) it involves signalling an event object and then waiting for the thread to get scheduled. However a user-mode thread on a tight-loop calling **C1394IsBusResetInProgress** can get the chance to send a transaction very soon after the bus reset completes, especially if the application runs on a multiprocessor system.

¹⁴ These conditions have been verified with a 1394 bus analyzer.

1394 developers should keep this in mind, and be prepared to handle such conditions in their code if they expect nodes to be added/removed to/from their 1394 bus while their system code is running.

Usually the best way to handle this condition is to delay 2 seconds, until the SBM initiates the software bus reset that should correct the situation.

- The code retrieves information from the class driver so that it identifies the link on nodes and only attempts to locate the device among these nodes. Two things should be kept in mind:
 - (a) It can occur that a device that is *'off'* has its LinkOn bit set to 1. This is attributed to either an error or a limitation in the hardware design or a bug in the software¹⁵. Such nodes will not acknowledge any transaction request sent to them, so the code must be prepared to receive `STATUS_1394_NOT_FOUND` as the result of a transaction request.
 - (b) For the same reasons a device might be active and have its LinkOn bit set to zero. If you encounter such a device then either modify the device's firmware or the sample code.
- Unless someone has sent a *forceroot* configuration packet, a software initiated bus reset most probably will not change the topology of the bus. The code takes this fact into consideration and when relocating the node it first attempts to read from the previous NodeID.

Simplifications\ShortComings in this implementation:

- Not multithreaded-safe. If you want to be able to use a single handle from multiple threads then you will have to use a synchronization object to prevent more than one threads from calling the relocation code at the same time.
- In each call to **C1394ReadDevice** the code tries to relocate the device. If the device is removed from the bus then there is no point in trying to relocate it unless a new bus reset occurs. However keep in mind that a device might *logically disappear* from the bus because it is rebooting, and it is unresponsive during the boot process. Such a device should initiate a bus reset when its boot process completes¹⁶. If the device does not do a bus reset on power up, then an implementation that does not attempt to relocate the device in each **C1394ReadDevice** call, will not see that the device is 'up' again until a bus reset occurs.
- The code uses a serialized logic when it tries to relocate a device. This has two drawbacks:
 - (a) If one or more timeouts occur (even on unrelated nodes), then the total delay time of the relocation procedure will be big. If the code issued all its transactions as non-blocking calls then it would possibly locate its device much faster in the case that timeouts occur.
 - (b) Many devices¹⁷ can only accept 1 transaction request at a time. If a second request arrives before they have sent the response to the first then they don't respond to it, and a timeout will occur.

If many applications on many nodes try all at the same time (after the bus reset) to relocate their devices and the 1394 bus contains one or more nodes that can only process up to 1 transaction at a time, then many timeouts will occur.
- It is suggested that applications try to optimize their logic¹⁸ and minimize the number of transactions they attempt for the following reasons:
 - (a) Reduced traffic on the bus.
 - (b) Less transaction requests mean less points of failure in the code.
 - (c) Less traffic on the bus means easier to analyze traffic capture logs.

¹⁵ When a device shuts down its software should clear the linkOn bit. If not, then on some chips this can stay one even after the device is shut down.

¹⁶ It is also suggested that it initiates a bus reset in the very last steps of its shutdown procedure, as the last step of shutting down its 1394 subsystem.

¹⁷ For example 1394 digital cameras.

¹⁸ The sample code optimizes the relocation procedure by only looking at nodes that claim to be *alive* (LinkOn bit set).

Retrying transactions with *C1394Retry* functions

A new set of functions were added to ubCore 5.50 that are related to performing reliably asynchronous transactions with devices on the 1394 bus.

To start with, asynchronous transactions are by definition “reliable” in the traditional meaning of the term when talking about communication protocols. There is an *acknowledge code* that informs the sender about the outcome of the *physical* transmission of the transaction request packet and then there is a *response code* that informs the sender about the *logical* outcome of the operation.

This way the sender is immediately aware of any failures and may retry a failed operation as appropriate, in essence building its own reliability layer on top of the reliability primitives provided by the Firewire protocol.

Most 1394 devices are built by engineering teams with limited resources and pressing schedules and thus do not always operate as expected. They may be fully within the 1394 specifications, but still may appear to occasionally have some kind of capricious behavior. Simply put, in practice not all 1394 devices are always well-behaved.

Unibrain has dealt with a wide series of devices and has repeatedly written code that retries transactions of one form or another. Unibrain’s experience has been encoded into a set of functions that are readily available for use to all applications. These functions will be continuously improved and fine-tuned as Unibrain’s experience grows further and as future additions are made to the Class Driver.

The *C1394Retry* functions contain two fundamental pieces of logic:

1. Precision timing of the device’s responses in order to automatically adjust the rate at which transaction requests are sent to the device.
2. An algorithm that based on the transaction outcome and current retry status decides if the transaction can and should be retried or not.

We have seen that a very reliable indicator about a 1394 device’s ability to accept a new transaction request is how long it took to respond to the previous transaction (which is usually to the same initiator).

The amount of time between sending the transaction request and receiving the transaction response is in the order of 30-500 microsec. Delaying the next outgoing transaction request for that device by the same amount of time yields a very smooth exchange between the PC and the device, with no timeouts and no busy acknowledges or other conflicts.

The Windows operating system provides the *High Resolution Performance Timer* for precisely measuring small amounts of elapsed time (of the order described above). Although the measurements may be significantly affected by outside events (a context switch, an interrupt, etc), on the average they can be quite reliable.

The major problem however is not measuring these small intervals, but introducing such small delays in code execution before sending off the next transaction request, without causing excessive delays in program execution or using excessively the CPU.

All the above have been embedded in the following functions:

Function	Description
<i>C1394MayRetryTransaction</i>	Encodes the logic of whether or not it makes sense to retry a failed transaction to a device.
<i>C1394RetryReadNodeInQuads</i>	Reads a block of memory using quadlet reads from a device given its NodeID. Retries are performed on failed transactions as needed.
<i>C1394RetryReadNodeExInQuads</i>	<i>Ex</i> variant of the previous function.
<i>C1394RetryReadDeviceInQuads</i>	Reads a block of memory using quadlet reads from a device given its Device Handle. Retries are performed on failed transactions as needed.
<i>C1394RetryWriteDeviceInQuads</i>	Writes a block of memory using quadlet reads to a device given its Device Handle. Retries are performed on failed transactions as needed.

Performing Asynchronous Streaming transactions

Asynchronous streaming transactions in FireAPI are performed through the **C1394TransmitPackets** function call. Asynchronous streaming packets have the same format and characteristics as the isochronous packets. Asynchronous stream packets are transmitted at a user specified channel number just like isochronous packets are. As opposed to normal asynchronous packets there is no acknowledge or response code returned by the target node for asynchronous stream packets.

In order to perform an asynchronous streaming operation the user should prepare a **FIREAPI_TRANSACTION** structure or an array of them and specify a transaction code of **TCODE_STREAM_DATA**. The user should then fill the desired channel and sycode in this structure as well as some other parameters and pass it to a call to **C1394TransmitPackets**.

For more information about the structure and format of asynchronous streaming packets the user should consult the IEEE 1394 specification.

The sample source code below demonstrates how to perform an asynchronous stream transmission. The transmission is performed at channel 1.

```
#define MAX_TRANSACTIONS      32

int main(int argc, char **argv)
{
    C1394_GUID                AdapterGUID;
    STATUS_1394               Status1394;
    ULONG                     uAdapters;
    C1394_ADAPTER_HANDLE      C1394AdapterHandle;
    PFIREAPI_TRANSACTION      transArray[MAX_TRANSACTIONS];
    HANDLE                    hEvent;
    C1394_ASYNC_HANDLE        asynchHandle;
    PFIREAPI_TRANSACTION      transaction;
    UINT                      i, j;
    ULONG                     fillValue;
    UINT                      transactionsNum = 5;
    UINT                      bufferSize     = 0x200;
    C1394_CHANNEL              channel       = 1;
    C1394_TAG                  tag           = 0;
    C1394_SY_CODE              syCode       = 0;

    // Initialize with 1394.
    if (STATUS_1394_SUCCESS != C1394Initialize())
    {
        puts("C1394Initialize Failed.");
        return -1;
    }

    // Get the number of adapters.
    uAdapters = C1394GetAdapters( &AdapterGUID, 1 );

    // Try to open the adapter.
    Status1394 = C1394OpenAdapter( &AdapterGUID,
        (CLIENT_ADAPTER_HANDLE) &C1394AdapterHandle,
        &C1394AdapterHandle
    );

    if (STATUS_1394_SUCCESS != Status1394)
    {
        printf("FAILED to open adapter. 1394 Status Code is %X\n",
            Status1394
        );
        return -2;
    }

    // Initialize the array that holds the transactions
    ZeroMemory(transArray, MAX_TRANSACTIONS *
        sizeof(PFIREAPI_TRANSACTION)
    );

    // Initialize the value that will be used to fill the packets
    fillValue = bufferSize / sizeof(QUADLET);
}
```

```

// Prepare the transactions that will be transmitted
for (i = 0; i < transactionsNum; i++)
{
    transaction = (PFIREAPI_TRANSACTION)HeapAlloc(GetProcessHeap(),
        HEAP_ZERO_MEMORY, sizeof(FIREAPI_TRANSACTION));

    if (transaction == NULL)
    {
        return -5;
    }

    transArray[i] = transaction;
    transaction->PacketHeader.uHeaderBytes = 0;
    transaction->PacketHeader.data_length = bufferSize;
    transaction->PacketHeader.TransactionCode = TCODE_STREAM_DATA;
    transaction->PacketHeader.Channel = channel;
    transaction->PacketHeader.Tag = tag;
    transaction->PacketHeader.SyCode = syCode;
    transaction->uBusResetCount =
        C1394GetBusResetCount(C1394AdapterHandle);
    transaction->TransmissionSpeed =
        C1394GetAdapterSpeed(C1394AdapterHandle);
    transaction->Status1394 = STATUS_1394_PENDING;
    transaction->Buffer.pBytes =
        HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, bufferSize);

    if (NULL == transaction->Buffer.pBytes)
    {
        printf("Memory allocation FAILED\n");
        return -3;
    }

    // Fill the memory with a specific pattern, so receiver can
    // check data integrity
    for (j = 0; j < bufferSize / sizeof(ULONG); j++)
    {
        *((PULONG)transaction->Buffer.pBytes + j) = fillValue++;
    }
}

// Create the event that will indicate that transmission is complete
hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (hEvent == NULL)
{
    return -4;
}

// Transmit the packets.
asynchHandle = C1394TransmitPackets(C1394AdapterHandle,
    &transArray[0], transactionsNum, &Status1394, NULL, hEvent
);

if (asynchHandle != NULL)
{
    if (Status1394 == STATUS_1394_PENDING)
    {
        if (WaitForSingleObject(hEvent, 5000) == WAIT_OBJECT_0)
        {
            printf("Wait success...\n");
        }
        else
        {
            printf("Wait failure.\n");
        }
        C1394CompleteAsynch(asynchHandle);
    }

    // Check the status of each transaction
    for (i = 0; i < transactionsNum; i++)
    {
        Status1394 = transArray[i]->Status1394;
        printf("C1394TransmitPackets for transaction %d returned %s\n",

```

```
        i, C1394StatusString(Status1394)
    );
}
}
else
{
    printf("C1394TransmitPackets returned %s\n", C1394StatusString(Status1394));
}

// Free allocated memory
for (i = 0; i < transactionsNum; i++)
{
    if (transArray[i]->Buffer.pBytes != NULL)
    {
        HeapFree(GetProcessHeap(), 0, transArray[i]->Buffer.pBytes);
    }
    if (transArray[i] != NULL)
    {
        HeapFree(GetProcessHeap(), 0, transArray[i]);
    }
}

if (hEvent != INVALID_HANDLE_VALUE)
{
    CloseHandle(hEvent);
}

// Officially close the adapter.
C1394CloseAdapter(C1394AdapterHandle);

// Cleanup 1394 support.
C1394Terminate();
return 0;
}
```

Accepting Transactions from Remote Nodes

This involves making available address ranges in the 1394 address space of the local host (more precisely on the 1394 address space of an adapter on the local host), so that remote nodes can perform IEEE 1394 asynchronous read, write or lock transactions to these ranges.

The 1394 Address Space

According to this paragraph there are the following ranges:

- **Low Address Space: 0 to `physicalUpperBound`.** All write requests are immediately acknowledged with `ack_complete` even if the write operation has not been yet completed.
- **Middle Address Space: `physicalUpperBound` to `FFFE_FFFF_FFFF`.** All transaction requests are handled by software, but write requests with non-zero extended transaction code are automatically acknowledged by the adapter hardware with `ack_complete`.
- **Upper Address Space: `FFFF_0000_0000` to `FFFF_EFFF_FFFF`.** All transaction requests are handled by software and an `ack_pending` acknowledge is automatically returned by the adapter.
- **CSR Space: `FFFF_F000_0000` to `FFFF_FFFF_FFFF`.** All transaction requests are forwarded to software for processing with the exceptions of certain CSRs that are handled by the adapter. An `ack_pending` acknowledge is automatically returned by the adapter for all requests.

The mechanism of the `physicalUpperBound` register is too limited to fully accommodate a secure operating system like Windows NT. Providing NT host memory physical access to remote nodes could only possibly occur to restricted ranges of physical pages, FireAPI does not currently implement direct physical access.

This means that in the adapters where `physicalUpperBound` is configurable, it is set to zero, and in those adapters that have a hardwired value, the feature is disabled and the class driver will not permit a CSR below that value.

The applications should follow the following guidelines when using the node's address ranges:

1. Read or write requests within the range 0 to `FFFE_FFFF_FFFF` (*low and middle address spaces*) shall **not** have 1394 visible side-effects.

The term *visible side-effect* is used to denote an indirect action caused by a request or response which results in the alteration of the contents or usage of host memory outside the address scope of the request or response.

What the above means is that an application should only use the *CSR Space* when it wants to implement Control and Status registers, i.e. registers whose manipulation through incoming 1394 transactions may alter the contents of other addresses in the 1394 address space.

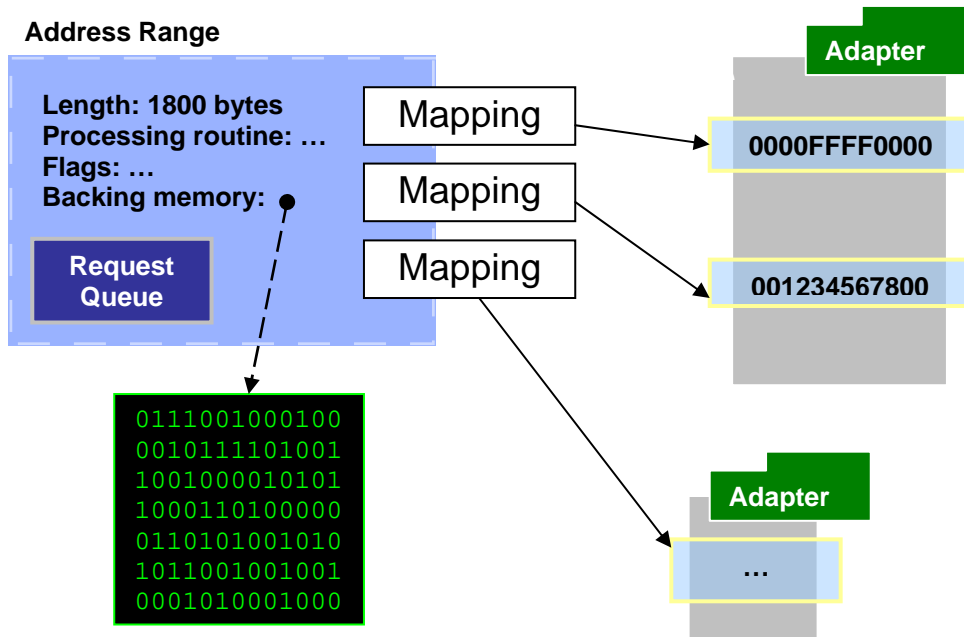
Address Ranges in the 1394 Address Space

The fundamental object required in order to accept incoming transactions is the *Address Range*.

The *Address Range* is an object that has a length, a request queue, optionally some associated memory and a set of *Mappings*. An *Address Range Mapping* associates an *Address Range* with a specific base-offset in the 1394 address space of one of the locally installed adapters.

This means that a single *Address Range*, can possibly have many mappings on the same adapter at different offsets, or to more than one adapters.

The figure below illustrates the concept of *Address Ranges & Mappings*, displaying an address range of size 1800 bytes, that is mapped at two different offsets on the first adapter, and on another offset on the second adapter¹⁹.



Each mapping is associated with a set of access-rights, which define which transaction codes will be accepted through this mapping. The class driver automatically responds with `resp_type_error` to transaction requests that specify a non-permitted transaction code on a mapping. This way the application developer need only write code to handle the transaction requests that are of interest to the application.

Similarly the application can specify which transaction requests are *automatically serviced* by the class driver without bothering the application. For example an application could specify that it wants the class driver to automatically respond to read requests by returning the data found in the memory associated with the address range, and only pass to the application write and lock requests.

It is of primary importance to note that no matter how many *mappings* an *Address Range* has, incoming transaction requests are queued by the class driver in a single request queue and a single Win32 event object is used by the applications for processing.

All incoming transaction requests²⁰ are stored in the address range's request queue and are kept there until the application proceeds to process them. The mechanism of operation provided by FireAPI makes it much easier for applications to process incoming transaction requests because there is no

¹⁹ We are always talking about adapter installed in the local host.

²⁰ The ones that are actually passed the access right checks.

limitation as to how soon should the application process the packets, or what will happen if more than one packets arrive before the application has the change to process the first packet.

A request always contains information about the mapping it was received from. Transaction responses are automatically sent back through the adapter on which the mapping is allocated.

Allocating and Freeing an Address Range

A client can allocate a new memory range and at the same time map it in the adapter's 48-bit 1394 address space by calling **C1394MapAddressRange**. The same function is used for re-mapping an existing address range to another 1394 adapter, or to another offset on the same adapter.

An address range mapping can be removed with **C1394UnmapAddressRange**.

The sample below demonstrates how to create an address range of size 8192 bytes, that accepts read transactions (block and quadlet) and quadlet write transactions, and maps it at a fixed offset value. The application specifies that the class driver should automatically service any read requests and only pass to the application any quadlet write requests.

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADDRESS_RANGE_CHARACTERISTICS arc;
    C1394_ADAPTER_HANDLE      C1394AdapterHandle;
    C1394_RANGE_HANDLE        C1394RangeHandle;
    HANDLE                    hStartProcessingEvent;
    STATUS_1394               Status1394;
    void *pMemory;

    // Initialize everything to NULL so that clean up is safe.
    C1394AdapterHandle = NULL;
    C1394RangeHandle   = NULL;
    hStartProcessingEvent = NULL;
    pMemory = NULL;

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    Status1394 = C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

    if (STATUS_1394_SUCCESS != Status1394)
        goto Cleanup;

    pMemory = malloc( 8192 );

    if (NULL == pMemory)
        goto Cleanup;

    // Clear the structure.
    ZeroMemory(&arc, sizeof(arc));

    arc.BaseAddress          = 0x800013941EEE;
    arc.uLength              = 8192;
    arc.pAddressRangeMemory  = pMemory;
    arc.fAccessRights        = ACCESS_READ_REQUESTS | ACCESS_QUADLET_WRITE;
    arc.fClientTransactions  = CLIENT_QUADLET_WRITE;
    arc.uMaxRequestQueueItems = 100;
    arc.ClientMappingHandle  = NULL;
    C1394RangeHandle         = NULL;

    Status1394 = C1394MapAddressRange( C1394AdapterHandle,
                                       &C1394RangeHandle,
                                       &hStartProcessingEvent,
                                       &arc );
}
```

```
// Did it fail ?
if (STATUS_1394_SUCCESS != Status1394)
{
    KdPrint(( "C1394MapAddressRange FAILED with status %s\n",
             C1394StatusString(Status1394) ));
    return -3;
}

Cleanup:;
if (NULL != C1394RangeHandle)
    C1394UnmapAddressRange( C1394AdapterHandle, C1394RangeHandle, 0x800013941EEE);

if (NULL != pMemory)
    free(pMemory);

if (NULL != C1394AdapterHandle)
    C1394CloseAdapter( C1394AdapterHandle );

C1394Terminate();
return 0;
}
```

Incoming Transaction Request Processing

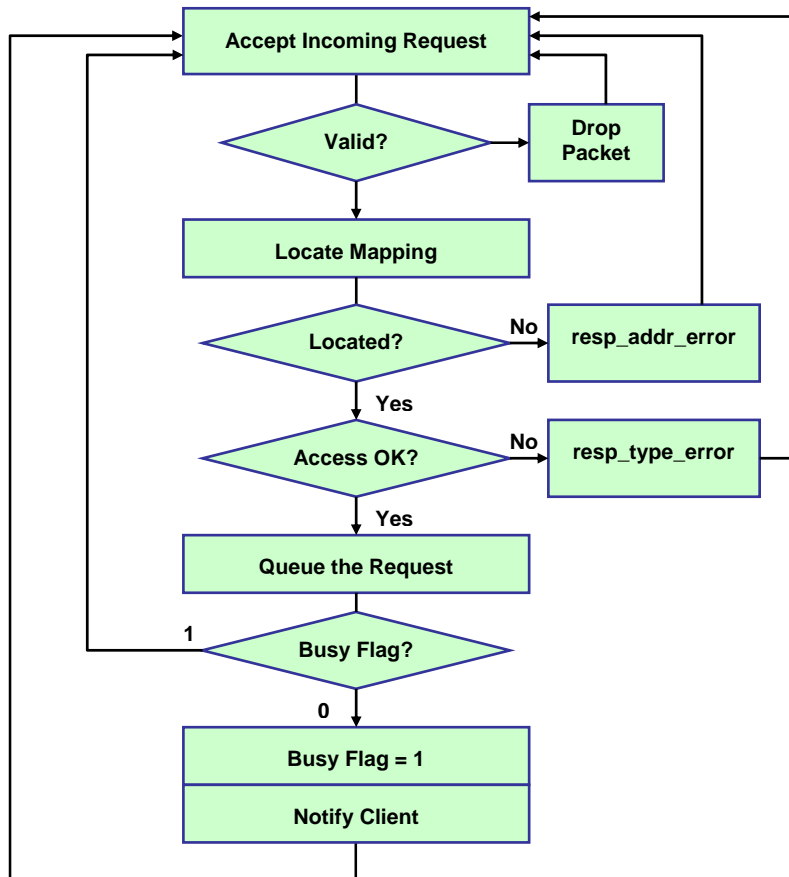
The underlying support mechanism for processing incoming transactions provides an auto-reset Win32 event object that is associated with an address range. Whenever a 1394 transaction request arrives for an address range mapping and the address range queue contains no requests, then the event object is set by the class driver.

The application should wait on this event and when it finds it signalled (set), it should call **C1394GetNextRequest** repeatedly, until a NULL pointer is returned. This is the indication that the address range's request queue is empty. Then the application can wait again on the event object until a new request arrives for this address range.

See the description of **C1394GetNextRequest** for more information on this method.

The processing of asynchronous transactions can be conceptually split in two parts: The part performed by the class driver, and the part performed by the application.

The class driver's high-level logic for the processing of incoming asynchronous requests is depicted in the following flowchart.



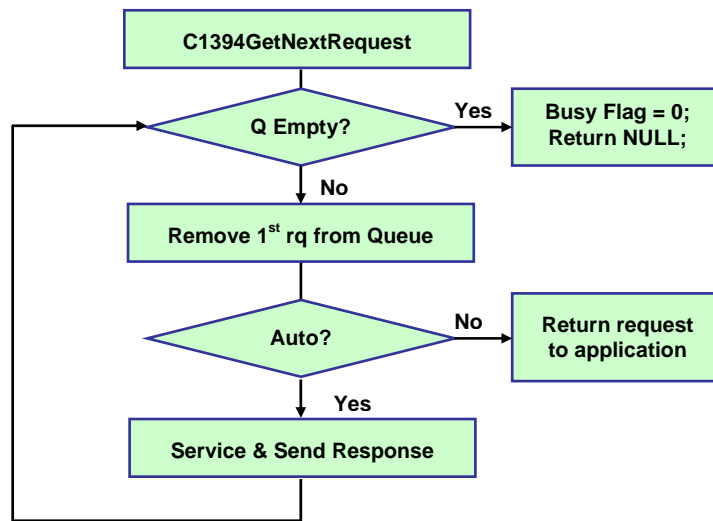
Incoming Transaction Request Processing Logic

From this flowchart we can see that the class driver does not check whether the request queue is empty, but directly queues the request and then checks the address range's *Busy Flag*. If the *Busy Flag* is set then the class driver does not notify the application. If the *Busy Flag* is clear, then the class driver sets it to 1 and notifies the application.

The *Busy Flag* is cleared in two occasions:

- When the address range is first created.
- When **C1394GetNextRequest** returns NULL.

The internal logic of **C1394GetNextRequest** is depicted in the following flowchart.



C1394GetNextRequest Logic

The operating logic of **C1394GetNextRequest** guarantees several things:

- A request is returned to the application only when the application requests it.
- Some transaction types that the client requested get automatically serviced by the class driver without being returned to the application. The response packet is also automatically generated and sent.
- Requests that get automatically serviced get serviced from within **C1394GetNextRequest**. This means that an auto-request will not get serviced at the moment it arrives, but it will be put in the request queue and will be processed when its *turn* has arrived and the application is ready. This way the application knows that NO transaction requests will be processed out of order, or while the application is updating the memory that is associated with the address range.

This model of operation allow for maximum flexibility on the application side, since an application is free to do the transaction processing in any way it finds suitable, either synchronous or asynchronous²¹ processing.

Whenever the application is ready to process a packet it makes a call to **C1394GetNextRequest**. The application is not forced to respond to a request before calling **C1394GetNextRequest** again to retrieve the next request from the queue.

For example an application can retrieve 2 requests from the queue, and respond to the 2nd before it responds to the 1st.

It is also very important to note that a single *request queue* is maintained for an address range, no matter how many mappings it has. This means that if for example an application has a register mapped in two adapters, the ordering of transactions is automatically provided by FireAPI. If two different request queues were used, and at one moment the application found 10 requests in each queue, then the application could not infer the overall order in which the transactions were received but only the relative order for each queue.

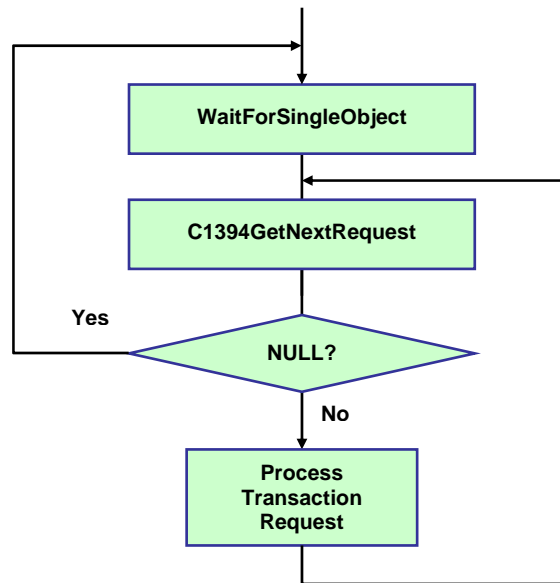
Additionally the application uses a single event object in order to get notified for either mapping, and a single address range handle to retrieve the requests from either mapping. This makes the coding of the application itself much simpler.

The client can use the translated packet header inside the packet structure to find out the exact information regarding each transaction request (transaction code, offset, data size etc), and optionally use the API provided functions to create the transaction responses. The client is completely free to respond to the requests in any order that suits its needs, or even not respond at all.

²¹ That is *immediate* or *deferred*.

Application Control Flow

The flowchart below illustrates the most common way for an application to process the incoming transaction requests for one of its address ranges.



Application Transaction Request Processing Logic

Simple CSR Server Sample

The following sample demonstrates the implementation of a register that accepts write and read transactions. Read transactions are automatically serviced by the class driver and write transactions are retrieved and serviced by the application. The application simply prints out some information about each write request and then services it by using function **C1394ServiceTransactionRequest**. When the application receives a zero length block write transaction at offset 0x123456780050 then it will exit its infinite processing loop.

```

#include <windows.h>
#include <stdio.h>
#include <ctype.h>
#include "FireAPI.h"

// The address of our register.
#define SIMPLE_CSR_OFFSET 0x123456780000
#define SIMPLE_CSR_SIZE 2048
#define SIMPLE_CSR_EXIT_OFFSET 0x50

UCHAR g_szMemory[SIMPLE_CSR_SIZE] =
    "FireAPI Simple CSR Server. Copyright Unibrain S.A. 1998-1999";

// A macro for producing printable characters out of a char value.
#define PRINTCHAR(a) (isprint(a) ? (a) : '.')

//*****
main(void)
{
    STATUS_1394          Status1394;
    HANDLE               hStartProcessingEvent;
    C1394_ADAPTER_HANDLE C1394AdapterHandle;
    C1394_RANGE_HANDLE  C1394RangeHandle;
    PC1394_PACKET       pPacket;

    C1394_ADDRESS_RANGE_CHARACTERISTICS arc;
  
```

```

// Initialize with 1394.
if (STATUS_1394_SUCCESS != C1394Initialize())
    return -1;

// Open the default adapter.
Status1394 = C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

if (STATUS_1394_SUCCESS != Status1394)
    return -2;

////////////////////////////////////
// Map our register.
////////////////////////////////////
arc.BaseAddress          = SIMPLE_CSR_OFFSET;
arc.uLength              = SIMPLE_CSR_SIZE;
arc.pAddressRangeMemory = g_szMemory;
arc.fAccessRights        = ACCESS_READ_REQUESTS | ACCESS_WRITE_REQUESTS;
arc.fClientTransactions = CLIENT_WRITE_REQUESTS;
arc.uMaxRequestQueueItems = 100;

// We don't need separate context information for each mapping,
// because we are only going to have one on each adapter.
arc.ClientMappingHandle = NULL;

// Set this to NULL so that it actually allocates an address range.
C1394RangeHandle = NULL;

Status1394 = C1394MapAddressRange( C1394AdapterHandle,
                                   &C1394RangeHandle,
                                   &hStartProcessingEvent,
                                   &arc );

// Did it fail ?
if (STATUS_1394_SUCCESS != Status1394)
    return -3;

// Service requests until a write is received.
for (;;)
{
    // Wait for the event to get signalled.
    WaitForSingleObject(hStartProcessingEvent, INFINITE);

    // Call C1394GetNextRequest until NULL is returned.
    for (;;)
    {
        pPacket = C1394GetNextRequest(C1394RangeHandle);

        // Did we empty the request queue ?
        if (NULL == pPacket)
            break;

        // Display some information about the incoming request.
        printf("-----\n");
        printf("TCode      : %s\n",
               C1394TCodeString(pPacket->PacketHeader.TransactionCode));
        printf("Offset      : %#I64X\n", pPacket->PacketHeader.Offset);
        printf("Data Length : %u\n", pPacket->PacketHeader.data_length);

        // Is it the exit-write ?
        if ((0 == pPacket->PacketHeader.data_length) &&
            ((SIMPLE_CSR_OFFSET + SIMPLE_CSR_EXIT_OFFSET) ==
             pPacket->PacketHeader.Offset))
        {
            // Just send a response packet. This might not really be
            // necessary as the write request might have already been acknowledged
            // with ack_complete. However this will be checked by
            // C1394SendResponse, which will also complete the packet.
            C1394SendResponse(pPacket, RESP_COMPLETE, NULL, 0);
            goto FinishOperation;
        }

        // Service the request using the standard routine.
        // This will send the response to the adapter that originated the
        // request packet, and will also complete the packet.
        C1394ServiceTransactionRequest(pPacket);
        pPacket = NULL;
    } // inner for (;;)
}

```

```

    } // outer for (;;)

FinishOperation:;
    C1394UnmapAddressRange( C1394AdapterHandle, C1394RangeHandle, SIMPLE_CSR_OFFSET );
    C1394CloseAdapter(C1394AdapterHandle);
    C1394Terminate();
    return 0;
}

```

Incoming request processing is implemented as two nested loops. In the outer loop the application waits for the event object to be signalled and once it gets signalled the application enters the inner loop where it retrieves transaction requests until the **C1394GetNextRequest** returns NULL.

It is very important to note that the code should not loop until the request queue is emptied but until **C1394GetNextRequest** returns NULL. For example, if an application knows that it expects 5 packets, then after the event gets signalled it should call **C1394GetNextRequest** 6 times. If it only calls it 5 times, then the *Busy Flag* of the address range will not get cleared and the event object will not get signalled again when a new request arrives.

Mapping an Address Range to more than one adapters

The DATETIME sample included with FireAPI demonstrates how to map an address range at the same offset on more than one adapters, and how to process the incoming requests from these mappings. This sample also registers two notifications and uses the Win32 function **WaitForMultipleObjects** so that it can process all events using a single thread.

Performance Optimization for Incoming Requests

The 1394 stack provides a performance optimization that can seriously improve the performance of applications that accept many asynchronous transaction requests. This section provides some extra information on this issue so that application developers can fine-tune their applications and achieve optimum performance.

The first important overhead of receiving an incoming transaction request is the fact that the packet has to be transferred from kernel mode to user mode. For this to happen a call must be made to the kernel mode portion of the 1394 stack. Each such call involves a user-mode to kernel-mode transition which is an operation that involves a certain amount of overhead.

The 1394 stack reduces this overhead by transferring more than one packets in each call, provided of course there are more than one packets available.

When an application calls **C1394GetNextRequest** the 1394 stack may transfer more than one incoming transaction request packets to user mode. The next calls to **C1394GetNextRequest** will not switch to kernel mode but will retrieve the packet from an internal queue maintained by **UB1394.DLL** in user mode.

At some point this queue will be emptied and the next call to **C1394GetNextRequest** will switch to kernel mode to check if there are any other packets available for transfer to user mode.

This way the number of user-to-kernel transitions is greatly reduced for address ranges that accept a heavy load of packets. For address ranges that represent CSRs (control & status register) this makes no difference. Usually such registers accept a small amount of requests per second so in most cases the address range's request queue only contains one packet.

The 1394 stack internally associates with each address range a *maximum packet transfer count*. This counter indicates the maximum packets that may be transferred from kernel mode to user mode in a single user-to-kernel transition.

The 1394 stack maintains for each application a global default value for this counter, the *default maximum packet transfer count*, and uses this value in order to initialize the *maximum packet transfer count* of newly created address ranges. The initial value that **UB1394.DLL** assigns to the *default maximum packet transfer count* is 5, but applications can use **C1394SetInformation** with **OID_DEF_AR_PACKET_TRANSFER** in order to modify this value.

From the time an address range is created, the 1394 stack assigns to it its own *maximum packet transfer count*. This is initialized from the value that the *default maximum packet transfer count* had when the address range was created. Applications can modify this value for a specific address range by using **C1394SetInformation** with the **OID_AR_PACKET_TRANSFER** identifier.

The maximum value that the *maximum packet transfer count* can take is 250.

The *maximum packet transfer count* of an address range does not only control the maximum number of packets that may be transferred in one user-to-kernel transition but also another item that can be very critical to performance.

As stated earlier when packets are transferred from kernel to user mode, they are cached by **UB1394.DLL** in a user-mode packet queue. **UB1394.DLL** has to dynamically allocate **C1394_PACKET** structures to populate this queue. These structures are freed when **C1394CompletePacket** is called for each packet.

However it is well known that dynamic memory allocations are expensive operations, which can present serious overhead to an application.

For this exact reason **UB1394.DLL** implements an additional performance optimization. Along with the queue of **C1394_PACKET** structure, **UB1394.DLL** maintains for each address range a *lookaside list* with **C1394_PACKET** structures.

A *lookaside list* is a list of available memory blocks of known size that can be used to greatly optimize the performance of memory allocations. When **C1394GetNextRequest** allocates a **C1394_PACKET** structure it first checks this lookaside list. If a block is available, then it retrieves it from this list with minimal overhead. If a block is not available then a *normal* memory allocation is performed.

One of the parameters of a lookaside list is its *maximum free list depth*, which is the maximum number of items that it will hold in its list of available memory blocks. When a block is to be freed **UB1394.DLL** checks the free list depth. If it is equal to the maximum value, then the memory block is normally freed, otherwise it is chained back to the free list and the depth of the free list is increased by one.

The consequence of this mechanism is that an application that does not force the lookaside list to empty completely will not be performing *normal* memory allocations, which have a lot of overhead, during its execution. If the free list depth is exhausted then necessarily some *normal* memory allocations will take place.

In the case of **UB1394.DLL**, the *maximum free list depth* of each address range is equal to its *maximum packet transfer count*. An application will exhaust this free list if it calls **C1394GetNextRequest** too many times without calling **C1394CompletePacket** in between. You can picture this as follows:

The *maximum free list depth* is M . The variable *Depth* is increased by one with each call to **C1394GetNextRequest** and decreased by one with each call to **C1394CompletePacket**. If $Depth > M$ then the application is not utilizing the lookaside list correctly, and should readjust the *maximum packet transfer count* of the address range.

Requests Spanning Address Ranges

The class driver will not permit an incoming request to span more than one mappings. If such a request is received then the class driver will respond to it by sending a response packet with the `resp_address_error` response code. This should be taken into consideration when designing an application.

For example an application that has 10 quadlet registers that represent statistics counters should not implement them as 10 address ranges of 4 bytes each, because then it would not be possible for a remote node to read all of them with a single 40-byte block read request.

If on the other hand the application has 10 quadlet control registers, each implementing a different operation, then it can choose between creating 10 distinct CSRs at successive addresses, or a single 40 byte CSR that only accepts quadlet transactions.

In general it is suggested to avoid creating more address ranges than necessary, because although this makes things simpler for the application, it introduces more overhead to the system.

Receiving Asynchronous Streaming transactions

Asynchronous streaming packets have the same format and characteristics as isochronous stream packets. A user can thus perform reception of incoming asynchronous streaming packets via the FireAPI provided isochronous reception mechanisms. You can find a detailed explanation of the isochronous receive mechanism provided with FireAPI at the Isochronous Operations chapter of this manual. This chapter explains all about how to receive incoming packets at a specified channel and the various available methods in FireAPI that allow the user to do so. You can also find a variety of source samples that demonstrate reception of isochronous data that can be used without a single modification in order to receive incoming asynchronous streaming data.

Advantages of FireAPI Incoming Transaction Request Processing

Overall the advantages of incoming transaction request processing by FireAPI are:

- Queueing and storage of incoming requests by FireAPI.
- A single queue for requests from all mappings.
- Auto-cleanup of pending requests when range is freed.
- Transaction Filtering - Automatic Responses.
- Flexible client processing.
- Single notification for multiple requests.
- Processing when client is ready.
- One-by-One or Many-at-a-Time.
- In-Order or Out-of-Order responses.

Summary of Class Driver Transaction Processing functions

The functions provided by the class driver that can be used in transaction processing are:

1. **C1394MapAddressRange**: Creates a new address range or remaps an existing address range.
2. **C1394UnmapAddressRange**: Removes an address range mapping. The address range itself is deleted when its last mapping is removed.
3. **C1394GetNextRequest**: Retrieves the next transaction request from an address ranges request queue.
4. **C1394ServiceTransactionRequest**: Gets a pointer to an incoming transaction request packet and the memory that backs the address range, and performs the actions required including memory updates, response generation and transmission. This function is used when the request is legitimate and can be executed (a `resp_complete` will be returned).
5. **C1394SendResponse, C1394SendErrorResponse**: Even if a client needs to access the requests for some checks, it need not bother much with response generation. These two functions take all the appropriate steps to create valid and correct response packets.
6. **C1394CompletePacket, C1394CompletePackets**: Completes processing of transaction request packet that were not passed to any of **C1394ServiceTransactionRequest, C1394SendResponse, C1394SendErrorResponse**.

Common Errors in Transaction Processing

Developers working with address ranges should always keep the following things in mind:

- Each packet that is retrieved from the address range's request queue with **C1394GetNextRequest** must be completed with **C1394CompletePacket**. **C1394ServiceTransactionRequest**, **C1394SendResponse** and **C1394SendErrorResponse** internally call **C1394CompletePacket** so it would be an error to call **C1394CompletePacket** for a packet after calling one of those functions for the same packet.

- Unless **C1394GetNextRequest** returns NULL, you should not wait on the address range's event object again. For example, if an application knows that it only expects 1 packet from its peer application, then it must make 2 calls to **C1394GetNextRequest** otherwise the event object associated with the address range will never be signalled again.

- Application developers should fully understand the significance of the *uMaxRequestQueueItems* field of **C1394_ADDRESS_RANGE_CHARACTERISTICS**.

For example if this is set to 10, and an application sends 100 packets to this address range before the application that owns the range has the chance to call **C1394GetNextRequest**, then only the first 10 packets will be stored in the queue. The rest will be discarded by the class driver.

If these packets require a response then the class driver will put *resp_type_error* as the response code, and the sender will eventually get a **STATUS_1394_TRANSACTION_FAILED**.

If however the packets are unified write transactions (acknowledged with *ack_complete*), then the sender has no way of knowing that the receiver has not received these packets.

This flag is only a limit. It is not connected to how much memory will be allocated by the address range, or how many packets can the address range request queue physically hold.

For example if an application sets this to 10000, then this will not mean that its request queue can hold 10000 packets. Each incoming request packet consumes space in the asynchronous receive buffer used by the 1394 stack. By default, the asynchronous receive buffer can hold 256 packets, regardless of their size. If an application accepts 256 packets without retrieving them from their request queue then the 1394 stack will not be able to receive any more packets until these packets are retrieved from the request queue.

The size of the asynchronous receive buffer can be modified through the appropriate registry settings. An application can find out about the size of the asynchronous receive buffer by using **C1394QueryInformation** with the **OID_RECEIVE_BUFFER_SIZE** identifier.

Event Notifications

FireAPI provides a way for applications to get notified about various types of 1394-related events that occur during their course of operation.

Currently user mode applications can only be notified about bus reset start and bus reset complete events.

When an application requests a notification for an event it calls function **C1394RegisterNotification**. The application usually specifies an event handler that it wants to get called when this event occurs. An application can specify different event handler routines for various events, or it can use a single event handler for more than one notifications, as in the sample below.

When there is any notification for an application, the application's *notification event* is being signalled. An application gets a handle to this event by calling **C1394GetAsynchEventHandle**. When this event gets signalled, the application should call **C1394GetAsynchEvent** until this function returns **STATUS_1394_NOT_FOUND**, which means that there are no more notification events for the application.

Usually applications specify handler routines for the event types that they register. These handler routines get called from within **C1394GetAsynchEvent**. If the application has only registered notifications with event handlers, then a single call to **C1394GetAsynchEvent** is enough to handle all the events that were pending for the application at that moment. This is exactly what the sample code below does.

This means that **C1394GetAsynchEvent** acts more or less like an *event notification pump*.

The parameters of an event handler routine include the **CLIENT_ADAPTER_HANDLE** that the application specified when it opened the adapter with **C1394OpenAdapter**, the event type that is being indicated, an optional pointer to an event-specific information structure, and an additional context pointer that was specified into the call to **C1394RegisterNotification**.

When an application wants to stop receiving information about an event type, it should call **C1394UnregisterNotification**.

Registering a Bus Reset Notification

The sample code below, opens the first adapter and registers 2 notifications: *Bus Reset Start* and *Bus Reset Complete*.

It specifies the same handler routine for both events. Inside the handler routine, it prints out a message depending on the type of event being indicated. The main body of the program loops until 20 events are indicated and then proceeds to exit, doing a proper resource cleanup procedure.

This application also demonstrates a practical use of the **CLIENT_ADAPTER_HANDLE** value that is specified as a parameter to **C1394OpenAdapter**. The application maintains a structure (**APP_ADAPTER_INFO**) with information about the adapter that it opened. This structure contains the adapter's GUID and the **C1394_ADAPTER_HANDLE** returned by **C1394OpenAdapter**. The application specifies the pointer to the Info variable as its **CLIENT_ADAPTER_HANDLE**, and it is this pointer that is being returned to it as the first parameter of the event handler routine.

```
#include <windows.h>
#include <stdio.h>
#include <FireAPI.h>

typedef struct
{
    C1394_GUID          AdapterGuid;
    C1394_ADAPTER_HANDLE AdapterHandle;
    HANDLE              hWaitEvent;
}
APP_ADAPTER_INFO, *PAPP_ADAPTER_INFO;

//*****
void EventHandler( IN CLIENT_ADAPTER_HANDLE a_ClientAdapterHandle,
                  IN C1394_EVENT_TYPE      a_EventType,
                  IN PC1394_EVENT_PARAMETERS_STRUCT a_pEventParameters,
                  IN PVOID                  a_Context )
{
    PAPP_ADAPTER_INFO pInfo;

    pInfo = (PAPP_ADAPTER_INFO) a_ClientAdapterHandle;

    switch (a_EventType)
    {
        case EventPhyBusResetStart:
            printf( "EventPhyBusResetStart on adapter %I64X : %s\n",
                   SwapEndian64( *((ULONGLONG*) pInfo->AdapterGuid.Bytes) ),
                   a_Context );
            break;

        case EventPhyBusResetComplete:
            printf( "EventPhyBusResetComplete on adapter %I64X : %s\n",
                   SwapEndian64( *((ULONGLONG*) pInfo->AdapterGuid.Bytes) ),
                   a_Context );
            break;

        default:
            puts( "This will not happen.\n" );
    }
}

//*****
main(void)
{
    APP_ADAPTER_INFO Info;
    STATUS_1394      Status1394;
    ULONG            I;
    char             szBRStartMsg[] = "Bus Reset START Event";
    char             szBRCompleteMsg[] = "Bus Reset COMPLETE Event";

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    C1394GetAdapters( &Info.AdapterGuid, 1 );
}
```

```
C1394OpenAdapter( &Info.AdapterGuid,
                  (CLIENT_ADAPTER_HANDLE)&Info,
                  &Info.AdapterHandle );

// Set up a bus reset start and a bus reset complete handler.
Status1394 = C1394RegisterNotification( Info.AdapterHandle,
                                       EventPhyBusResetStart,
                                       NULL,
                                       szBRStartMsg, // context
                                       EventHandler );

// Did it fail? This should not happen.
if (STATUS_1394_SUCCESS != Status1394)
    return -2;

// Set up a bus reset start and a bus reset complete handler.
Status1394 = C1394RegisterNotification( Info.AdapterHandle,
                                       EventPhyBusResetComplete,
                                       NULL,
                                       szBRCompleteMsg, // context
                                       EventHandler );

// Did it fail? This should not happen.
if (STATUS_1394_SUCCESS != Status1394)
    return -3;

Info.hWaitEvent = C1394GetAsynchEventHandle(Info.AdapterHandle);

for ( I=0; I<20; I++)
{
    WaitForSingleObject( Info.hWaitEvent, INFINITE );
    C1394GetAsynchEvent( Info.AdapterHandle, NULL );
}

C1394UnregisterNotification( Info.AdapterHandle, EventPhyBusResetStart );
C1394UnregisterNotification( Info.AdapterHandle, EventPhyBusResetComplete );
C1394CloseAdapter( Info.AdapterHandle );
C1394Terminate();
return 0;
}
```

Using a separate thread for events

Applications can either use a separate thread to service all their notifications, or use a single thread that uses the Win32 functions **WaitForMultipleObjects** or **MsgWaitForMultipleObjects** as appropriate.

If the application uses a separate thread, then this thread could possibly wait on 2 event objects, the first being the event handle returned by **C1394GetAsynchEventHandle** and the other one created by the application with the Win32 **CreateEvent** function. The second event should be used by the application as the *Exit Signal* for the thread.

This is demonstrated by the sample code below, which is a variation of the previous sample and uses two different handler functions.

```
#include <windows.h>
#include <stdio.h>
#include <FireAPI.h>

#define DBG_MSG_PREFIX "---Sample--- "

typedef struct
{
    C1394_ADAPTER_HANDLE AdapterHandle;
    HANDLE hWaitEvent;
    HANDLE hExitEvent;
}
APP_ADAPTER_INFO, *PAPP_ADAPTER_INFO;

//*****
DWORD WINAPI EventThreadProc(void *Context)
{
    PAPP_ADAPTER_INFO pInfo;
    HANDLE WaitObjects[2];

    pInfo = (PAPP_ADAPTER_INFO)Context;

    WaitObjects[0] = pInfo->hWaitEvent;
    WaitObjects[1] = pInfo->hExitEvent;

    for (;;)
    {
        switch (WaitForMultipleObjects(2, WaitObjects, FALSE, INFINITE))
        {
            case WAIT_OBJECT_0:
                C1394GetAsynchEvent( pInfo->AdapterHandle, NULL );
                break;

            case WAIT_OBJECT_0+1:
                return 1;
        }
    }
}

//*****
void BusResetStart( IN CLIENT_ADAPTER_HANDLE a_ClientAdapterHandle,
                   IN C1394_EVENT_TYPE a_EventType,
                   IN PC1394_EVENT_PARAMETERS_STRUCT a_pEventParameters,
                   IN PVOID a_Context )
{
    PAPP_ADAPTER_INFO pInfo;

    pInfo = (PAPP_ADAPTER_INFO) a_ClientAdapterHandle;
    KdPrint(( DBG_MSG_PREFIX "BusResetStart Handler\n" ));
}

```

```

/*****
void BusResetComplete( IN CLIENT_ADAPTER_HANDLE      a_ClientAdapterHandle,
                      IN C1394_EVENT_TYPE          a_EventType,
                      IN PC1394_EVENT_PARAMETERS_STRUCT a_pEventParameters,
                      IN PVOID                      a_Context )
{
    PAPP_ADAPTER_INFO  pInfo;

    pInfo = (PAPP_ADAPTER_INFO) a_ClientAdapterHandle;
    KdPrint( ( DBG_MSG_PREFIX "BusResetComplete Handler\n" ) );
}

/*****
main(void)
{
    APP_ADAPTER_INFO  Info;
    ULONG             I;
    HANDLE            hEventThread;
    DWORD             dwThreadID;

    if (STATUS_1394_SUCCESS != C1394Initialize())
        return -1;

    C1394OpenAdapter( NULL, (CLIENT_ADAPTER_HANDLE)&Info, &Info.AdapterHandle );

    // Set up a bus reset start and a bus reset complete handler.
    C1394RegisterNotification( Info.AdapterHandle,
                               EventPhyBusResetStart,
                               NULL,
                               NULL,
                               BusResetStart );

    // Set up a bus reset start and a bus reset complete handler.
    C1394RegisterNotification( Info.AdapterHandle,
                               EventPhyBusResetComplete,
                               NULL,
                               NULL,
                               BusResetComplete );

    Info.hWaitEvent = C1394GetAsynchEventHandle(Info.AdapterHandle);
    Info.hExitEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    // Start our event thread.
    hEventThread = CreateThread( NULL, 4096, EventThreadProc, &Info, 0, &dwThreadID);

    if (NULL == hEventThread)
    {
        printf("CreateThread FAILED. Win32 error code is %u\n", GetLastError());
        return -4;
    }

    // Do our processing processing.
    for ( I=0; I<20; I++)
    {
        puts("Processing...");
        Sleep(1000);
    }

    puts("Exiting...");

    // Time to exit, signal the event thread.
    SetEvent(Info.hExitEvent);

    // Wait until the event thread has exited.
    WaitForSingleObject(hEventThread, INFINITE);

    CloseHandle(hEventThread);

    C1394UnregisterNotification( Info.AdapterHandle, EventPhyBusResetStart );
    C1394UnregisterNotification( Info.AdapterHandle, EventPhyBusResetComplete );
    C1394CloseAdapter( Info.AdapterHandle );
    C1394Terminate();
    return 0;
}

```


Notes on Bus Reset Processing

In the most usual case the *bus reset start* handler of an application will stop any operations that the application is doing, cleanup any internal tables and return.

Then the application's *bus reset complete* handler will read the bus topology, restore the information that the application was using and restart the application's operations.

Not all applications need bus reset handlers. An application can use **C1394GetBusResetCount** in conjunction with the return values of various functions in order to find out whether a bus reset has occurred.

Anyway, there are several important details to remember that are related to *post bus reset* processing:

- The *post bus reset* processing does not take the same amount of time to execute on all nodes. This means that it is possible that the *bus reset complete* handler of an application running in PC-1 is called while on PC-2 the *bus reset complete* event has not been indicated (**C1394IsBusResetInProgress** still returns TRUE on PC-2).
This means that if PC-1 sends a transaction request to PC-2 from within a *bus reset complete* handler, then the class driver on PC-2 will **abort** the packet because it has not still completed its bus reset processing. As a consequence, the transaction request will timeout or get *lost* if we are talking about a broadcast.
It is suggested that applications either delay their *post bus reset* processing by 10-20 msec, or be prepared for the case of their requests timing out or their broadcasts getting lost.
- The class driver follows the requirement of P1394a 2.1 paragraph 9.13 (modification of P1394a 2.0 paragraph 9.10) and will not permit a bus reset to occur for 2 seconds after the last bus reset occurred.

Isochronous Operations

Adapter Channels & DMA Channels

The *Adapter Channel* is the abstraction the API presents to client applications that want to perform *Isochronous Stream Operations*. Stream operations involve three kinds of activities: *Isochronous Receive*, *Isochronous Transmit* and *Asynchronous Stream Receive*²².

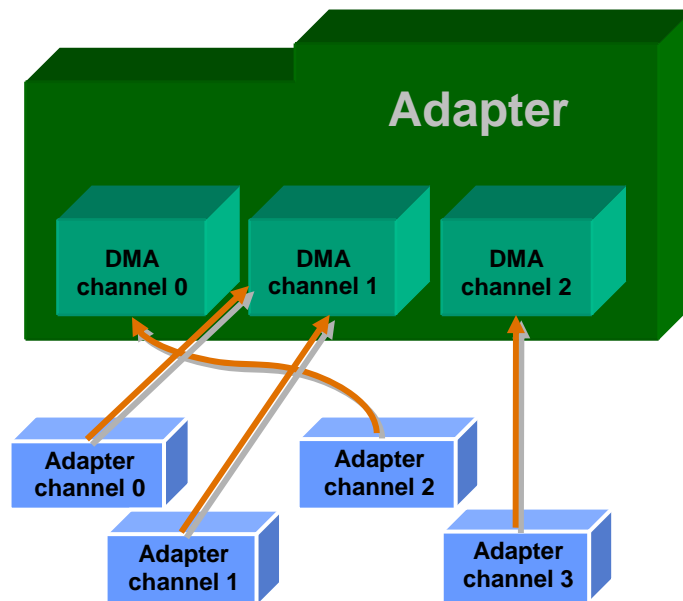
It must be emphasized that the term *Adapter Channel* abstracts an actual *DMA channel* on the adapter (a *DMA context* according to OHCI terminology). The *Adapter Channel* should not be confused with the term *isochronous channel* or *stream*.

An *isochronous stream* is identified by an *isochronous channel number* that is used in the header of each packet.

In this documentation the terms *isochronous channel* and *isochronous stream* refer to a sequence of stream packets that appear *on the cable* and all use the same *isochronous channel number*.

An *Adapter Channel* is an abstraction of the *DMA channel* because it is possible (in OHCI 1.1 or later chips) that the Class Driver uses the same *DMA channel* to service more than one *Adapter Channels*. This means that there will not necessarily be a 1-to-1 correspondance between *Adapter Channels* and *DMA channels*.

This concept is illustrated in the following figure.



Adapter Channels vs DMA Channels

Each *DMA channel* can either be a *Receive DMA channel* or a *Transmit DMA channel*. This means that the *DMA channels* are neither generic nor interchangeable. Each 1394 chip has a given number of *DMA channels* available for isochronous receive and transmit.

The OHCI specification for example *requires* a minimum of 4 and *allows* a maximum of 32 *DMA channels* of each type. Typical 1394 adapter implementations provide 4 isochronous receive *DMA*

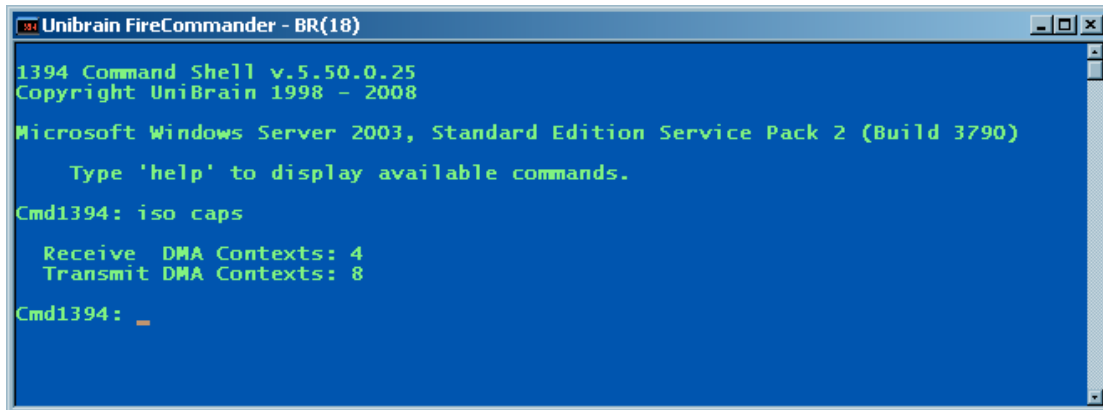
²² Asynchronous stream transmit is performed through the asynchronous transmit functions, specifically `C1394AsynchronousTransmit`.

channels and 4 isochronous transmit DMA channels, while it is not very unusual to find an adapter with 8 isochronous transmit DMA channels.

In most applications, what is important on the PC side is isochronous receive capability. Engineers and system designers must have a clear picture of the 1394 chip capabilities and FireAPI provides programmatic access to this information.

FireAPI defines the object identifiers **OID_ISO_RECEIVE_DMA_CONTEXTS** and **OID_ISO_TRANSMIT_DMA_CONTEXTS** that can provide this information through the **C1394QueryInformation** function.

This is the same information that the FireCommander tool reports through the ISO CAPS command:



```

Unibrain FireCommander - BR(18)
1394 Command Shell v.5.50.0.25
Copyright UniBrain 1998 - 2008

Microsoft Windows Server 2003, Standard Edition Service Pack 2 (Build 3790)
Type 'help' to display available commands.

Cmd1394: iso caps
    Receive DMA Contexts: 4
    Transmit DMA Contexts: 8

Cmd1394: _
  
```

An *Adapter Channel* can be used to perform any kind of stream operation on any isochronous stream (channel numbers 0 to 63).

An *Adapter Channel* accepts *stream requests*²³ and executes them. It is the *stream request* itself that identifies the *isochronous channel number(s)* to be used in the actual execution of the operation. In fact a *stream request* may involve more than one *isochronous stream*.

This gives complete flexibility to applications and allows them to implement any kind of design that best suits their needs.

In the most usual case an application will only care to use an adapter channel in order to transmit or receive a single isochronous stream.

In more complex designs an application can use a single *adapter channel* to transmit or receive multiple *isochronous streams*. For example an application can pass to an *adapter channel* a buffer that contains interleaved packets for 3 different outgoing isochronous streams.

However, these advanced capabilities are tightly connected to the capabilities of the actual adapter that is being used. For example on adapters based on the PCILynx family of chips, *adapter channels* have the capability to transmit packets for more than one isochronous stream on the same isochronous cycle. Applications should take into consideration the supported behavior of each adapter and utilize it as appropriate. For instance, a video server application could be designed so that instead of sending 3 packets of 200 bytes on each cycle (for 3 different isochronous streams), send one packet of 600 bytes on each cycle, and have each isochronous stream appear every 3rd isochronous cycle.

This way an application can perform quite complex tasks using only a single *adapter channel*. As a result the adapter resources, which are limited, are better utilized, and more than one applications can perform isochronous operations at the same time.

²³ Also referred to as *isochronous requests*, *isochronous commands*, *stream commands* or simply *commands*.

DMA Multiplexing

As stated earlier the mapping between *Adapter Channels* and *DMA channels* is not always 1-1. This is true only for isochronous receive DMA channels on 1394 adapters that use OHCI chips.

According to the OHCI specification each isochronous receive *DMA channel* can be programmed to receive only one isochronous stream (isochronous channel number). This means that the maximum number of isochronous streams that a software system can receive is determined by the number of isochronous receive DMA contexts found on the 1394 adapter in use.

In several cases, designers implement systems with multiple 1394 adapters connected to the same 1394 bus simply for the purpose of getting more isochronous receive DMA contexts.

However, the OHCI specification allows for exactly one isochronous receive DMA context to be configured as “shared”, that is to be set up in a way that it may receive isochronous packets from multiple channel numbers. This way the software that controls the OHCI chip can overcome the limitation imposed by the number of available isochronous receive DMA contexts.

Programming a DMA context in this “shared” mode is significantly more complex compared to the “dedicated” DMA contexts, requiring delicate and timely handling, which is the main reason why software vendors had not implemented this feature before.

Unibrain has finally made this feature available in ubCore 5.50 and FireAPI 5.50, in an absolutely transparent way for applications. There is no special code required for an isochronous receive operation to run on the “shared” MultiDMA context. The binary of the application will run in exactly the same manner, regardless of whether it is running on a “dedicated” DMA context or on the “shared” MultiDMA context.

Still the application developer can have control of where the isochronous receive operations execute and thus configure the solution as desired. Currently the determining factor is the order in which *Adapter Channels* are opened.

An *Adapter Channel* will run on a “dedicated” DMA context if:

1. There are at least two free isochronous receive DMA contexts.
2. There is only one free DMA context, but another DMA context is already operating in “shared” mode.

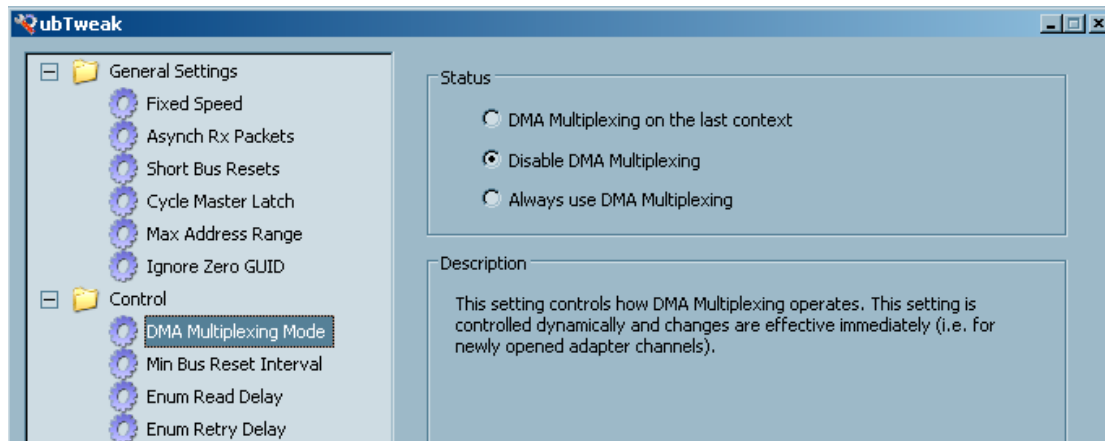
At the moment of this writing there is no direct way to **force** an adapter channel to run on the “shared” DMA context, although there are free DMA contexts. Unibrain is considering adding such a capability in future versions of FireAPI.

DMA Multiplexing Modes

In order to make ubCore 100% backwards compatible and not break existing code and running systems, it is possible to operate the Firewire adapter in the “Only Dedicated DMA context” mode, in fact this is the default mode of isochronous receive operation.

The Isochronous Receive DMA mode is now a configuration setting of ubCore. Actually it is not only a registry setting that determines how the system operates at startup, but can also be changed dynamically for easier testing.

The setting is actually named **DMA Multiplexing Mode**, as shown below in the ubTweak utility:



- Always use DMA Multiplexing:** When this option is selected, then all isochronous receive operations get executed on the “shared” DMA context, leaving the rest of the isochronous receive DMA contexts inactive. This option has been added primarily for testing reasons. It allows application designers to easily stress test the DMA Multiplexing implementation of both the software and the underlying 1394 adapter.
- Disable DMA Multiplexing:** This is the default option after the installation of ubCore 5.50, for reasons of backwards compatibility. When this option is active then all isochronous DMA contexts are operated in “dedicated” mode.
- DMA Multiplexing on the last context:** When this option is selected then the operation of MultiDMA is enabled. The option name actually describes the internal logic of isochronous receive DMA programming. When there are more than one available isochronous receive DMA contexts then a newly opened *isochronous adapter channel* is operated in “dedicated” mode. When there is only one available, the last free one, then the DMA context is operated in “shared” mode. This of course means that if there are 4 isochronous receive DMA contexts available and the application sets up isochronous receive on 4 channels, the fourth will be running on “shared” mode, even though it will be the only one sharing the DMA context. This is implemented this way because it is technically impossible to shift a DMA context from “dedicated” to “shared” without disrupting isochronous receive on the context, which would result in at least one failed isochronous operation for an application, simply because another application tried to set up its own isochronous receive operations. In practice however, solution designers usually know the number of isochronous operations that will be running at one time, and if they require a maximum of 4 then they can simply disable MultiDMA.

Changing the DMA Multiplexing Mode setting from ubTweak saves the new value in the registry and optionally immediately applies it.

You can also use the **MULTIDMA** command in FireCommander to see and dynamically change the current setting, without saving it in the registry.

Type **MULTIDMA** /? to see the supported options as shown below:

```
Unibrain FireCommander - BR(8)
1394 Command Shell v.5.50.0.25
Copyright UniBrain 1998 - 2008

Microsoft Windows Server 2003, Standard Edition Service Pack 2 (Build 3790)
Type 'help' to display available commands.

Cmd1394: MULTIDMA /?
    USAGE: MULTIDMA [DEFAULT|LAST|DISABLE|ALWAYS]

Cmd1394: MULTIDMA
    Current MultiDMA Mode: DISABLED (default)

Cmd1394:
```

Entering **MULTIDMA** without parameters displays the current setting.

The DMA Multiplexing Mode value **can only be dynamically changed** when there are no isochronous receive adapter channels open. Doing a dynamic change of the operating mode is supported mainly for testing reasons, but could theoretically be utilized in some specialized scenarios as well.

The object identifier defined for querying and controlling the DMA Multiplexing Mode through **C1394QueryInformation** and **C1394SetInformation** is **OID_MULTIDMA_MODE**, which uses a ULONG argument with values from the following enumeration:

```
typedef enum
{
    MultiDMAOnLastContext = 0,
    MultiDMADisabled = 1,
    MultiDMAForced = 2,
}
MultiDmaOperation;
```

Opening an Adapter Channel

An application opens an *Adapter Channel* by calling **C1394OpenAdapterChannel**. *Adapter Channels* have an associated type, which defines the type of operation they can be used for. This can be *Isochronous Receive*, *Isochronous Transmit* or *Asynchronous Stream Receive*.

For each *Adapter Channel* the application should specify the maximum number of isochronous packets that it intends to specify in a single *isochronous request*. This is indicated by specifying the **PACKETS_PER_REQUEST** flag in the *Adapter Channel* options. For more information see the description of **C1394OpenAdapterChannel**.

When an application has completed its use of an *Adapter Channel*, it should close it by calling **C1394CloseAdapterChannel**.

The sample code below demonstrates how to open an *Adapter Channel* for isochronous receive on the default adapter. The application indicates that it intends to use a maximum of 640 isochronous packets in each *isochronous request*.

```
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE      AdapterHandle;
    C1394_CHANNEL_HANDLE      ChannelHandle;
    HANDLE                    hStartProcessingEvent;
    STATUS_1394               Status1394;
    FIREAPI_CHANNEL_PARAMETERS ChannelParams;

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &AdapterHandle );

    // Initialize the channel parameters structure.
    ChannelParams.Tag = TAG_FIREAPI_CHANNEL_PARAMETERS;
    ChannelParams.AdapterChannelType = ChannelIsochReceive;
    ChannelParams.IsochReceive.fAdapterChannelOptions = PACKETS_PER_REQUEST;
    ChannelParams.IsochReceive.uMaxPacketsPerRequest = 640;

    // Try to open the adapter channel.
    Status1394 = C1394OpenAdapterChannel( AdapterHandle,
                                         &ChannelHandle,
                                         &hStartProcessingEvent,
                                         NULL,
                                         &ChannelParams );

    if (STATUS_1394_SUCCESS != Status1394)
    {
        printf( "C1394OpenAdapterChannel failed with status %s\n",
              C1394StatusString(Status1394) );
        return -1;
    }

    puts( "Adapter channel opened." );

    C1394CloseAdapterChannel( AdapterHandle, ChannelHandle );

    C1394CloseAdapter( AdapterHandle );
    C1394Terminate();
    return 0;
}
```

Enabling stream channel numbers for an Adapter Channel

After an application opens an *Adapter Channel* it has to inform the 1394 stack about which stream channel numbers it intends to use through this adapter channel.

This is necessary so that the 1394 stack can:

- Make sure that the same channel number is not used on two different *Adapter Channels* on the same PC.
- Perform parameter validation checks.

The 1394 stack maintains for each *Adapter Channel* a 64-bit mask which has a bit set for each stream channel number that the application has enabled for the *Adapter Channel*. This is called the *Channel Mask*. An application can retrieve the current value of the *channel mask* for an adapter channel by using **C1394QueryInformation** and specifying the **OID_CHANNEL_MASK** object identifier.

Similarly the application can use **C1394SetInformation** with the **OID_CHANNEL_MASK** identifier in order to set the value of the *channel mask* for an *Adapter Channel*.

The sample application below demonstrates how to use **C1394SetInformation** in order to enable the channel number that it intends to use with its *adapter channel*.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE      AdapterHandle;
    C1394_CHANNEL_HANDLE      ChannelHandle;
    HANDLE                    hStartProcessingEvent;
    STATUS_1394               Status1394;
    ULONG                     uChannelNumber;

    FIREAPI_CHANNEL_PARAMETERS ChannelParams;
    CHANNEL_MASK_STRUCT       ChannelMaskStruct;

    srand(GetTickCount());

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &AdapterHandle );

    // Initialize the channel parameters structure.
    ChannelParams.Tag = TAG_FIREAPI_CHANNEL_PARAMETERS;
    ChannelParams.AdapterChannelType = ChannelIsochReceive;
    ChannelParams.IsochReceive.fAdapterChannelOptions = 0;

    // Try to open the adapter channel.
    Status1394 = C1394OpenAdapterChannel( AdapterHandle,
                                         &ChannelHandle,
                                         &hStartProcessingEvent,
                                         NULL,
                                         &ChannelParams );

    if (STATUS_1394_SUCCESS != Status1394)
        return -1;

    puts("Adapter channel opened.");

    // Select a random channel number other than 31.
    do
        uChannelNumber = (rand() % 64);
    while (uChannelNumber == 31);
}
```



```

// Enable a channel number for this adapter channel.
ChannelMaskStruct.ChannelHandle = ChannelHandle;
ChannelMaskStruct.UChannelMask = ((ULONGLONG)1) << uChannelNumber;

Status1394 = C1394SetInformation( AdapterHandle,
                                OID_CHANNEL_MASK,
                                &ChannelMaskStruct,
                                sizeof(ChannelMaskStruct) );

switch (Status1394)
{
    case STATUS_1394_SUCCESS:
        printf( "Enabled channel number %u for adapter channel.\n",
              uChannelNumber );
        break;

    case STATUS_1394_CONFLICT:
        printf( "Channel number %u appears to be in use.\n",
              uChannelNumber );
        return -2;

    default:
        printf( "C1394SetInformation failed with status %s\n",
              C1394StatusString(Status1394) );
        return -2;
}

C1394CloseAdapterChannel( AdapterHandle, ChannelHandle );

C1394CloseAdapter( AdapterHandle );
C1394Terminate();
return 0;
}

```

It is important to note the following:

- Enabling a channel number on an *adapter channel* is completely unrelated to allocating a channel number from the IRM.
- In general applications should check for a return code of **STATUS_1394_CONFLICT** from **C1394SetInformation** when trying to set the channel mask. If this value is returned then it means that one or more of the channel numbers that the application is trying to enable are already enabled by other adapter channel.
- When an adapter channel is closed, the channel numbers that were enabled for it are automatically freed by the 1394 stack, so the application need not set the channel mask to all zeroes before closing an adapter channel.
- In an application that has tested and debugged, **C1394SetInformation(OID_CHANNEL_MASK)** will not return any other status than **STATUS_1394_SUCCESS** or **STATUS_1394_CONFLICT**. All other return codes have to do with passing invalid parameters (like an invalid adapter handle, or channel handle, an invalid OID etc), and will never be returned in an application that operates correctly.

Using **OID_CHANNEL_MASK** makes things a little bit more complicated if an application wants to enable or disable more than one channel numbers. This is because calling **C1394SetInformation** with **OID_CHANNEL_MASK** replaces the channel mask which means that it possibly performs more than one enable/disable actions at a time.

For example if an application has already enabled some channels, and at some point needs to enable another 2 channel numbers, then using **OID_CHANNEL_MASK** it would have to retrieve the current channel mask using **C1394QueryInformation**, OR-in the new channel numbers and set it back with **C1394SetInformation**.

In order to make it easier for application writers to perform such tasks, FireAPI also includes two OIDs²⁴ in addition to **OID_CHANNEL_MASK**.

There are **OID_CHANNEL_MASK_ENABLE** and **OID_CHANNEL_MASK_DISABLE**.

²⁴ OIDs is short for Object IDentifiers.

OID_CHANNEL_MASK_ENABLE enables the channel numbers specified in the provided mask, without affecting the other channel numbers that are possibly currently set into the channel mask of the adapter channel.

Similarly **OID_CHANNEL_MASK_DISABLE** disables the channel numbers specified in the provided mask, without affecting the other channel numbers that are possibly currently set into the channel mask of the adapter channel.

The following sample demonstrates how to use **OID_CHANNEL_ENABLE** and **OID_CHANNEL_DISABLE** to selectively enable and disable channel numbers for an *adapter channel*, without having to retrieve the channel mask each time.

```

for (I=0; I<5; I++)
{
    // Select a random channel number other than 31.
    do
        uChannelNumber = (rand() % 64);
    while (uChannelNumber == 31);

    // Enable this channel number for the adapter channel.
    ChannelMaskStruct.ChannelHandle = ChannelHandle;
    ChannelMaskStruct.UChannelMask = ((ULONGLONG)1) << uChannelNumber;

    Status1394 = C1394SetInformation( AdapterHandle,
                                    OID_CHANNEL_MASK_ENABLE,
                                    &ChannelMaskStruct,
                                    sizeof(ChannelMaskStruct) );

    switch (Status1394)
    {
        case STATUS_1394_SUCCESS:
            printf( "Enabled channel number %u for adapter channel.\n",
                  uChannelNumber );
            break;

        case STATUS_1394_CONFLICT:
            printf( "Channel number %u appears to be in use.\n",
                  uChannelNumber );
            return -2;

        default:
            printf( "C1394SetInformation failed with status %s\n",
                  C1394StatusString(Status1394) );
            return -2;
    }
}

```

Adapter Channel Operating Models

In the kernel mode API there are 2 basic operating models that are supported by *adapter channels*: *Queued-Completion* and *Instant-Completion*. Client drivers can use either one, and if necessary a mixture of them.

In the user mode API, the operating model is a little more restricted, as there is no capability for direct callback calls. An application necessarily uses the *Queued-Completion* model for **ALL** isochronous requests, but can also associate an independent event handle with specific requests.

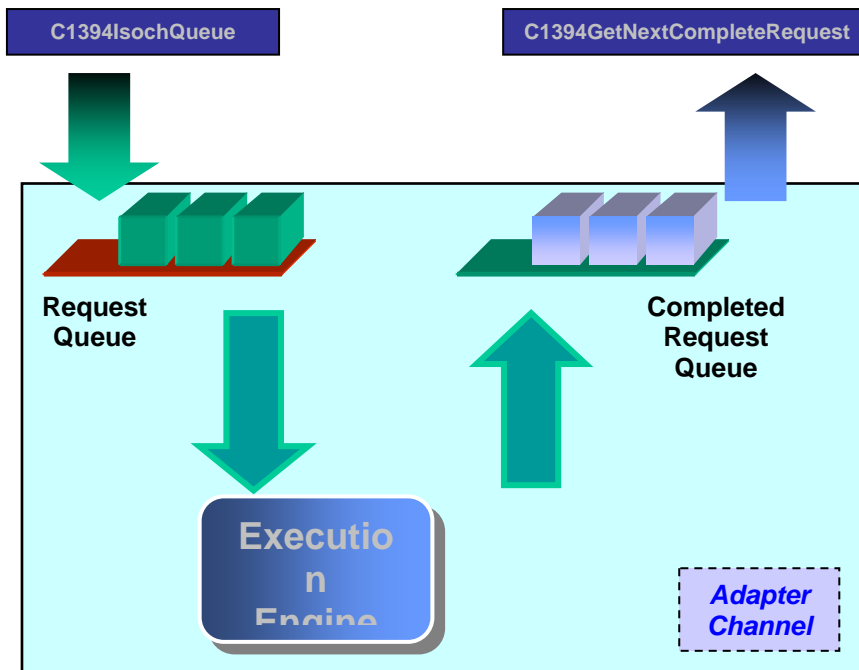
The application submits to the 1394 stack one or more *isochronous requests* for an *adapter channel* and then these requests start executing one-by-one, 'in the background'. This means that the 1394 stack programs the 1394 adapter to execute these operations, and then lets the adapter execute them without any CPU intervention.

Usually, applications queue more than one *isochronous requests*, so that the *adapter channel* can continue executing the next *isochronous request(s)*, while the application is processing the outcome of the completion of a previous request.

This way, the application can ensure that no incoming isochronous packets are missed in the case of isochronous receive, and no cycles remain idle²⁵ in the case of isochronous transmit.

An application uses **C1394IsochQueue** in order to queue one or more isochronous requests for execution by the *adapter channel*. The isochronous requests are being executed in the exact same order in which they were submitted²⁶ by the application.

When an isochronous request is submitted it gets stored into the adapter channel's *Request Queue*. When an isochronous request has completed its execution it is stored in the *Completed-Request Queue*, from where the application can retrieve it by calling **C1394GetNextCompleteRequest**.



Logical Structure of an Adapter Channel

²⁵ Unintentionally idle. An application might specifically request for a number of idle cycles.

²⁶ It is the responsibility of the application to submit isochronous requests in a serialized fashion. If the application makes concurrent calls that submit isochronous requests for the same adapter channel, then the relative order of the two sets of isochronous requests will be randomly selected depending on which thread obtained some internal synchronization objects first.

In principle different *adapter channels* operate completely independently of one another. However in actual hardware implementations DMA channels are not completely independent of one another, but instead are prioritized and the operation of a higher priority channel can affect the operation of a lower priority DMA channel.

This can happen if the higher priority DMA channel completely ties up the 1394 chip of the adapter and the chip does not perform an internal context-switch to the lower priority channel in time to complete the requested operations.

In general application developers should verify that the 1394 adapter that they intend to use can perform the tasks that they have in mind, especially if they plan to use more than one adapter channels.

PCILynx-based adapters can handle total isochronous load of up to 4KB per isochronous cycle, provided that the FIFO has been configured appropriately. In cases where a lot of isochronous traffic is involved, application designers should also consider using two 1394 adapters in order to perform their operations without heavy constraints on the FIFO sizes.

The sample code of FireAPI includes a sample application that can generate a constant stream of isochronous traffic of a specified packet size²⁷, and an application that can continuously receive an isochronous stream of a specified packet size.

Using these two applications developers can simulate the demand that their system will put on the 1394 adapters and make sure that the hardware they use is suitable for the task they are after.

²⁷ The packet size is specified on the command line.

Queued-Completion Model

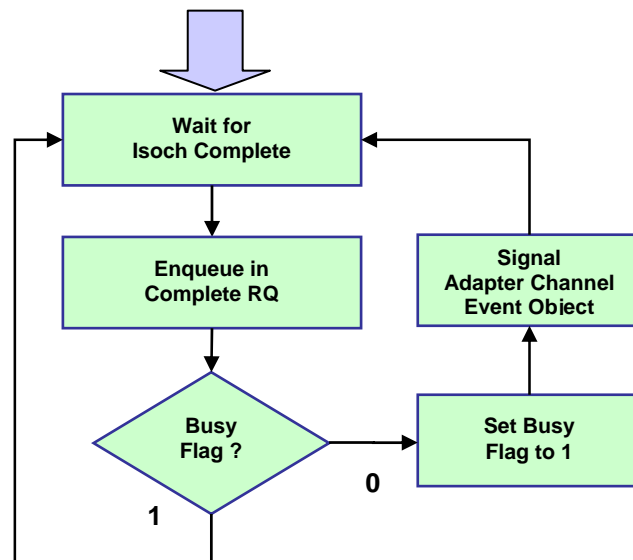
The *Queued-Completion* model closely resembles the operating model of an address range:

- Isochronous requests for an *adapter channel* are being submitted with calls to **C1394IsochQueue**.
- When a request is completed, then the 1394 stack will insert the completed request into the adapter channel's *complete-request queue*.
- Before the completed request is inserted in the queue the 1394 stack checks if the adapter channel's *Completed-Request Queue* is empty and its *Busy Flag* is clear. If this is the case then the 1394 stack notifies the application that there is at least one complete request for this adapter channel, by setting event object associated with the adapter channel. At the same time the 1394 stack sets the *Busy Flag* for this adapter channel.
- The client can then repeatedly call **C1394GetNextCompleteRequest** in order to retrieve the completed commands from the adapter channel's *Completed-Request Queue*.
- When the *Completed-Request Queue* is emptied, **C1394GetNextCompleteRequest** returns NULL and at the same time resets the adapter channel's *Busy Flag* and the channel's event object.
- If any other isochronous requests complete while the adapter channel's *Busy Flag* is set (which implies that the application is still in its processing loop), then the completed request will be queued but the client's event object will not be signalled again.

The *Queued-Completion* model of operation makes it much easier for clients to process the completed requests, because it imposes an inherent serialization in the processing of completed request, and also provides queuing of all completed isochronous requests.

The application need not provide any kind of management for its isochronous requests. It is not necessary that the application maintains information on which requests are pending, which one are completed and demand application-side processing, which ones have failed, etc. The 1394 stack takes the complete responsibility for maintaining all this information, and presents to the application an interface through which the application can find out about completed requests.

The operating logic of isochronous request completion is illustrated in the following flowchart.



Isochronous Request Queued-Completion Logic

When an isochronous request has completed, the adapter raises an interrupt and automatically starts executing the next isochronous request. This is the reason why the action of starting the next isochronous request is not shown anywhere in this flowchart.

The *Busy Flag* is zero-initialized, and is being reset to zero each time NULL is returned by **C1394GetNextCompleteRequest**.

Instant Completion Notification

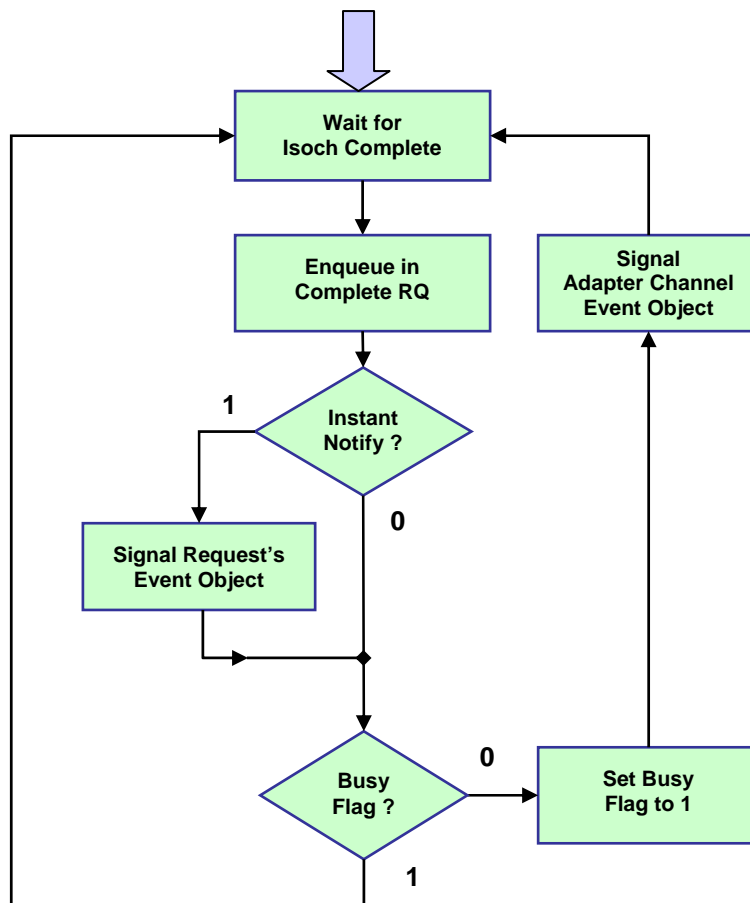
The *Queued-Completion* model has a property that could be seen as a drawback for some kinds of applications. Specifically, the application cannot be informed “timely” when a specific isochronous command completed, unless it completely empties the complete-request queue each time it gets notified.

Then it knows that it will get its event signalled immediately when the next request completes, but at the same time loses another advantage that the *Completed-Request Queue* offers. Once it retrieves the requests from the queue, then it has to manage these requests on its own (using a private queue, a table or some other structure).

Because this “timely” notification might be critical for some applications, the 1394 stack provides a method, through which the application can receive an immediate notification at the exact moment when an specific isochronous command is completed. This is received through an event object, that can be separately associated with each isochronous request.

However the request will still be queued in the adapter channel’s *Completed-Request Queue*²⁸.

The flowchart below expands the logic of the previous page to include the *Instant-Completion* processing logic.



Isochronous Request Completion Logic

²⁸ In kernel mode, the client has the option to directly retrieve this request, without having it stored into the *Completed-Request Queue*.

Isochronous Request Types

FireAPI includes several different isochronous requests in order to provide developers with all the functionality that they may need.

The list of supported isochronous requests and a short description of their functionality is shown in the table that follows:

Title	Description
Receive Fixed Packets	<p>An isochronous receive operation, where complete isochronous packets (header quadlet + data) are received in a virtually contiguous buffer.</p> <p>A maximum size M (in quadlets) is specified for the isochronous packets to be received. The Nth packet received in the buffer is stored at offset (M+1)*4*N.</p> <p>If a packet smaller than M quadlets is received, then simply a couple of bytes stay unused²⁹. If a packet larger than M quadlets appears on the isochronous stream, then the packet will either be ignored or will be partially received depending on the operation-specific flags set for this operation.</p>
Receive Fixed Data	<p>Similar to <i>Receive Fixed Packets</i>, with the difference that the headers and the data are received into <u>two separate buffers</u>. This way the isochronous payload data appear <u>contiguously</u> in memory.</p> <p>A maximum size is specified for the isochronous packets to be received. The caller can request that more than one quadlets of each packet are moved into the header buffer. This way the caller can strip additional <i>protocol headers</i> from each isochronous packet.</p> <p>This method is known to present problems on some 64-bit machines with lots of memory (data corruption during DMA) so in these cases it is suggested that you either use the <i>Receive Fixed Packets</i> method or the <i>Receive Fixed Data No Headers</i> method described below.</p>
Receive Fixed Data No Headers	<p>Similar to <i>Receive Fixed Data</i>, with the difference that the isochronous packet header (1 quadlet) is discarded. The isochronous payload data is received in a single buffer and appears <u>contiguously</u> in memory.</p> <p>A maximum size is specified for the isochronous packets to be received.</p>
Transmit Fixed Packets	<p>An isochronous transmit operation, where complete isochronous packets (header quadlet + data) are layed out in a virtually contiguous buffer, each one starting at a multiple of a fixed offset.</p> <p>A maximum size M (in quadlets) is specified for the isochronous packets to be transmitted. The Nth packet is stored in the buffer at offset (M+1)*4*N.</p>
Transmit Packets	<p>An isochronous transmit operation, where complete isochronous packets (header quadlet + data) are layed out in a virtually contiguous buffer, one after the other, with each packet starting at the next quadlet boundary after the previous packet.</p>
Transmit Data	<p>An isochronous transmit operation, where the caller provides two buffers. The first is the header buffer, which can contain $H \geq 1$ quadlets for each isochronous packet to be transmitted. The second is the data buffer which contains the rest of the payload bytes for each packet.</p>
Idle Cycles	<p>An operation that allows a client to request a certain number of idle cycles. This is mostly useful for transmit channels, but it is also supported on receive channels as well.</p>

Table 4. FireAPI Isochronous Operations

²⁹ Bytes stay unused between the end of the 'small' packet and the location where the next isochronous packet will be stored.

Any operation that contains the keyword “*Packets*” describes an operation that processes complete isochronous packets, while operations that contain the keyword “*Data*” describe operations that involve isochronous packets that are split in two parts, a ‘header’ part of 1 or more quadlets and a data part.

Any operation that contains the keyword “*Fixed*” describes an operation where fixed values are used for the boundaries where isochronous packets/data are/will be stored. Such operations are best suited for applications that deal with isochronous streams which only contain packets of a constant size. For example all formats of 1394 digital cameras³⁰ that transmit uncompressed data (YUV or RGB encoded) uses packets of fixed sizes. However these operations also support streams whose packet size is variable, provided that the maximum packet size is known in advance.

Operations that don’t contain the keyword “*Fixed*” describe operations where the boundaries where packets/data are stored are not fixed, but the packets/data are contiguous.

Some examples are necessary in order to clarify all these concepts.

Example 1

Isochronous Request: Receive Fixed Data, Receive 400 isochronous packets, Fixed packet size is 160 quadlets, Header size is 5 quadlets.

This is a type of *buffer-fill* isochronous receive.

When isochronous packets will be received, the first 5 quadlets of each packet (1 header quadlet and 4 payload quadlets) will be received into the *header buffer* of this request, and the rest 155 quadlets (the *pure payload*) will be received into the *data buffer*.

The ‘header’ of the Nth isochronous packet will be at byte offset $4*5*N$ in the *header buffer*, and the rest of the payload will be at byte offset $4*155*N$ in the *data buffer*.

This command will be processed by the 1394 chip, and when 400 isochronous packets get received, then a hardware interrupt will occur and the command will be indicated as *complete*. At the same time the 1394 chip will automatically continue its isochronous operations by executing the next isochronous request that is queued for the adapter channel.

If a packet whose size is 140 quadlets appears on the stream, then this packet will be received as well. However since packets get received at fixed offsets, the pure payload of this packet will not fill the 155 quadlets that are ‘allocated’ for it, but only 135 quadlets. The next packet will be received on the next 155-quadlet boundary, so a ‘hole’ will be created in the data buffer.

Example 2

Transmit Data, 920 Packets, Header size is 1 quadlet.

In this case the *header buffer* only contains the header quadlet for each isochronous packet. The header quadlet of the Nth packet is located at byte offset $4*N$ inside the *header buffer*. The size of the *header buffer* is $4*920$ bytes.

The amount of data that will be retrieved from the *data buffer* for each packet depends on the value of the *data_length* field in the header quadlet of the packet.

The offset of the payload bytes of the Nth packet is not known beforehand. It is equal to the sum of the *data_length* field of all previous packets.

When the 1394 chip finishes the transmission of the last packet for this request, it will raise an interrupt in order to indicate the completion of this request, and automatically continue executing the next command that is queued for the adapter channel.

³⁰ Cameras that comply with the *1394-Based Digital Camera Specification*, version 1.04 or 1.20.

Why use 'Packet' or 'Fixed' operations?

It is obvious that *Fixed* operations are a subset of the functionality of *non-Fixed* operations. Moreover *Packet* operations are less practical than *Data* operations, because the data are not contiguous.

However there is an important reason why these operations are included in FireAPI: **Performance**.

It is important to realize that isochronous operations are actually executed by the 1394 chip that is found on the 1394 adapter. For example it is the 1394 chip that performs an isochronous receive and splits an incoming packet in 2 buffers, by doing 2 DMA transfers³¹ per packet.

The drivers prepare a set of instructions that the 1394 chip will execute in order to perform the required isochronous operations. Each of these instructions has to be transferred over the PCI bus to the 1394 chip, quadlet by quadlet, and get executed by the chip. By itself the execution time of each of those commands is usually in the order of 10-40 microseconds, and to this we have to add the required amount of time for transfer of the isochronous payload. For example a 2KB isochronous packet at S400 requires 40 microseconds to be received from the 1394 bus.

On the other hand the isochronous cycle is 125 microseconds, of which only the 100 microseconds are available for isochronous traffic.

A simpler set of instructions for the 1394 chip means two things:

1. Less overhead on the PCI bus and on the system in general.
2. The 1394 chip has more 'time' per cycle to execute isochronous commands from other adapter channels.

The more complex these instructions get, the more tied-up the 1394 chips become.

When an application performs a *fixed packet receive* the drivers prepare a simpler set of instructions for the adapter to execute, because they know in advance the location where each packet should get received into, and the adapter needs to fetch less instructions from main memory and only perform a single logical DMA transfer per packet.

This would leave more free processing time to the 1394 chip which could use it to perform an additional isochronous receive or transmit at the same time.

In an isochronous *fixed data receive*, the drivers once more know in advance the locations where headers and data get stored. However this time they have to prepare a more complex set of instructions because the chip needs to know two (*address-length*) pairs for each isochronous packet.

In *isochronous transmit* the differences between *Fixed* and *non-Fixed* operations are not that crucial, because in all cases the drivers have in advance the headers of all the packets that will be transmitted in a request, and so they can calculate the exact offset where each packet/payload starts.

However the difference between *Packet* and *Data* operations are the same; in *Data* operations the 1394 adapter will have to execute slightly more instructions and perform 2 DMA transfers instead of 1.

In conclusion, there is always a tradeoff. The simpler the set of instructions that the 1394 chip executes, the more complex the processing that the application has to do gets. Application designers should study the tradeoffs in conjunction with their system requirements, probably execute some simulation of their system and then decide which operations to use.

In general, Unibrain suggests that relatively small software complexity is preferable to over-loading the 1394 chip.

- If an application can receive complete isochronous packets and process them in-place, without having to copy the payloads to another memory location so that data becomes contiguous, then it should opt for this design, unless there is too much extra coding complexity involved.

³¹ The actual number of DMA transfers performed on the PCI level might be more, depending on how busy the system is. By saying 2 DMA transfers, we mean 2 *logical* transfers where a logical DMA transfer is a destination address and a number of bytes to transfer to it.

- If *packet receive* means that the application has to copy the data elsewhere in order to do its processing, the stream bandwidth is quite big and the system is already heavily loaded, then designers should probably opt for a *data receive*.
- If the application does not have to copy the data regardless of whether it uses *packet receive* or *data receive*, and the target system will not be under heavy load, then *data receive* would be preferable.

Design Examples

For example³², suppose that we have an application that wants to receive from the 1st 1394 adapter the image of a 1394 camera, in the format [640x480, YUV 4:2:2, 30 fps], convert the image to RGB565, and retransmit it through the 2nd 1394 adapter that is installed on the PC.

YUV 4:2:2 uses 16-bits per pixel, which is the same as RGB565. This means that the amount of outgoing data is the same as the amount of incoming data. In turn this implies that the application can perform the conversion in place, and not use a separate output buffer. This will improve locality of reference and as a result improve performance.

The application has to convert each source pixel from the YUV format into the RGB format. This is conceptually a pixel-by-pixel operation (in practice a quadlet by quadlet operation).

The size of the isochronous packets is 2560 bytes, which is an exact multiple of 4, that is 640 quadlets. The number of isochronous packets per frame is 240. The data coming from the camera amounts to 18 MB/sec.

This is an example of a relatively loaded system, because the application has to perform a lot of calculations and non-trivial data transfers. 18 Mb/sec have to be received from 1394, processed by the CPU and then retransmitted from 1394.

Doing a *fixed data receive*, with each receive request specifying [1 header quadlet, 640 payload quadlets, 240 packets], would make the processing of each frame a simple loop like:

```
for (Quad=0; Quad<640*240; Quad++)
    ConvertQuad(&DataBuffer[Quad]);
```

In order to retransmit the stream, the application would then issue a *data transmit* request that would specify as *header buffer* and *data buffer* the same data buffers that were used in receiving the frame.

Doing a *fixed packet receive*, with each receive request specifying [packet size 641 quadlets, 240 packets], we would have to use a nested loop like:

```
PacketPayload = &PacketBuffer[1];

for (Packet=0; Packet<240; Packet++)
{
    for (Quad=0; Quad<640; Quad++)
        ConvertQuad(&PacketPayload[Quad]);

    PacketPayload += 641;
}
```

In order to retransmit the stream, the application would then issue a *fixed packets transmit* request that would specify as *packet buffer* the same packet buffer that was used in receiving the frame.

Even if the application wanted to support this conversion at smaller image resolutions and fps speeds, the additional coding complexity in the second case is not that much, while there is the double overhead saved in the 1394-to-PCI interactions.

In this case an application designer could use the *packet receive* and *packet transmit* methods in order to lessen the overall system load.

³² This is from a real world application, not just a fictitious example.

Let us now suppose that the application needs to process the image in 4x4 pixel blocks and perform some internal calculations on the outcome. In this case using a *packet receive* coding would start getting a little more complex, because the position of each pixel is a not only a function of its position and the image size, but also a function of the fps (different fps use different number of packets per frame).

This could be possibly written with relatively simple-looking code using a macro like `PIXEL_POS(X,Y,Width, Height, fps)`, but firstly the code would have to perform more actual calculations to calculate the position of each pixel, and secondly it would not be as easy for the application developer to optimize this code or write it in assembly language because more parameters would have to be taken into consideration.

In this case, *data receive* would probably be preferable.

Processing Isochronous Requests

Outline of Isochronous Processing Loop

In general, the processing of an application that uses a constant-flowing isochronous stream can be described by the following pseudo-code:

```
C1394OpenAdapterChannel( ChannelHandle, hChannelEvent );
SetupIsochRequests( MyReqeustArray, APP_REQUESTS );
C1394IsochQueue( ChannelHandle, MyReqeustArray, APP_REQUESTS );

for ( ;; )
{
    WaitForSingleObject( hChannelEvent, INFINITE );

    while ( NULL != ( pRequest = C1394GetNextCompleteRequest( ChannelHandle )) )
    {
        ProcessRequest( pRequest );
        C1394IsochQueue( &pRequest, 1 );
    }
}
```

That is, the application uses **C1394IsochQueue** to queue an array of isochronous requests, so that it can keep the adapter channel busy while it is processing a complete request, and then proceeds into the processing loop.

In this loop it first waits for the adapter channel's event object to be signalled. This would signal that there is at least one isochronous request that has been completed.

Then the application goes into the inner processing loop, where it calls

C1394GetNextCompleteRequest, until this function returns NULL. At the point it knows it has to exit the inner loop, and once more wait for the adapter channel's event object to be signalled.

As you can see, the application need not keep track itself of which is the next isochronous request that was completed. **C1394GetNextCompleteRequest** returns a pointer to this request, in order to simplify the processing of the application.

Queueing Isochronous Requests

The following sample code displays the steps that an application would go through in order to prepare and queue a set of isochronous requests that will perform a *fixed packet receive*.

The application specifies the appropriate flags and buffer sizes so that each isochronous request is synchronized with the isochronous packet that has the *SyCode* field in its header equal to 1. Moreover the application specifies the **BR_START_NEXT** option which means that the 1394 stack should abort the currently executing request when a bus reset occurs, and continue with the next isochronous request.

```
// Setup the isoch receive requests.
for (I=0; I<RCV_COMMANDS; I++)
{
    RtlZeroMemory(&g_IsochRequest[I], sizeof(FIREAPI_ISOCH_REQUEST));

    g_IsochRequest[I].Tag = TAG_FIREAPI_ISOCH_REQUEST;
    g_IsochRequest[I].uOperationCode = ISOCH_OP_RCV_FIXED_PKTS;
    g_IsochRequest[I].fOptions = BR_START_NEXT;

    g_IsochRequest[I].RcvFixedPkts.Tag = TAG_ISOCH_RCV_FIXED_PKTS;
    g_IsochRequest[I].RcvFixedPkts.PacketBuffer = IsochRcvBuffer[I];
    g_IsochRequest[I].RcvFixedPkts.uBufferBytes = uBufferSize;
    g_IsochRequest[I].RcvFixedPkts.usMaxPayloadQuads = (USHORT) uPayloadQuads;
    g_IsochRequest[I].RcvFixedPkts.ChannelNumber = (UCHAR) uChannelNumber;

    // Sync with frame start
    g_IsochRequest[0].RcvFixedPkts.Flags = RCV_START_ON_SYCODE;
    g_IsochRequest[0].RcvFixedPkts.IsochSyCode = 1;

    pIsochRequestArray[I] = &g_IsochRequest[I];
}

Status1394 = C1394IsochQueue( AdapterHandle,
                             ChannelHandle,
                             pIsochRequestArray,
                             RCV_COMMANDS );

if (STATUS_1394_SUCCESS != Status1394)
{
    printf( "C1394IsochQueue FAILED with status %s.\n",
           C1394StatusString(Status1394) );
    exit(-1);
}
```

In this code fragment, the array of requests is supposed to have been globally declared (hence the 'g_' prefix) and are thus never out of scope until the application exits.

The *IsochRcvBuffer* variable is an array of pointers (either *void** or *UCHAR**) that has been earlier initialized with code like this:

```
for (I=0; I<RCV_COMMANDS; I++)
{
    IsochRcvBuffer[I] = malloc( uPacketsPerRequest*(uPayloadQuads+1)*4);

    if (NULL == IsochRcvBuffer[I])
    {
        printf("Memory Allocation of %u bytes FAILED.\n", (uPayloadQuads+1)*4);
        exit(-1);
    }
}
```

This means that each request will receive *uPacketsPerRequest* isochronous packets.

Applications should always check the return value of **C1394IsochQueue**. The most usual reason of failure for this function is an incorrect setup of the request structures, which will be indicated with a return code of **STATUS_1394_INVALID_PARAMETER**.

In debugged applications, **C1394IsochQueue** should not fail unless the system is so low in memory that an internal memory allocation fails.

Retrieving Complete Isochronous Requests

The code fragment below demonstrates the isochronous processing loop of an application. This is taken from the same sample as the code fragments in the previous example.

```

while ( uRqIndex < uTotalIsochRequests )
{
    WaitForSingleObject(ChannelStartProcessingEvent, INFINITE);

    while (NULL != (pIsochRequest = C1394GetNextCompleteRequest(ChannelHandle)))
    {
        if (STATUS_1394_SUCCESS == pIsochRequest->Status)
            ProcessRequest(pIsochRequest);
        else
            DisplayError(pIsochRequest);

        // Requeue this command if necessary.
        if (uTotalIsochRequests - uRqIndex > RCV_COMMANDS)
        {
            Status1394 = C1394IsochQueue( AdapterHandle,
                                         ChannelHandle,
                                         &pIsochRequest,
                                         1 );

            if (STATUS_1394_SUCCESS != Status1394)
            {
                printf( "Requeueing C1394IsochQueue FAILED with status %s.\n",
                       C1394StatusString(Status1394) );
                exit(-1);
            }
        }

        uRqIndex++;
    }
}

```

As you can see, the code that requeues an isochronous request structure need not change any of the fields in the structure, provided of course that the operation's characteristics (channel number, packet sizes, etc) have not changed.

With regards to error handling it is important to notice once again that under normal circumstances (valid handles, valid pointers, valid request parameters, available system memory, etc) the call to **C1394IsochQueue** will not fail, so developers need not worry about writing too sophisticated error handling code to handle this case.

What is more likely to happen, is an unsuccessful status on an isochronous request. For example an isochronous receive can fail for various reasons, depending on the options that the caller specifies. The sample code that queued the isochronous requests specified the **BR_START_NEXT** option which instructs the 1394 stack to abort the currently running isochronous request with the **STATUS_1394_BUS_RESET** status code when a bus reset occurs, and start the next isochronous request in the queue.

Other common reasons for isochronous request failure are that developers should check for during the development of their application are:

- FIFO underruns (for isochronous transmit).
- FIFO overruns (for isochronous receive)
- CRC check failures.

FIFO problems can occur if the system is very busy with other tasks (for example disk accesses), the isochronous packets are relatively³³ big and the adapter's FIFOs are relatively small. Such errors are usually indication that the PCI bus is very busy and as a result the 1394 adapter did not get access to the PCI bus in time.

³³ What *relatively big* means depends on the adapter.

On some adapters (PCILynx based) it is usually possible to overcome FIFO problems by adjusting the size of the isochronous and asynchronous FIFOs that the 1394 chip uses.

A bad CRC is usually an indication that the transmitter of the stream failed to transmit a packet correctly (for example due to a FIFO underrun). If an isochronous packet with a bad CRC is received then the current isochronous request is completed with the **STATUS_1394_CRC_ERROR** status code.

How many requests to queue?

A question that is often faced by application designers is how many isochronous requests to initially queue and from then on circulate by processing and requeuing.

The answer to this basically depends on several factors:

- How many isochronous packets comprise the *logical unit*³⁴ that the application processes?
- How long does it take the application to process its *logical unit*?
- What variations does the application wish to handle in the time it takes to process a frame?
- How critical is the data?
- What are the real-time requirements of the data?

For ease of understanding we will simply call the *logical unit* as *frame*. This however does not necessarily mean that we are talking about digital camera applications.

What should be obvious is that if the application needs more time to process a *frame*, than the amount of time between two successive *frames*, then the application will inevitably start losing *frames*, no matter how many isochronous requests it queues³⁵.

If the frame processing time is smaller than the inter-frame time then exactly 2 isochronous receive requests are enough to do the job.

Things get more complicated if the frame processing time is varying, either due to a varying system load or due to the nature of the image, and as a result it is sometime less than the inter-frame time and sometime greater.

An application *loses* a frame if the adapter channel is at some moment left idle, while there are isochronous packets transmitted on the bus for the specified channel number.

On the other hand an application may actively decide to *drop* a frame that it has successfully received, because there are newer data that it should process instead.

Depending on the nature of the application, the application designer decides whether it is better for the application to lose or drop frames.

Let us consider the example of a video display application that involves format conversion or frame decompression with software. Software decoders/decompressors are usually much slower than their hardware counterparts. Depending on the size of the frame, the level of code optimization in the software, and the frame rate at which the video is transmitted, the time required to decode/decompress a frame might be greater than the inter-frame time.

If the application designer decides to lose frames instead of dropping frames, then it means that the application queues N isochronous receive requests, each corresponding to a video frame. When one request gets completed the application it processes the request and when it's done it requeues it.

If the application designer decides to drop frames, then this means that the application queues N isochronous receive requests, each corresponding to a video frame. After displaying a frame the application retrieves ALL completed requests. If the application retrieved M>1 request, then it

³⁴ In the case of digital cameras, the logical unit is a video frame.

³⁵ What may not be evident to beginning 1394 developers is that the time between 2 logical frames is not necessarily the time it takes to transmit a frames. Many devices (for example 1394 digital cameras) that perform isochronous transmissions leave some idle cycles between their frames.

requeues the first (M-1) and keeps the last request (which is the newer frame). Then it proceeds to process and display this frame.

If the frame processing time is greater than the inter-frame time, then the frame rate achieved in either approach is practically independent of N!

In the *lost frame* approach, the greater N becomes, the greater the delay gets between the time a frame was sent by the camera and the time it was displayed on screen.

In the *drop frame* approach, the greater N becomes, the smaller the delay is between the time a frame was sent by the camera and the time it was displayed on screen. This delay at some point reaches a lower bound.

However in both cases, the visual impression (the achieved frame rate) is the same. Indeed the *drop frame* approach might even be worse as N is increased, because the system spends a lot of time receiving data that it then discards. Since the system is CPU bound, receiving and discarding data slows down the frame processing, thus making the overall achieved frame rate worse.

If the processing time of each frame varies, below and above the inter-frame time then the *lost frame* approach can give some undesirable visual results (the picture is continuously left behind in time, sometimes there are short fast-motion bursts, etc), but if N is small (<5) it can in general provide a good impression without unnecessarily loading the system by receiving and discarding data.

If the processing time of each frame is smaller than the inter-frame time, then N should be 2 in which case there is no difference between the two approaches.

The above however are true if we are talking about real-time video display. If our application has to do with a monitoring and control application that monitors measurement data from devices, then the application requirements might change. In this case it might not be desirable at all to drop any data, because this way the system processes as many data values as possible.

In any case the application designer should carefully analyze the requirements of the target system and design the appropriate strategy. For example in some cases of varying system load, it could make sense that the system utilizes a modified *drop* strategy in which the application will drop the oldest frame if the number of retrieved frames exceeds 3. This means that the application will drop a frame only if it finds out that it starts to stay behind in its processing of incoming data.

Queueing 'Small' Requests

The isochronous cycle is 125 microsec, which means that more or less one isochronous packet appears on the bus every 125 microsec (or 0.125 msec). If an application queues isochronous requests that involve very few packets, then these request will complete very fast.

For example an isochronous receive request for 20 isochronous packets will complete at the 1394 level in 2.5 msec. This is a very small amount of time and as a result there are some special considerations that should be taken into account.

In order for an application to be notified about isochronous request completion, an interrupt has to be raised, an interrupt handler routine has to be executed in order to acknowledge the interrupt and schedule a DPC³⁶, the DPC has to be scheduled, an event object has to be signalled, and the respective user mode thread that is waiting on the object has to be scheduled for execution.

Depending on the load of the system, the amount of time between the actual completion of a request on the hardware level, and the moment the application gets the chance to execute can be comparable to the amount of time required for 1394 to complete more than one isochronous requests.

Even though the actual CPU processing that the application performs for each isochronous request might be much faster, the application will stay behind. If the application has queued a small number of such isochronous requests, then there will be small periods of time where the adapter channel would

³⁶ DPC stands for Deferred Procedure Call, a high priority callback mechanism in NT.

stay inactive, because it has completed executing all queued requests before the application had the chance to process and request some.

An application that is doing isochronous receive will start losing packets. The application will think that it is missing some isochronous packets.

An application that is doing isochronous transmit will leave idle cycles on the bus without intending to do so. As a result it will appear to it that its isochronous operations are slower than it should be (for example it would take the application more than 1000 msec to send 8125 isochronous packets).

What such an application of this sort must do should be clear by now. If the application wants to queue isochronous requests for very few isochronous packets each, then it should queue a sufficient number of isochronous requests so that the adapter channel never stays idle.

Isochronous Options

FireAPI offers a variety of options that can help an application recover from problematic situations with regards to isochronous operation.

The first major option is the timing of isochronous requests. What should happen to an isochronous receive request, if the transmitter suddenly stops transmitting the isochronous stream? The isochronous request will never complete. The same thing will happen if for any reason the cycle master stops sending cycle start packets. All isochronous talkers will stop transmitting their streams³⁷.

An application can specify a timeout value in msec for each isochronous request. If the request has not completed within the specified amount of time, the 1394 stack will abort it with

STATUS_1394_TIMEOUT and then continue the operation of the adapter channel in accordance with the flags that the caller has specified.

The application can specify that the adapter channel should continue with the next request, or that it should cancel all remaining queued requests.

This kind of timing provided by the 1394 stack relieves the application developer from having to write complex code in order to handle such an event.

The second major set of options has to do with handling a bus reset that occurs during an isochronous request. The application has the option of specifying that the current request is restarted, or the current request gets aborted and the next one is started, or all queued isochronous requests get aborted.

This way the application can have full control of what will happen with its isochronous requests when a bus reset occurs. For example an application could specify the **BR_FLUSH_QUEUE** flag so that when a bus reset occurs, all queued requests get immediately cancelled. The application will then empty the completed request queue by calling **C1394GetNextCompleteRequest**, then possibly proceed to perform isochronous resource reallocation, possibly update the request structures and enable a different channel number for the adapter channel, and finally requeue the requests with **C1394IsochQueue**.

Another important set of options has to do with controlling the number of isochronous packets that will be transmitted in each isochronous cycle.

Finally the application has the flexibility to cancel one or more, or all its isochronous requests at any point. This can be achieved with the **C1394IsochCancel** function. This makes it very easy for an application to change its mode of processing at any time.

For example if an application that performs an isochronous receive wants to change channel number, then it can call **C1394IsochCancel** with the *IsochCancelAll* flag, empty the completed request queue using **C1394GetNextCompleteRequest**, if not already enabled then enable the new channel number using **C1394SetInformation**, update the isochronous request structures and requeue them again with **C1394IsochQueue**.

³⁷ Some 'buggy' devices initiate bus resets if they have pending outgoing isochronous traffic and there is no cycle start on the bus.

For more specific details on the available options, see the documentation of the **FIREAPI_ISOCH_REQUEST** structure and the isochronous request types later in this document.

Common Mistakes in Isochronous Processing

This section lists common mistakes that are made by developers who write code to handle isochronous operations. All the traps mentioned below are described in the documentation of the respective functions, but often developers cannot fully understand the consequences of some statements unless they see what those statements mean in practice.

C1394GetNextCompleteRequest returns completion status to user mode

An application has to call **C1394GetNextCompleteRequest** in all cases, because it is the call that returns status information about the completion of the isochronous operation to user mode.

Consider an application that wants to grab a single frame from a camera. The following code would not work correctly:

```
C1394IsochQueue( ChannelHandle, &MyRegeust, 1 );
WaitForSingleObject( hChannelEvent, INFINITE );

if (STATUS_1394_SUCCESS == MyRequest.Status)
{
    /* Do something */
}
```

The value of `pRequest->Status` and other important fields inside the request structure are not updated until **C1394GetNextCompleteRequest** is called. When **C1394IsochQueue** is called, the class driver stores **STATUS_1394_PENDING** into the *Status* field of an isochronous request, and this is the value that the application will ‘see’ in the *if* statement in the code above.

Another instance of this mistake is the following:

```
C1394IsochQueue( ChannelHandle, &MyRegeust, 1 );

//Do some other processing until the command completes.
while (STATUS_1394_SUCCESS != MyRequest.Status)
{
    MyProcessing();
}

// Check out the results of the isochronous operation.
...
```

The *Status* field will never be updated, and the *while* loop will execute for ever.

If an application wants to perform a loop of the kind intended above, then it has two options:

```
C1394IsochQueue( ChannelHandle, &MyRegeust, 1 );

//Do some other processing until the command completes.
while (NULL == C1394GetNextCompleteRequest(ChannelHandle))
{
    MyProcessing();
}

// Check out the results of the isochronous operation.
...
```

-OR-

```

C1394IsochQueue( ChannelHandle, &MyReqeust, 1 );

//Do some other processing until the command completes.
while (WAIT_TIMEOUT == WaitForSingleObject(ChannelEvent, 0))
{
    MyProcessing();
}

// We have to do this, remember?
C1394GetNextCompleteRequest(ChannelHandle);

// Check out the results of the isochronous operation.
...

```

Isochronous Requests execute in the background

This means that request variables must be available throughout the *lifetime* of the operation, or unpredictable behaviour will follow for the application. This means that an application can only use local variables for the isochronous request structures that it passes to **C1394IsochQueue**, if these local variables will stay *alive* until the isochronous requests are completed.

The following pseudo code demonstrates this mistake.

```

void MyQueueRequest1( C1394_CHANNEL_HANDLE a_ChannelHandle )
{
    FIREAPI_ISOCH_REQUEST IsochRequest;

    // Initialize the isoch request structure.
    // ...

    // Queue it for execution.
    C1394IsochQueue( a_ChannelHandle, &IsochRequest, 1 );
}

```

When the function returns, the `IsochRequest` variable is not available any more, and will be overwritten by the stack of other function calls that the calling code of `MyQueueRequest` will make.

Rather than that, such code should be written as:

```

void MyQueueRequest2( C1394_CHANNEL_HANDLE a_ChannelHandle )
{
    PFIREAPI_ISOCH_REQUEST pIsochRequest;

    pIsochRequest = malloc(sizeof(FIREAPI_ISOCH_REQUEST));

    // Initialize the isoch request structure.
    // ...

    // Queue it for execution.
    C1394IsochQueue( a_ChannelHandle, &pIsochRequest, 1 );
}

```

Of course the same kind restriction applies to the memory that will be used as the packet/data buffer in the isochronous operation. This memory must stay valid until the isochronous operation completes.

C1394IsochQueue accepts an array of pointers to isochronous request structures

This means that **C1394IsochQueue** accepts a pointer to a pointer to a **FIREAPI_ISOCH_REQUEST** structure.

In the previous sample, `MyQueueRequest1` is mistaken on this aspect as well, because it passes to **C1394IsochQueue** the address of a **FIREAPI_ISOCH_REQUEST** structure.

`MyQueueRequest1` is correct and demonstrates how to call **C1394IsochQueue** if you want to queue one request structure and you have got a pointer to it.

The code below demonstrates how should an application proceed to queue an array of isochronous request structures.

```
// Global Request Structures
FIREAPI_ISOCH_REQUEST MyRequestArray[APP_REQUESTS];

void QueueMyRequests(C1394_CHANNEL_HANDLE a_ChannelHandle)
{
    PFIREAPI_ISOCH_REQUEST PointerArray[APP_REQUESTS];
    ULONG I;

    for (I=0; I<APP_REQUESTS; I++)
        PointerArray[I] = &MyRequestArray[I];

    C1394IsochQueue(a_ChannelHandle, PointerArray, APP_REQUESTS);
}
```

The pointers are copied inside the 1394 stack data structures, and the array itself can be on the caller's stack and get out of scope as soon as **C1394IsochQueue** returns, exactly as shown on the code fragment above.

Remember to empty the request queue when cancelling all pending isochronous requests

An application can cancel at any moment all its pending isochronous requests by calling **C1394IsochCancel**. As a result all the pending requests of the adapter channel get immediately completed with the status code **STATUS_1394_ABORTED**.

There requests as well are stored into the adapter channel's completed request queue, and the application must call **C1394GetNextCompleteRequest** before it queues any new requests in this channel.

The steps that an application should take when it cancels all its pending requests are demonstrated in the code fragment below:

```
C1394IsochCancel( ChannelHandle, IsochCancelAll );

// Empty the completed-request queue.
while (NULL != (pRequest = C1394GetNextCompleteRequest( ChannelHandle )))
{
    /* Do something with pRequest */
}
```

Forgetting to empty the completed request queue is a mistake that can cause very *funny* behaviour to the application if it continues to use the adapter channel:

- If the application frees the request structures, then the adapter channel's complete request queue will contain pointers to memory that has been freed. When the application calls **C1394GetCompleteRequest**, the implementation of this function in **UBUMAPI.SYS** will try to access this memory, and read/write status information to it.
 - If this memory has been invalidated and is not accesible, then an access violation will occur in kernel mode and **STATUS_1394_DRIVER_INTERNAL_ERROR** will be returned.
 - If this memory is still accesible, but has been overwritten, then **UBUMAPI.SYS** will not identify the pointer to the **FIREAPI_ISOCH_REQUEST** structure as valid, and will again return **STATUS_1394_DRIVER_INTERNAL_ERROR** because it expects that **UB1394.DLL** has given it a valid pointer.

- Worse of all, if the memory is still accessible and has not been overwritten, **UBUMAPI.SYS** will think the memory is still valid and write back status information to it. This can cause the application to fail in completely unrelated pieces of code.
- If the application had used local variables for the request structures, and these variables are not any more in scope, then inside the adapter channel's complete request queue there will be pointers to some place on the application's stack. Unfortunately, the same scenarios as above apply.
- If the application has used global variables for the request structures, or local variables that are still in scope, then sooner or later the application's complete request queue will contain one or more pointers two times. This can as well lead to wildly unpredictable behaviour.

You might wait again only after C1394GetNextCompleteRequest returns NULL

It has to be emphasized, is that an application should not wait again on the adapter channel's event object until **C1394GetNextCompleteRequest** returns NULL.

The following example is an actual bug that occurred during application development:

```

HandleArray[0] = hChannelEvent;
HandleArray[1] = RestartEvent;

C1394IsochQueue( ChannelHandle, MyReqeustArray, APP_REQUESTS );

for (;;)
{
    WaitStatus = WaitForMultipleObjects( 2, HandleArray, INFINITE );

    if (WAIT_OBJECT_0 == WaitStatus)
    {
        // There is a complete request.
        while (NULL != (pRequest = C1394GetNextCompleteRequest( ChannelHandle )))
        {
            ProcessRequest( pRequest);
            C1394IsochQueue( &pRequest, 1);
        }
    }
    else if ((WAIT_OBJECT_0+1) == WaitStatus)
    {
        // The user required us to restart in a new channel number.
        // Cancel all our requests, update them and requeue them.
        C1394IsochCancel( ChannelHandle, IsochCancelAll );

        // Empty the completed-request queue.
        for (I=0; I<APP_REQUESTS; I++)
            C1394GetNextCompleteRequest( ChannelHandle );

        UpdateRequests( MyRequestArray, APP_REQUESTS );
        C1394IsochQueue( ChannelHandle, MyReqeustArray, APP_REQUESTS );
    }
}

```

The mistake in this code is that when the application empties the adapter channel's completed request queue, it did not make the call to **C1394GetNextCompleteRequest** that returns NULL. This means that the 1394 stack did not reset the adapter channel's *Busy Flag* to zero, and thus will not signal the adapter channel's event object when the next isochronous request gets completed.

This in turn means that the **WaitForMultipleObjects** call will never return (WAIT_OBJECT_0 + 1).

Instead of the for-loop the application should have used the following code:

```

// Empty the completed-request queue.
while (NULL != C1394GetNextCompleteRequest( ChannelHandle ))
    /* Do Nothing */;

```

FIREAPI_ISOCH_REQUEST.CompletionEventHandle is for special purposes only

Many programmers confuse the purpose of the *CompletionEventHandle* field of the **FIREAPI_ISOCH_REQUEST** structure. This field is used as a complement to the event object associated with the adapter channel.

An example of a possible use is the following: the application queues 10 isochronous requests and wants to know when ALL 10 are completed. The event object associated with the adapter channel will get signalled when the first request is completed. How can the application find out when all have completed?

This is possible by associating another event object with the 10th isochronous request, and check the status of this object. This is done by storing the handle of this object in the *CompletionEventHandle* field of the 10th **FIREAPI_ISOCH_REQUEST** structure, and setting the **COMPLETE_SET_EVENT** flag in the *Flags* field of the same structure.

When this event gets signalled the application knows that all 10 requests have completed, so it has to call **C1394GetNextCompleteRequest** to remove them from the queue as explained in the previous paragraphs.

Isynchronous Operation Limits

In the current implementation of ubCore each isochronous request is programmed to the adapter as a single DMA operation, meaning that the physical addresses of all pages in the data buffer are obtained and given to the adapter so that the actual DMA transfers can occur.

This works without any limitation on 32-bit systems. The isochronous buffer in an isochronous operation can be 5 or 10MB big without any problems. Actually there is a limit set by the operating system, but on 32-bit systems this is close to 2GB, so almost any reasonable buffer size will work.

But with the advent of 64-bit systems things have changed. The operating system now, depending on the hardware platform and the amount of available memory, puts a limit that is much more restricting and is usually 1MB.

This means that on those systems you can't queue isochronous requests that are bigger than 1MB in size. You will have to break your original request in multiple smaller requests. It is suggested that you break the requests in $((DMA_LIMIT/2)-4KB)$ bytes so that two requests can be programmed to the adapter at a time. So when the first request completes, the adapter will be processing the second one while the driver software will be preparing the third one, and so on. This way you don't run the risk of losing isochronous packets.

This method of breaking bigger logical requests into smaller isochronous requests has been implemented in both Firei.DLL, in the FireiAPI ubCore interface, and in the ubCore DirectShow driver (UBDCAM.SYS) and works without any problems.

Isynchronous Resource Allocation

Isynchronous Timing

Isynchronous operation on the physical level is clocked by the so called *Cycle Start Packets*. This is a special kind of packet that is transmitted by the *Cycle Master* of the 1394 bus every 125 microsec. This period of 125 microsec is also known as the *Isynchronous Cycle*. There are 8000 *isynchronous cycles* per second.

At the hardware level, once a DMA context of the 1394 adapter transmits an isynchronous packet then it will not attempt to transmit the next isynchronous packet until the next *cycle start packet* is received³⁸. This means that if the *cycle master* stops transmitting *cycle start packets*, then all isynchronous traffic will stop.

Only the root can be the *Cycle Master*, so the root must be a *Cycle Master Capable* node. All FireAPI nodes are *cycle master capable*. Unibrain's Serial Bus Manager driver (**UBSBM.SYS**) examines the bus after each bus reset and makes certain that the root node is a *cycle master capable* node. If the root is not *cycle master capable*, then **UBSBM.SYS** will designate a *cycle master capable* node as the root (by sending a Configuration PHY packet) and then initiate a bus reset³⁹. In this way **UBSBM.SYS** guarantees that the bus itself will be *isynchronous capable*.

You can observe this behaviour by using the **CMD1394** utility in order to forceroot a device that is not cycle master capable, for example a 1394 digital camera. Open the **BUSVIEW** utility to see the topology of the bus, and then initiate a bus reset from **CMD1394** with the **BR** command. You will see in **BUSVIEW** that the camera has become the root node. In about 2-3 seconds **UBSBM.SYS** will initiate another bus reset and the root node will change.

The Protocol

The 1394 specification describes a 'protocol' that applications should use in order to ensure proper isynchronous operation of the 1394 bus.

The central entity in this protocol is the Isynchronous Resource Manager (IRM), which is one of the bus nodes. The IRM is not exactly a 'manager' like the Serial Bus Manager (SBM), but rather acts like a central, well-known location where information about available isynchronous resources is maintained.

There are two isynchronous resources:

- Isynchronous Channel Numbers (CHANNELS_AVAILABLE 64-bit register)
- Isynchronous Bandwidth Units (BANDWIDTH_AVAILABLE 32-bit register)

Each bit of the CHANNELS_AVAILABLE register corresponds to an isynchronous channel number. If the bit is set then the channel is available.

Isynchronous Bandwidth Units (also referred to as Bandwidth Allocation Units) are defined as the time required to send one quadlet of data at the S1600 data rate, roughly 20 nanoseconds.

Since S1600 is theoretically defined as being 4 times as fast as S400, then 1 bandwidth allocation unit at S400 practically corresponds to the time taken to transmit 1 byte. In S200 it takes 2 bandwidth units to transmit a byte, while in S100 four bandwidth units are required.

³⁸ In cases of multi-channel transmissions it is possible to transmit 2 or more packets per cycle if they belong to different channel numbers, but the driving software will have to specially program the chip for this operation.

³⁹ The UBSBM.SYS on the node that was elected as Serial Bus Manager will be the one to perform these actions.

In order to calculate the bandwidth units required to transmit an isochronous packet of a given size, the transmission speed, the packet header, the header and data CRCs, possibly the payload padding and the arbitration time should also be taken into consideration.

FireAPI provides the **BANDWIDTH_UNITS** macro so that applications can simply specify the payload size in bytes and the transmission speed and get back the required bandwidth allocation units.

The ‘prototype’ of this macro is defined as shown below:

```

ULONG BANDWIDTH_UNITS(
    IN ULONG                uPayloadBytes,
    IN C1394_SPEED_CODE    TransmissionSpeed
);

```

It is the responsibility of the IRM to implement the CHANNELS_AVAILABLE and BANDWIDTH_AVAILABLE registers according to the requirements of the 1394 specifications, and it is the responsibility of applications to allocate isochronous resources from the IRM before they start transmitting on the bus⁴⁰. Resource allocations are performed with compare-swap lock transactions.

However, it must be noted that this protocol is simply a convention. The isochronous resource allocations are performed on a ‘logical’ level, not in the physical level. This means that nothing can physically prevent an adapter from transmitting on a channel number that is already in use, or transmit without having allocated bandwidth first.

Also the same kind of convention applies to the total amount of bandwidth that can be used for isochronous traffic. The 1394 specification mandates that the maximum isochronous traffic permitted is about the 80% of an isochronous cycle (~100 out of 125 microsec), so as to leave some free time for asynchronous traffic as well. This corresponds to 4915 bandwidth units, which is the initial value of the BANDWIDTH_AVAILABLE register.

If some application needs to violate this limit then this is certainly possible. However this should only be performed in dedicated environments, where the designer knows in advance the exact operating conditions of the system. Such a violation should be avoided in general purpose applications.

Similarly it is equally feasible to violate the isochronous cycle duration, for example have 4 isochronous channels transmitting 2KB blocks each.

However this has much more serious side-effects:

- The *cycle master* will **repeatedly** not be able to transmit the *cycle start packet* in time and this will result in total loss of isochronous timing. Simply put, an unknown number of cycle start packets will be transmitted per second; however certainly less than 8000. Since isochronous timing is lost, then there is no meaning in calling these transfers *isochronous*.
- In some cases it is possible for the 1394 chip to stop functioning (hang).

Note that the word *repeatedly* above is emphasized. An isochronous cycle will be violated every now and then on a bus that has modest traffic. Most chips (if not all) are not sophisticated enough to be able to calculate in real time whether the transmission of their next asynchronous packet will cause a cycle start packet to be delayed.

So it often happens that an asynchronous packet might cause a cycle start packet to be delayed.

However the 1394 chips have logic that allows them to compensate for this situation by sending the next cycle start packet sooner than 125 microsec so that the overall isochronous timing is maintained.

So you only need to worry about extensive and repetitive violations. These are the ones that can cause trouble to the isochronous timing.

⁴⁰ Sometimes the application does not transmit itself, but controls a device that transmits isochronously. Such devices usually expect that their controlling application will perform all the necessary resource allocations before starting the device.

Bus Reset & Isochronous Resources

When a bus reset occurs, the IRM frees all isochronous resources and a new IRM might be elected if the bus topology has changed. This means that applications should reidentify the IRM and reallocate their isochronous resources after each bus reset. If an application fails to reallocate its resources then it should stop its isochronous transmit operations.

The 1394 specification requires that applications perform all their isochronous resource reallocations within 1 second after the bus reset. Only after 1 second has elapsed should an application attempt to allocate new isochronous resources.

Identifying the IRM

The 1394 specification specifies that the IRM is the node with the greatest physical ID whose SelfID packet has both the *LinkOn* and the *Contender* bits set.

In order for an application to allocate isochronous resources, it must identify the IRM. Although an application could use the official definition in order to identify the IRM, FireAPI provides the **C1394GetIRMNodeID** function for this purpose.

This function is prototyped as:

```
C1394_NODE_ID C1394GetIRMNodeID(
    IN C1394_ADAPTER_HANDLE AdapterHandle,
    IN C1394_BUS_ID BusID
);
```

For the time being applications should set the *BusID* parameter to **LOCAL_1394_BUS_ID** which is defined as 1023₁₀ (3FF₁₆).

Allocating a channel number using compare swap

The sample code below displays a way that an application could use to allocate a random channel number between 0 and 30 from the IRM.

A couple of details should be kept in mind when allocating channel numbers:

- According to P1394a 2.1, channel number 31 should only be used by the Broadcast Channel Manager (BCM).
- According to the 1394 specification, the CHANNELS_AVAILABLE register only supports quadlet (32-bit) reads, but it is not clear whether it supports both 32-bit and 64-bit compare swap or only the 32-bit. Software implementations of this register should always support the 32-bit compare swap, but not necessarily the 64-bit compare swap. FireAPI's CHANNELS_AVAILABLE implementation supports 64-bit compare swaps as well. It is suggested that applications that want to allocate channels from the IRM attempt to do so using 32-bit locks.
- The 1394 specification uses big endian and IEEE numbering of bits. This means that channels 0-31 are in the CHANNELS_AVAILABLE_HI register (offset CSR_CHANNELS_AVAILABLE) and channels 32-63 in the CHANNELS_AVAILABLE_LO register (offset CSR_CHANNELS_AVAILABLE+4). Moreover bits are numbered left-to-right. This means bit 0 is the leftmost bit of CHANNELS_AVAILABLE_HI (big endian 0x8000000), and bit 31 is the rightmost bit (big endian 0x0000001).
- The register contains bits set to 1 for available channel numbers. When you allocate a channel number you have to clear the respective bit. If you try to check whether a channel number is allocated, then you have to check for a zero bit. Although this sounds obvious, it is a common source of errors.

```

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <FireAPI.h>

main(void)
{
    C1394_ADAPTER_HANDLE  C1394AdapterHandle;
    C1394_NODE_ID          IRMNodeID;
    STATUS_1394            Status1394;
    ULONG                  uChAvailNewValue, uChAvailOldValue;
    ULONG                  uChannelNumber, uArgValue;

    srand(GetTickCount());

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

    // Let us find out who the IRM is.
    IRMNodeID = C1394GetIRMNodeID( C1394AdapterHandle, LOCAL_1394_BUS_ID );

    // Pick up a random channel number in 0-30 (31 is for use by the BCM).
    uChannelNumber = rand() % 31;

    // This is the last known value that we suppose the register has.
    uArgValue = 0xFFFFFFFF;

    // Set up the channel number we want to allocate.
    uChAvailNewValue = uArgValue & ~(((ULONG)1) << (31-uChannelNumber));

    for (;;)
    {
        Status1394 = C1394CompareSwapNode( C1394AdapterHandle,
                                           IRMNodeID,
                                           CSR_CHANNELS_AVAILABLE,
                                           Lock32,
                                           (ULONGLONG) uArgValue,
                                           (ULONGLONG) uChAvailNewValue,
                                           &uChAvailOldValue );

        if (STATUS_1394_SUCCESS == Status1394)
        {
            printf( "Allocated channel number %u (ChAvailHi is %0#10X).\n",
                   uChannelNumber,
                   uChAvailNewValue );

            break;
        }
        else if (STATUS_1394_LOCK_FAILED == Status1394)
        {
            // Was it because we did not specify a good arg_value, or
            // because the channel number is busy?
            if ( 0 == (((ULONG)1) << (31-uChannelNumber)) & uChAvailOldValue )
            {
                printf("Channel Number %u is NOT free.\n", uChannelNumber);
                break;
            }
            else if (uArgValue != uChAvailOldValue)
            {
                printf( "Channel Number %u is FREE but arg_value is not valid.\n",
                       uChannelNumber );

                uArgValue = uChAvailOldValue;
                uChAvailNewValue =
                    uChAvailOldValue & ~(((ULONG)1) << (31-uChannelNumber));
            }
            else
                printf("This should never happen.\n");
        }
        else
        {
            printf( "Return status is %s\n", C1394StatusString(Status1394));
            break;
        }
    }

    C1394CloseAdapter( C1394AdapterHandle );
    C1394Terminate();
    return 0;
}

```

The sample does not read the value of the CHANNELS_AVAILABLE register before attempting to allocate the channel that it wants, but directly attempts the lock.

There are two reasons for that:

- A failed lock also acts like a read operation because the old value of the register is returned. This way if the application is 'lucky' enough it can succeed with one transaction instead of two (a read then a lock), thus minimizing network traffic.
- If the handling of lock requests for CHANNELS_AVAILABLE is implemented according to paragraph 9.28 of P1394a 2.1, then if the channel number(s) are available, a lock with *arg_value* set to 0xFFFFFFFF and the *data_value* as appropriate, will **always** succeed. FireAPI is the first software implementation of this algorithm, indeed the algorithm itself was proposed by Unibrain and adopted by P1394a 2.1.

The sample above treats the CHANNELS_AVAILABLE register as if it was in the CPU's native format. If the program read the value of CHANNELS_AVAILABLE in order to initialize *uArgValue* with the current value of the register instead of 0xFFFFFFFF, then the application would have to byte swap the results of this read before continuing.

This is demonstrated in the code fragment below, that would replace the assignment of 0xFFFFFFFF to *uArgValue*:

```
// Read the current value of the register.
Status1394 = C1394ReadNode( C1394AdapterHandle,
                           IRMNodeID,
                           CSR_CHANNELS_AVAILABLE,
                           4,
                           &uArgValue,
                           NULL,
                           NULL );

if (STATUS_1394_SUCCESS != Status1394)
{
    printf( "C1394ReadNode failed with status %s\n",
           C1394StatusString(Status1394) );
    return -1;
}

SWAP_ENDIAN_32(uArgValue);
```

In this case the programmer knows that the program is executing on a little-endian platform, so after reading the contents of CHANNELS_AVAILABLE into a ULONG variable it has to byte swap the contents of this variable before proceeding to work with it.

FireAPI offers two ways for byte swapping a quadlet: the **SwapEndian32** inline function and the **SWAP_ENDIAN_32** macro, which are both implemented with inline assembly instructions. See the section on **Endianness Swapping** for more information.

Allocating bandwidth using compare swap

Similar considerations apply as in the case of allocating a channel number. The application can directly attempt to do a compare swap, using 4915₁₀ as the *old_value*.

If the application attempts to read the BANDWIDTH_AVAILABLE register first, then it should keep in mind that the quadlet it will read will be in big endian.

Freeing Isochronous Resources

When freeing an isochronous resource, typically the application must read the current value of the target register from the IRM, keeping in mind that it reads big endian data, then add its own resources to that value and attempt a compare swap. If the compare swap operation fails then the application should add its resource to the *old_value* returned⁴¹ and retry the compare swap.

⁴¹ This is always presented to the application in its native endianness.

VersaPHY Operations

VersaPHY Basics

This section of the documentation gives a brief overview of the VersaPHY standard. For the full details and technical explanations please refer to the *VersaPHY Additions to IEEE1394 1.0.pdf* document included in the FireAPI setup.

VersaPHY is a new standard technology in the IEEE1394 family of standards that is meant to support the implementation of low cost devices, devices that don't require a LINK chip (and usually an associated IP core) to function.

VersaPHY devices do not perform the standard asynchronous transactions of IEEE1394 but instead implement a new set of read and write transactions that are based on PHY packets (8-byte packets). In the PHY packets used by the VersaPHY protocol the second quadlet is not the complement of the first quadlet, thus all the 8 bytes are available for use.

The assignment of a physical ID (PhyID) to every node on the 1394 bus is one of the core elements in the operation of the 1394 bus. Like all 1394 devices, VersaPHY devices also have a physical ID. Physical IDs are of course temporary and can change after each bus reset. However a fundamental departure that VersaPHY brings, is that VersaPHY devices can be assigned a permanent label, known as VersaPHY label (or VPLabel for short), that is sticky across bus resets and possibly across power cycles as well.

Thus VersaPHY devices have two addressing modes:

- PhyID based addressing → PhyID transactions.
- VPLabel based addressing → VPLabel transactions.

The basic operational model is that controller devices use PhyID transactions to discover (enumerate) the VersaPHY devices, then assign them a VPLabel and from that point on communicate with them solely using VPLabel transactions. Some VersaPHY devices may come with a preconfigured (static) VPLabel, so they don't require the discovery & setup phases.

A core point to the operation of VersaPHY is that all VersaPHY traffic is broadcast. Thus all nodes can receive all the VersaPHY traffic on the bus and talk or listen to any device.

Since VersaPHY traffic is broadcast it is possible to use it as a notification-from-the-device mechanism. When a device has *something* to tell the world about, it just sends an *unsolicited response packet* and all nodes may receive and process this packet and thus get notified by the device.

VersaPHY Functions & Profiles

A VersaPHY device, like all 1394 devices, may implement one or more independent logical functions, which are not surprisingly called *VersaPHY functions*. A VersaPHY function is basically a set of registers that the device implements, that give access to some specific functionality. Most devices implement just one VersaPHY functions, like most 1394 devices that have a single unit in their configuration ROMs.

A device vendor may define its own VersaPHY functions (and resulting register sets), but some of the well known behaviors have already been designed as register sets (GPIO, I2C, etc) and the resulting definitions are called VersaPHY profiles.

So, saying that a VersaPHY device implements the GPIO profile is equivalent to saying that it implements a VersaPHY function with the standard register set of GPIO behavior.

VersaPHY Transactions

The behavior of VersaPHY transactions is not as strictly defined as that of 1394 transactions for two reasons:

- Differences in implementation between 8-bit and 16-bit devices may cause an 8-bit device to respond differently to the same PhyID-based transaction than a 16-bit device would.
- The transaction response behavior is heavily “*profile-dependent*” which means that depending on the profile being implemented a read transaction may result in a different number of response packets, depending on the register addressed and the data passed.

This means that the 1394 Class Driver has no certain way of knowing how many response packets to expect for each VersaPHY transaction. However the application is supposed to know, since the application knows what type of device it is talking to.

Thus, we have another departure from the standard 1394 transaction request functions: *All the VersaPHY transaction request functions provide as arguments the number of expected response packets and a storage buffer where they should be put.*

VersaPHY API Overview

The API provided by Unibrain implements the minimum required functionality so that higher level modules can perform all the VersaPHY related activities like device discovery, label space annexing, read/write transactions, etc.

The Unibrain API is logically split in two independent portions:

- PhyID based access.
- VersaPHY Label based access.

VersaPHY PhyID functions

The PhyID functions permit software to enumerate the devices connected to the bus on any 1394 adapter and discover the VersaPHY functions implemented by them.

The functions provided are:

- **C1394VPReadNode**
- **C1394VPWriteNode**
- **C1394VPSendPacket**

VersaPHY Label functions

Each VersaPHY label is represented as an object to which a handle must be opened. Any application can open a handle to *any* VersaPHY label, at *any* time. VLabel handles are not exclusive access.

This means that:

- Two or more applications on the same PC can have open handles to the same device. The applications should act in a coordinated manner so as not to *confuse* each other and the device. A usual scenario could be having one application control several devices while a different application is monitoring the same devices.
- The 1394 Class Driver does not check at VLabel-open time whether a device with that VLabel exists on the bus or not. It is the application’s responsibility to check whether the VLabel used actually corresponds to an existing device.

In essence, each VLabel is treated by the 1394 Class Driver as a *channel* to which the application can *tune-in* and then receive or send traffic through.

The VersaPHY label functions provided are:

- **C1394VPChannelOpen**
- **C1394VPChannelClose**
- **C1394VPChannelGetNextPacket**
- **C1394VPChannelRead**
- **C1394VPChannelWrite**

VersaPHY Packet Structures

All 1394 packet structures are defined using big-endian memory layout. On the other hand PC architecture is little-endian, which sometimes causes trouble when a packet structure has to be mapped to a C structure definition that will be used in a little-endian programming environment. The goal is to define a C structure in such a way that it accurately reflects the big-endian memory layout of the packet, so that the programmer can manipulate the packet without first byte-swapping it.

This was a problem in the case of VersaPHY packet structures, because the VPLabel transaction request/response packets cannot be *nicely* mapped in C structures for a little-endian programming environment. Here *nicely* refers to the ability to map every field without breaking them apart in two pieces. However, when there are fields crossing 8-bit boundaries in the big-endian memory layout it is impossible to produce a proper mapping.

In these cases a little endian structure is defined and the packet has to be byte-swapped. In the case of VersaPHY packets we are talking about 8-byte (2 quad) packets so the byte-swapping operation involves no major CPU processing costs.

The table below lists all the defined VersaPHY packet structures and whether the respective packets need swapping. Since all 1394 structures used in FireAPI so far correspond exactly to the big endian layout, the suffix `_LITTLE` has been added to those structures that are defined as little endian.

Packet Structure	Packet Needs Swapping
C1394_VERSAPHY_PACKET_PHYID_READ_REQUEST	NO
C1394_VERSAPHY_PACKET_PHYID_READ_RESPONSE	NO
C1394_VERSAPHY_PACKET_PHYID_WRITE_REQUEST	NO
C1394_VERSAPHY_PACKET_PHYID_WRITE_RESPONSE	NO
C1394_VERSAPHY_PACKET_VPLABEL_READ_REQUEST_LITTLE	YES
C1394_VERSAPHY_PACKET_VPLABEL_READ_RESPONSE_LITTLE	YES
C1394_VERSAPHY_PACKET_VPLABEL_WRITE_REQUEST_LITTLE	YES
C1394_VERSAPHY_PACKET_VPLABEL_WRITE_RESPONSE_LITTLE	YES

All incoming packets and all outgoing packets through `C1394VPSPendPacket` are in big endian format. This means that when an incoming packet is a VPLabel Response Packet you have to byte swap it before assigning its buffer to a `C1394_VERSAPHY_PACKET_VPLABEL_READ_RESPONSE_LITTLE` pointer. But how do you know what type the packet is so as to determine whether it needs byte swapping in the first place?

Two functions have been defined for this purpose:

- **C1394PHYPacketIsVersaPHY:** This is a helper function to check whether a PHY packet is a normal PHY packet or a VersaPHY packet.
- **C1394VPGetPacketType:** This helper function returns an enumeration value of type `VersaPhyPacketType` that contains the exact type of the VersaPHY packet.

The `VersaPhyPacketType` enumeration is defined as shown below:

```
typedef enum
{
    VPIInvalid=0,
    VPPhyIdWriteRequest,
    VPPhyIdWriteResponse,
    VPPhyIdReadRequest,
    VPPhyIdReadResponse,
    VPLLabelWriteRequest,
    VPLLabelWriteResponse,
    VPLLabelReadRequest,
    VPLLabelReadResponse,
}
VersaPhyPacketType;
```

The general rule is that if a VersaPHY API entry point takes a `C1394_PHY_PACKET*` as a parameter, or returns such a pointer, then the packet data are in big endian format. Otherwise if the parameter is a type ending in `_LITTLE` then the packet data are in little endian.

VersaPHY Packet Initialization/Handling

Apart from the above two functions, more specialized BOOL functions exist, one for each packet type as shown in the table below:

BOOL functions for VersaPHY Packet Type
C1394VPISPhyIdReadRequestPacket
C1394VPISPhyIdReadResponsePacket
C1394VPISPhyIdWriteRequestPacket
C1394VPISPhyIdWriteResponsePacket
C1394VPISVPLLabelReadRequestPacket
C1394VPISVPLLabelReadResponsePacket
C1394VPISVPLLabelWriteRequestPacket
C1394VPISVPLLabelWriteResponsePacket

Similarly, a set of helper macros have been defined so that the developer can initialize properly a raw outgoing VersaPHY packet.

VersaPHY Packet Initialization Functions
VPInitPhyIdReadRequestPacket
VPInitPhyIdReadResponsePacket
VPInitPhyIdWriteRequestPacket
VPInitPhyIdWriteResponsePacket
VPInitVPLLabelReadRequestPacket
VPInitVPLLabelReadResponsePacket
VPInitVPLLabelWriteRequestPacket
VPInitVPLLabelWriteResponsePacket

Note that the transaction functions provided in the API internally take care of packet formatting, so in most cases you only have to worry about endianness if you want to use **C1394VPISendPacket**.

VersaPHY Transaction Serialization

All VersaPHY PhyID transactions with the same target PhyID are serialized with one another. This means that regardless of the originating application, they are all put on a separate, PhyID-specific, execution queue and then executed one by one.

Although most VersaPHY devices respond using concatenated transactions, it is possible for future devices to have to perform some internal processing before replying. Moreover, if there are multiple response packets then there is the chance that the VersaPHY device receives a new transaction request before it is done processing the current one. Most VersaPHY devices are expected to be low-cost implementations and thus most likely unable to internally queue multiple incoming requests and handle them properly.

In order to minimize the chances that this situation occurs (it would be extremely timing-sensitive and thus very hard to troubleshoot), the 1394 Class Driver serializes the requests.

Exactly the same thing is done for VersaPHY labels. All outgoing transaction requests are serialized through a separate execution queue for each VPLLabel.

However that it is impossible for the Class Driver to completely prevent this type of mistake:

- The Class Driver does not maintain information about the current PhyID of every VPLabel that is in use, thus it is possible to send one PhyID transaction request at the same time as a VPLabel transaction request to the same device.
- Different PCs may be on the same bus and try to talk to the same VersaPHY devices at the same time. In this case higher level software is responsible for coordinating the actions of the multiple agents.

VersaPHY Transaction Timeout

The VersaPHY standard does not make any mention as to the value that should be used for timing out VersaPHY transactions. Thus the Unibrain implementation thought it would be reasonable to use the same value used for normal 1394 transactions, which is the value contained by the **SPLIT_TIMEOUT** register.

Miscellaneous Topics

Endianness Considerations

The IEEE 1394 standard uses the big endian format for the representation of all 32-bit values involved in the standard (packet headers, Control and Status Registers (CSRs), etc).

This means that the low byte of a 32-bit word occupies the highest byte address while the highest byte the lowest address.

The following illustration demonstrates the practical difference between the little and big endian schemes:

Little Endian Scheme

Byte 3	Byte 2	Byte 1	Byte 0
Address X+3	Address X+2	Address X+1	Address X

Big Endian Scheme

Byte 3	Byte 2	Byte 1	Byte 0
Address X	Address X+1	Address X+2	Address X+3

Endianness is a slightly confusing concept because by talking about little and big endian one is driven away from the heart of the matter, which is actually the way that data are going to appear in the target host/device memory and how they are going to be interpreted there. While there are two ways to interpret a quadlet as an unsigned long (little endian and big endian) there is only one way to interpret a C-like ASCII string: The 1st character is at *string_base*, the 2nd character is at *string_base+1*, and so on until you find a zero byte. If node A wants to transmit a string to node B, then it does not care about CPU endianness. The string has to appear in the memory of node B, in exactly the same form as it initially appears in node A's memory.

If we are talking about sending an array of 32-bit long integers then necessarily we have to take the issue of endianness into consideration. The convention that the 1394 stack follows is the one mandated by the standard: quadlet data are always in big endian, and in general data are transmitted as they appear in memory (low address byte is transmitted first, highest address byte is transmitted last).

In general an application that communicates with another peer is free to use any kind of endianness in its data. However a server application for example that implements a 32-bit CSR that can accept lock transactions from ANY application, should keep in mind that the 32-bit data value and argument value in the lock transaction request packet is always in big endian format, and of course the 32-bit value in the response packet should also be big-endian format.

Endianness Swapping

The following macros and functions have been defined to assist in endianness swapping. The implementations are as optimized as possible for each platform (inline assembly for Intel, best direct calculation for other platforms). All functions are declared as inline, but inline expansion must be enabled for them to have the exact same performance as the respective macros.

All macros change their argument, while the functions return the swapped value and can be used in assignments and also take expressions as their argument.

Macros

```
SWAP_ENDIAN_16(a)    "a" should be an unsigned short
SWAP_ENDIAN_32(a)    "a" should be an unsigned long
SWAP_ENDIAN_64(a)    "a" should be an unsigned __int64
```

Functions

```
__inline unsigned short  SwapEndian16(unsigned short Value);
__inline unsigned long   SwapEndian32(unsigned long Value);
__inline unsigned __int64 SwapEndian64(unsigned __int64 Value);
```

Additionally one more function is available to byte swap a whole buffer either in place or to another buffer. The function is prototyped as follows:

```
void BlockSwapEndian32(void *buf, void *dest, unsigned int nquads);
```

If `dest` is the same as `buf` then byte swapping is performed in place. The function internally treats `buf` as a `ULONG` array and swaps quadlets from the first quadlet in the array to the last. This means that it is only safe to have the buffers pointed to by `buf` and `dest` overlap if `dest ≤ buf`.

NOTE: The `SWAP_ENDIAN_XXX` macros translate into inline assembly on x86 platforms. If the argument contains a complex expression with operators that are not recognized by the assembler, then a compilation error will occur.

Utility String Functions

This section lists several functions that can translate a value of some type to its equivalent string representation, for example a 1394 status code to a string, an acknowledge code to a string, etc. These functions are basically provided in order to allow developers to provide more user friendly messages.

These functions are prototyped as shown below:

```
TCHAR *C1394StatusString(STATUS_1394);
TCHAR *C1394AckCodeString(ULONG);
TCHAR *C1394RespCodeString(ULONG);
TCHAR *C1394TCodeString(ULONG);
TCHAR *C1394SpeedCodeString(ULONG);
```

An important restriction that these functions present is that they return a pointer to a static variable. This means that the results of a function are valid until the next call to the same function. For example a call like:

```
printf( "%s %s",
        C1394StatusString(STATUS_1394_SUCCESS),
        C1394StatusString(STATUS_1394_NOT_FOUND));
```

will print the same string twice. Each function uses its own static variable so mixing calls to different routines presents no problem. The call shown below will produce the expected outcome.

```
printf( "%s %s",
        C1394StatusString(STATUS_1394_SUCCESS),
        C1394SpeedCodeString(S800));
```

NOTE: These functions were derived from the sample code found in file **1394Strings.C**. In order to make the life of developers easier, if a project defines the compilation flag **FIREAPI_OLD_STRING_FUNCTIONS**, then the include file **FireAPI.H** also defines the original aliases for each function (`StatusString`, `AckCodeString`, `RespCodeString`, `TcodeString` and `SpeedString`). This way a project using `1394Strings.C` can simply move to the new functions by removing this file from the project, removing any `#include "1394Strings.h"` lines from any files and defining the **FIREAPI_OLD_STRING_FUNCTIONS** preprocessor flag at the project settings. This has been done in the `CMD1394` project that is distributed with the sample code.

64-bit Integer Arithmetic

Developers must be a little cautious when performing operations with 64-bit integers. These types are usually compiler-specific or platform-specific extensions to ANSI C/C++.

In the Windows platform, the `LONGLONG` & `ULONGLONG` types are defined for signed and unsigned 64 bit numbers. These are equivalent to the `__int64` and unsigned `__int64` types. Some operations with 64-bit integers can cause unexpected results because usually the compiler by default treats integer constants as signed integers and/or makes several more silent assumptions.

Consider the following examples (compiled with VC++ 5.0). In all cases the compiler treats the constant 1 as a signed 32-bit integer and acts according to its beliefs of what it should do:

```
ULONGLONG Value;
Value = 1<<10;    // Will yield the expected result.
Value = 1<<38;    // Will yield ZERO!!!

K = 38;
Value = 1<<K;     // Will yield 0x0000000000000040 (equal to 1<<6)!!!

K=31;
Value = 1<<K;     // Will yield 0xFFFFFFFF80000000!!!

K=63;
Value = 1<<K;     // Will yield 0xFFFFFFFF80000000!!!
```

The correct way to do the operations shown above is to cast the constant 1 to `ULONGLONG`. For example:

```
Value = ((ULONGLONG)1)<<38;    // Will yield 0x0000004000000000
K = 38;
Value = ((ULONGLONG)1)<<K;     // Will yield 0x0000004000000000
K=31;
Value = ((ULONGLONG)1)<<K;     // Will yield 0x0000000800000000
K=63;
Value = ((ULONGLONG)1)<<K;     // Will yield 0x8000000000000000
```

The respective rule applies if `value` was declared as `LONGLONG`. The constant will have to be casted to `LONGLONG` before shifting.

Similar rules apply for the intermediate results of integer operations.

Consider the following sample program:

```
#include <windows.h>
#include <stdio.h>
main(void)
{
    unsigned __int64 Value;
    unsigned int    I,K,M;

    I = 0x60000000;
    K=I;
    M=K;

    Value = I+K+M;
    printf("%I64X\n", Value);

    Value = ((ULONGLONG)I)+((ULONGLONG)K)+((ULONGLONG)M);
    printf("%I64X\n", Value);

    return 0;
}
```

The output of this program is

```
0x20000000
0x120000000
```

As you see, in the first case the intermediate results were bounded to 32-bits, while in the second they were not.

Path Speed Information

After each bus reset the class driver retrieves the self-ID packets and performs a structural and logical analysis on them. If the self-ID packets contain no structural and logical errors, then the class driver performs bus topology analysis.

If the self-ID packets indicate a valid topology then the class driver has complete topology information, for example which node is the parent of each node, and which nodes are a given node's children and so on.

After the topology analysis, the class driver analyzes the tree and calculates the maximum transmission rate between any two nodes on the bus. This rate is equal to the speed of the slowest device on the path that connects the two nodes (since 1394 does not allow loops (cycles) in the bus topology, there is exactly one such path). The results of these calculations are store in the class driver's speed table.

The speed table is a virtual 63x64 table. Each entry of the form (m,63) represents the broadcast speed, which is the speed of the slowest device on the bus.

The [0..62, 0..62] part of the speed table is obviously symmetric, since SPEED[A,B] equals SPEED[B,A] for any A and B. The class driver takes advantage of this property in order to save memory, and implements its speed table as a triangular array (the main diagonal is included).

The class driver stores the upper right half of the table, since the table has 64 columns but 63 rows. The illustration below shows which items are actually stored by the class driver (marked with S) and which items are inferred from the stored items (marked with a dash):

```

000000000011111111112222222222333333333344444444445555555555666666
0123456789012345678901234567890123456789012345678901234567890123
00 SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
01 -SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
02 --SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
03 ---SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
04 ----SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
05 -----SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
06 -----SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS

...

58 -----SSSSSS
59 -----SSSSSS
60 -----SSSSSS
61 -----SSSS
62 -----SS
    
```

The resulting table will have 64 elements in row 0, 63 elements in row 1, ..., and 2 elements in row 62. This sums to a total of 2+3+4+...+64=65*64/2 - 1=2079 elements.

The 2079 elements are stored in a single dimensional array and a formula is used to convert a pair (A,B) which indexes the virtual 2-dimensional table, to a value which indexes the single dimensional array.

The formula which performs the conversion from the 2-D indices to the linear index is given below (all array indices are zero-based):

$$\begin{aligned}
 &\text{Linear Array Index of (m,n) where } m \leq n \\
 &(64+63+\dots+(64-(m-1))) + (n-m) = \\
 &64 + (64-1) + (64-2) + \dots + (64-(m-1)) + (n-m) = \\
 &m*64 - (1+2+3+\dots + (m-1)) + (n-m) = \\
 &m*64 - m(m-1)/2 + n - m = \\
 &m*(64-(m-1)/2) + n - m = \\
 &(m*(129-m))/2 + n - m = \\
 &(m*(127-m))/2 + n
 \end{aligned}$$

Since the speed codes use 3-bits only, the class driver performs additional memory savings by packing two speed codes in a single byte. Thus the size of the class driver's speed table is 1040 bytes (defined as **SPEED_TABLE_BYTES**). The high 4-bits contain the item with the even index, and the low 4-bits the item with the odd index.

For ease of use the following functions and macros are provided in order to access the elements of the speed table.

`SPEED_TABLE_LINEAR_INDEX(A, B)` *(macro)*

Returns the linear index in the triangular array implementation of the conceptual 2-dimensional speed table item [A,B]

```
void C1394SetSpeedTableEntry(
    IN UCHAR           SpeedTable[SPEED_TABLE_BYTES],
    IN C1394_PHYSICAL_ID PhyID1,
    IN C1394_PHYSICAL_ID PhyID2,
    IN C1394_SPEED_CODE SpeedCode
);
```

Stores the value *SpeedCode* in the appropriate place in the speed table pointed to by *SpeedTable*. The first argument of this function is defined as shown above in order to have stricter type checking.

`SET_SPEED_TABLE_ENTRY(SpeedTable, PhyID1, PhyID2, SpeedCode)` *(macro)*

Stores the value *SpeedCode* in the appropriate place in the speed table pointed to by *SpeedTable*.

```
C1394_SPEED_CODE C1394GetSpeedTableEntry(
    IN UCHAR           SpeedTable[SPEED_TABLE_BYTES],
    IN C1394_PHYSICAL_ID PhyID1,
    IN C1394_PHYSICAL_ID PhyID2
);
```

Returns the speed code for the pair (PhyID1, PhyID2). The first argument of this function is defined as shown above in order to have stricter type checking.

`GET_SPEED_TABLE_ENTRY(SpeedTable, PhyID1, PhyID2, pSpeedCode)` *(macro)*

Store the speed code of the pair (PhyID1, PhyID2) into the variable pointed to by *pSpeedCode*.

The functions are implemented as inline functions for faster execution. The macros are implemented with the use of a helper variable. These two factors increase the memory used by an application at run-time. If this is a consideration, developers are suggested to copy the implementation of these functions/macros (file **1394Common.h**) and implement them in their own projects as appropriate.

Any application can retrieve a copy of the class driver's speed table by specifying the object identifier **OID_SPEED_TABLE** in a call to function **C1394QueryInformation**. Currently this information is only available for the local 1394 bus.

Note that in the table returned by the class driver, **SPEED_CODE_INVALID** is the speed code stored in any entry which represents a non-existent pair of nodes. This is defined as `0xF` which is good enough for 4-bit entries.

Bus Topology Information

An application can retrieve the information that the 1394 stack has about the topology of a bus, by calling **C1394QueryInformation** with the **OID_BUS_TOPOLOGY** object identifier. Currently topology information is only available for the local 1394 bus.

The information retrieved is in the form of a **C1394_BUS_TOPOLOGY_INFO** structure. This structure is defined as shown below:

```
typedef struct
{
    // The status of the topology information.
    STATUS_1394          TopologyStatus;

    // The IRM of that bus.
    C1394_NODE_ID        IRMNodeID;

    // The maximum hop count of that bus.
    ULONG                uMaxHopCount;

    // The node information array.
    C1394_NODE_INFO      NodeInfoArray[63];
}
C1394_BUS_TOPOLOGY_INFO, *PC1394_BUS_TOPOLOGY_INFO;
```

In normal circumstances *TopologyStatus* is equal to **STATUS_1394_SUCCESS**, and all the information in the structure is valid. If there was some error in the self-ID packets (which is an indication of a malfunctioning device on the bus) then *TopologyStatus* is set to **STATUS_1394_SELFID_ERROR** and the information is not valid.

Note: When calling **C1394QueryInformation** with the **OID_BUS_TOPOLOGY** identifier the caller must specify the ID of the bus of interest. This information is written over the **C1394_BUS_TOPOLOGY_INFO** structure that is passed to the function as the output buffer parameter. For example the calling code sequence would be similar to what is shown below:

```
C1394_BUS_TOPOLOGY_INFO TopologyInfo;
STATUS_1394              Status1394;

// We are interested in the local bus.
*((C1394_BUS_ID *)&TopologyInfo) = LOCAL_1394_BUS_ID;

// Make the call.
Status1394 = C1394QueryInformation( ClassAdapterHandle,
                                    OID_BUS_TOPOLOGY,
                                    &TopologyInfo,
                                    sizeof(TopologyInfo),
                                    NULL,
                                    NULL );
```

Also note that the return value of **C1394QueryInformation** has nothing to do with the validity of the information returned for this OID. For example in the code above, *Status1394* might be **STATUS_1394_SUCCESS** after the call, but the *TopologyStatus* field of the *TopologyInfo* variable might be set to **STATUS_1394_SELFID_ERROR**.

C1394_NODE_INFO

The most important piece of information inside the **C1394_BUS_TOPOLOGY_INFO** structure is the array of **C1394_NODE_INFO** structures. Each one contains all the information about a node on the bus that is available to the class driver at the time of the call. Most of the information is filled in at bus reset complete time (before the 1394 stack declares the bus reset to be completed), so it is practically available to callers at all times when **C1394IsBusResetInProgress** would return **FALSE**.

Other information, like the contents of the node's Configuration ROM become available a little later, and they are associated with a separate event indication (*not yet documented*) so that applications can know when to make their calls.

The **C1394_NODE_INFO** structure is defined as shown below:

```
typedef struct
{
    // A set of flags that provide additional information about the node.
    union {
        ULONG    uValue;
        struct {
            ULONG    Present:1;
            ULONG    IncorrectSelfID:1;
            ULONG    IsRoot:1;
            ULONG    LinkOn:1;
            ULONG    Contender:1;
            ULONG    IRM:1;
            ULONG    Valid_Speed:1;
            ULONG    Valid_MaxSpeedToNode:1;
            ULONG    Valid_MaxPayloadToNode:1;
            ULONG    Valid_bInitiatedBusReset:1;
            ULONG    Valid_uchSelfIDPackets:1;
            ULONG    Valid_uchTotalPorts:1;
            ULONG    Valid_fChildrenPorts:1;
            ULONG    Valid_fParentPort:1;
            ULONG    Valid_fInactivePorts:1;
            ULONG    Valid_PowerClass:1;
            ULONG    Valid_gap_count:1;
            ULONG    Valid_ParentNode:1;
            ULONG    Valid_uchNumberOfChildren:1;
            ULONG    Valid_fChildrenNodeIDs:1;
            ULONG    Valid_BusInfoBlock:1;
        };
    } NodeFlags;

    // The PHY ID of the node.
    C1394_NODE_ID    NodeID;

    // The maximum speed of the node.
    C1394_SPEED_CODE    Speed;

    // The maximum transmission rate from our local node to that node.
    C1394_SPEED_CODE    MaxSpeedToNode;

    // The maximum asynchronous payload from our local node to that node.
    ULONG                MaxPayloadToNode;

    // Indicates whether this node thinks that it is the one to cause the bus reset.
    BOOLEAN              bInitiatedBusReset;

    // The number of self ID packets that this node sent.
    UCHAR                uchSelfIDPackets;

    // The number of ports that this node has.
    UCHAR                uchTotalPorts;

    // The numbers of the active ports that are connected to children nodes.
    // Each bit set indicates a port number.
    ULONG                fChildrenPorts;

    // The numbers of the active ports that are connected to a parent node.
    // Each bit set indicates a port number. Only one bit should be set.
    ULONG                fParentPort;

    // The numbers of the inactive (disabled, disconnected or suspended) ports.
    // Each bit set indicates a port number.
    ULONG                fInactivePorts;

    // The power class of the node as reported in its self ID packet.
    C1394_POWER_CLASS    PowerClass;

    // The node's gap_count as reported in the self ID packet.
    C1394_GAP_COUNT      gap_count;

    // The physical ID of the parent of this node. If it is the root then
    // this has the value 63.
    C1394_PHYSICAL_ID    ParentNode;

    // The number of children that this node has.
    UCHAR                uchNumberOfChildren;

    // The PhyIDs of this node's children. Each bit that is set indicates a child.
    ULONGLONG            fChildrenNodeIDs;

    // The node's configuration ROM.
    C1394_BUS_INFO_BLOCK    BusInfoBlock;
}
C1394_NODE_INFO, *PC1394_NODE_INFO;
```

The *NodeID* field is always valid. Obviously the information returned inside the *NodeInfoArray* member of **C1394_BUS_TOPOLOGY_INFO** is in ascending NodeID order. That is *NodeInfoArray[0]* is (1023,0), *NodeInfoArray[1]* is (1023,1) etc. The *NodeID* field was added to the **C1394_NODE_INFO** structure mainly so that a pointer to the **C1394_NODE_INFO** structure can be passed as a parameter to another routine without having to pass the NodeID as well.

NodeFlags.Present indicates whether the node whose 16-bit Node ID is equal to *NodeID* is physically present on the bus. If *NodeFlags.Present* is zero, then **NO OTHER** field in the **C1394_NODE_INFO** structure (except for *NodeID*) is valid. Applications should **always check** this field before proceeding to do anything else with this structure.

NodeFlags.IncorrectSelfID indicates whether the selfID packet of this node has any problem.

NodeFlags.IsRoot is set to 1 for the root node, and to 0 for all other nodes.

NodeFlags.LinkOn is set to 1 for a node, if the node's self ID packets had the *link-on* bit set to 1.

NodeFlags.Contender is set to 1 for a node, if the node's self ID packets had the *contender* bit set to 1.

NodeFlags.IRM is set to 1 for the node who is the IRM on the bus, and to 0 for all other nodes.

The *Valid_XXX* fields of *NodeFlags* indicate whether the respective XXX field of the **C1394_NODE_INFO** structure contains valid information.

The **IS_NODE_INFO_FIELD_VALID** macro can be used to conveniently test whether a specific field is valid.

For example:

```
IS_NODE_INFO_FIELD_VALID(pNodeInfo, MaxSpeedToNode)
```

checks whether the *MaxSpeedToNode* field of the **C1394_NODE_INFO** structure pointed to by *pNodeInfo* is valid.

See Also

C1394QueryInformation, OID_PHYSICAL_NODES, OID_LINK_ON_NODES, OID_CONTENDER_NODES, OID_SPEED_TABLE, OID_NODE_COUNT, OID_HOP_COUNT

SelfID Analysis Error Codes

The following flags are defined for indicating the errors detected during the selfID packet analysis that the class driver performed after a bus reset. If the selfID analysis error code retrieved through **OID_SELFID_ANALYSIS_ERROR** is not equal to **SELFID_OK**, then one or more of the flags described below may be set.

Value	Description
SELFID_OK	There is no problem in the selfID packets.
SELFID_NO_PACKETS	No self ID packets were received after the bus reset.
SELFID_NO_ROOT	No root node was found among the self ID packets.
SELFID_DISCONTIGUOUS	The phyIDs in the self ID packets are discontinuous. This will error will be indicated if after the selfID packet of node X follows a selfID packet for node Y, with $Y > (X+1)$.
SELFID_NO_PORTS_PRESENT	A self ID packet was received that indicated that no ports were present on a node.
SELFID_NO_PORTS_CONNECTED	A self ID packet was received that indicated that a node had no connected ports.
SELFID_DISCONNECTED_NONZERO_PHYID	The adapter is disconnected, but its PhyID is non zero.
SELFID_DUPLICATE	More than one set of selfIDs was found with the same PhyID.
SELFID_OUT_OF_ORDER	A selfID packet was received with a PhyID smaller than the PhyID of the previous selfID packet.
SELFID_INCORRECT_ROOT_PACKET	Some node thinks (incorrectly) that it is connected only to child nodes, but this cannot be the root node because there are more selfID packets.
SELFID_TOO_MANY_NODES	63 nodes were identified, and there were more selfID packets. This should not occur unless there is some problematic hardware on the bus.
SELFID_EXPECTED_PCK_0	The next selfID packet was expected to be a selfID #0 packet, but another type of selfID packet was found.
SELFID_EXPECTED_PCK_N	The next selfID packet was expected to be a selfID #1,#2 or #3 packet, but another type of selfID packet was found.
SELFID_EXPECTED_PCK_4	A selfID packet #3 had the 'more' bit set.
SELFID_NO_MORE_PACKETS	The last selfID packet of the block received after the bus reset, had the 'more' bit set.
SELFID_INCORRECT_PHYID	A selfID #1,2 or 3 packet did not have the same PhyID as the respective #0 packet.
SELFID_MULTIPLE_PARENT_PORTS	A selfID packet reported that more than one ports were connected to a parent node.
SELFID_NO_LEAVES	No leaf nodes were found in the selfID packets.
SELFID_PARENT_CHILD_IMBALANCE	The total number of parent ports is not equal to the total number of child ports.
SELFID_UNEXPECTED_ERROR	An unexpected error occurred.

NOTE: SelfID packet analysis is bus-specific. This means that in order to retrieve the selfID analysis error through **OID_SELFID_ANALYSIS_ERROR**, you have to specify a bus number as the input parameter to the C1394QueryInformation call.

This can be done as shown in the code below:

```

STATUS_1394 Status1394;
ULONG          uErrorCode;

// We want information for the local bus.
*((C1394_BUS_ID*)&uErrorCode) = LOCAL_1394_BUS_ID;

Status1394 = C1394QueryInformation( ClassAdapterHandle,
                                   OID_SELFID_ANALYSIS_ERROR,
                                   &uErrorCode,
                                   sizeof(ULONG),
                                   NULL,
                                   NULL );

// Is everything OK?
if (Status1394 != STATUS_1394_SUCCESS)
{
    // This should not happen if all parameters are valid.
}

```

Most of the errors shown in the table above will never occur, unless there is a problematic adapter on the bus.

In practice some of these errors are occur momentarily when plugging or unplugging a device from the bus. ‘Momentarily’ means that a simple software-initiated bus reset immediately solves the problem (in the sense that after the bus reset there is no problem with the selfID packets).

The Serial Bus Manager driver is deemed responsible for recovering from these errors, and UBSBM.SYS will initiate a bus reset to correct this situation whenever it occurs. Taking this into consideration, applications should be able to handle such an error gracefully, and wait for the Serial Bus Manager to correct the problem.

The following situations have been witnessed with hardware that is otherwise perfectly functional:

1. SELFID_NO_PACKETS
2. SELFID_NO_ROOT
3. SELFID_DISCONTIGUOUS
4. SELFID_OUT_OF_ORDER
5. SELFID_INCORRECT_ROOT_PACKET
6. SELFID_PARENT_CHILD_IMBALANCE

SELFID_DISCONTIGUOUS by itself occurs when an intermediate selfID packet (other than the root node’s) is missing.

SELFID_DISCONTIGUOUS occurs together **SELFID_OUT_OF_ORDER** when an intermediate selfID packet is found at the wrong position.

SELFID_NO_ROOT occurs if the selfID packet of the root node is missing or the selfID packet of the root node incorrectly reports a port connected to a parent.

SELFID_INCORRECT_ROOT_PACKET occurs if an intermediate selfID packet is incorrectly reporting the status of the node’s ports, or an intermediate selfID packet incorrectly appears after the selfID of the root node.

SELFID_PARENT_CHILD_IMBALANCE obviously appears in all cases when a selfID packet is missing. It has never occurred by itself.

Topology Analysis Error Codes

The following flags are defined for indicating the errors detected during the topology analysis that the class driver performed after a bus reset. Topology analysis is only attempted if there was no structural or validity problem detected in the selfID analysis.

Topology analysis performs a higher level validation of the selfID packets to see if they are describing a valid tree topology, according to the restrictions of the 1394 standard.

The topology analysis error code retrieved through **OID_TOPOLOGY_ANALYSIS_ERROR**. If the value returned is not equal to **TOPOLOGY_OK**, then one or more of the flags described below may be set (usually only one will be set).

Value	Description
TOPOLOGY_OK	The selfID packets describe a valid tree topology. The class driver has derived all the necessary information, and clients can retrieve it through the OID_BUS_TOPOLOGY identifier.
TOPOLOGY_ANALYSIS_NOT_PERFORMED	Topology analysis was not performed because an error was detected in selfID analysis.
TOPOLOGY_TOO_MANY_CHILD_PORTS	A selfID packet incorrectly reported the state of its ports, by reporting more ports connected to children nodes than actually are. This lead the topology analysis algorithm to a failure.
TOPOLOGY_PORT_STATUS	One or more selfID packets incorrectly reported the state of their ports, and this fact made a subset of the nodes on the bus to appear like they form a complete tree structure. However this is not possible since there are additional nodes on the bus.
TOPOLOGY_INCONSISTENT	The node that was recognized as the root from topology analysis, claims that it has a parent port. Either this information is wrong or there should have been more selfID packets.

‘Real’ topology errors are very rare. A topology error other than **TOPOLOGY_ANALYSIS_NOT_PERFORMED**, should practically never occur, unless there is a problematic device on the bus.

NOTE: Topology analysis is bus-specific. This means that in order to retrieve the topology analysis error through **OID_TOPOLOGY_ANALYSIS_ERROR**, you have to specify a bus number as the input parameter to the **C1394QueryInformation** call.

This can be done as shown in the code below:

```

STATUS_1394 Status1394;
ULONG      uErrorCode;

// We want information for the local bus.
*((C1394_BUS_ID*)&uErrorCode) = LOCAL_1394_BUS_ID;

Status1394 = C1394QueryInformation( ClassAdapterHandle,
                                   OID_TOPOLOGY_ANALYSIS_ERROR,
                                   &uErrorCode,
                                   sizeof(ULONG),
                                   NULL,
                                   NULL );

// Is everything OK?
if (Status1394 != STATUS_1394_SUCCESS)
{
    // This should not happen if all parameters are valid.
}

```

Manipulating CYCLE_TIME timestamps

FireAPI provides the **C1394GetCycleTime** function to clients so that they can read the value of the CYCLE_TIME register. An application can use the cycle time in order to perform very accurate timestamping.

The return value of **C1394GetCycleTime** can be assigned to the *uValue* member of the **C1394_CYCLE_TIME_REGISTER** structure, so that the structure bit fields can be used to parse the information in the cycle timer. This structure is defined as shown below:

```
typedef union
{
    ULONG    uValue;

    struct
    {
        ULONG CycleOffset:12;
        ULONG CycleCount:13;
        ULONG SecondCount:7;
    };
}
C1394_CYCLE_TIME_REGISTER, *PC1394_CYCLE_TIME_REGISTER;
```

The 12-bit *CycleOffset* field shall be updated on each tick of the local 24.576 MHz PHY clock, with the exception that an increment from the value 3071 shall cause a wraparound to zero and shall carry into the *CycleCount* field.

The value is the fractional part of the isochronous cycle of the current time, in units that are counts of the 24.576 MHz clock.

The 13-bit *CycleCount* field shall increment on each carry from the *CycleOffset* field, with the exception that an increment from the value 7999 shall cause a wraparound to zero and shall carry into the *second_count* field.

The value is the fractional part of the second of the current time, in units of 125 microsec.

The 7-bit *SecondCount* field shall increment on each carry from the *CycleCount* field, with the exception that an increment from the value 127 shall cause a wraparound to zero.

The sample below repeatedly calls the Win32 **Sleep** function and reads the CYCLE_TIME register before and after the call and calculates the time difference.

The sample calculates correctly the difference, provided that no more than 128 seconds have elapsed between the two timestamps.

Using this sample you can have a good estimate with regards to the accuracy, or rather the inaccuracy, of the **Sleep** function.

Initially the sample performs a tight loop calling **C1394GetCycleTime** in order to provide you with an estimate the overhead of this call on your system. On a test run on a PII-450, the program yielded about 225.000 calls to **C1394GetCycleTime** per second.

As you may find out, the overhead of calling this function is zero compared to the overhead of calling **printf** inside the calculations loop⁴². This is why the program stores all numbers in an array, and calculates and displays results after the measurements have completed.

⁴² This usually also involves the overhead of scrolling the console window output, an operation that is very slow on many graphics adapters.

```

#include <stdio.h>
#include <FireAPI.h>

ULONG CycleTimeDifference(ULONG a_Val2, ULONG a_Val1)
{
    ULONGLONG Offsets2, Offsets1;
    ULONG OffsetDiff;

    C1394_CYCLE_TIME_REGISTER CycleTime1;
    C1394_CYCLE_TIME_REGISTER CycleTime2;
    C1394_CYCLE_TIME_REGISTER CycleTimeDiff;

    CycleTime1.uValue = a_Val1;
    CycleTime2.uValue = a_Val2;

    Offsets1 = 8000*3072*CycleTime1.SecondCount +
        3072*CycleTime1.CycleCount +
        CycleTime1.CycleOffset;

    Offsets2 = 8000*3072*CycleTime2.SecondCount +
        3072*CycleTime2.CycleCount +
        CycleTime2.CycleOffset;

    // If the seconds' count have wrapped, then adjust it
    if (CycleTime2.SecondCount < CycleTime1.SecondCount)
        Offsets2 += 8000*3072*128;

    // Calculate the difference in cycle offsets.
    OffsetDiff = (ULONG) (Offsets2 - Offsets1);

    //////////////////////////////////////
    // Convert back into CYCLE_TIME format.
    //////////////////////////////////////

    // Cycle offsets
    CycleTimeDiff.CycleOffset = (ULONG) OffsetDiff % 3072;

    // Cycle counts
    OffsetDiff /= 3072;
    CycleTimeDiff.CycleCount = (ULONG) OffsetDiff % 8000;

    // Second counts.
    CycleTimeDiff.SecondCount = OffsetDiff/8000;

    return CycleTimeDiff.uValue;
}

#define DATA_ENTRIES 100
struct
{
    DWORD dwDelayMsec;
    C1394_CYCLE_TIME_REGISTER TimeStamp;
}
Data[DATA_ENTRIES];

#define TEST_CALLS 400000

main(void)
{
    C1394_ADAPTER_HANDLE C1394AdapterHandle;
    C1394_CYCLE_TIME_REGISTER Dif;
    STATUS_1394 Status1394;
    ULONG I;
    ULONG uStart, uEnd;
    C1394_CYCLE_TIME_REGISTER CyclDiff;

    srand(GetTickCount());

    C1394Initialize();
    C1394OpenAdapter( NULL, NULL, &C1394AdapterHandle );

```

```

////////////////////////////////////
// Make a bunch of calls to C1394GetCycleTime
////////////////////////////////////
uStart = C1394GetCycleTime( C1394AdapterHandle );

for (I=0; I<TEST_CALLS; I++)
    uEnd = C1394GetCycleTime( C1394AdapterHandle );

CyclDiff.uValue = CycleTimeDifference(uEnd, uStart);

printf( "[%03x:%04x:%03x] <=> (%3u msec) for %u calls to C1394GetCycleTime\n\n",
        CyclDiff.SecondCount,
        CyclDiff.CycleCount,
        CyclDiff.CycleOffset,
        CyclDiff.SecondCount*1000 + (CyclDiff.CycleCount*125)/1000,
        TEST_CALLS );

////////////////////////////////////
// Use C1394GetCycleTime to see how long Sleep takes.
////////////////////////////////////

// Initialize the random wait times.
for (I=0; I<DATA_ENTRIES; I++)
    Data[I].dwDelayMsec = 20+rand()%200;

// Simply store the timestamp after each wait.
for (I=0; I<DATA_ENTRIES; I++)
{
    Data[I].TimeStamp.uValue = C1394GetCycleTime( C1394AdapterHandle );
    Sleep(Data[I].dwDelayMsec);
}

// Now calculate the differences.
for (I=0; I<DATA_ENTRIES-1; I++)
{
    CyclDiff.uValue = CycleTimeDifference( Data[I+1].TimeStamp.uValue,
                                           Data[I].TimeStamp.uValue );

    printf( "[%03x:%04x:%03x]-[%03x:%04x:%03x] = [%03x:%04x:%03x] <=>"
           " (%3u msec) for Sleep(%3u)\n",
           Data[I+1].TimeStamp.SecondCount,
           Data[I+1].TimeStamp.CycleCount,
           Data[I+1].TimeStamp.CycleOffset,
           Data[I].TimeStamp.SecondCount,
           Data[I].TimeStamp.CycleCount,
           Data[I].TimeStamp.CycleOffset,
           CyclDiff.SecondCount,
           CyclDiff.CycleCount,
           CyclDiff.CycleOffset,
           (CyclDiff.CycleCount*125)/1000,
           Data[I].dwDelayMsec );
}

C1394CloseAdapter( C1394AdapterHandle );
C1394Terminate();
return 0;
}

```

The sample **CycleTimeDifference** routine, first converts the timestamps into the unit of *Cycle Offsets*, calculates the difference in cycle offsets and then converts this back into cycle time format. The program declares the `Offsets1` and `Offsets2` variables as `ULONGLONG` instead of `ULONG` in order to avoid a possible overflow during the calculations.

As you might see by this sample, the **Sleep** function is rather inaccurate. If you need accurate timing, then you can rely on **C1394GetCycleTime** to get rather accurate measurements, taking into consideration that a PC system can perform more than 200.000 calls per second while the cycle count portion of cycle time ‘only’ has 8000 counts per second.

Application Reaction Time

Application developers may need to measure the reaction time of an application to an incoming transaction request.

FireAPI provides the **C1394GetCycleTime** function that application can use to perform very accurate timestamping, but this is not enough to give to an application information about the *Reaction Time*. The *Reaction Time* can be defined as the amount of time between the moment a packet was received and the time the application got it.

Obviously the *Reaction Time* will vary depending on what kind of activity is running on the system at the same time.

The only way to accurately measure the reaction time of an application is to use a 1394 bus analyzer and capture the request and response packets.

The difference in time between the request and the response packet is equal to the amount of time it took the system to:

1. Raise an interrupt for the reception of a transaction request packet
2. Have the class driver process the packet and store it in the request queue of an address range.
3. Notify the application about the incoming packet by signalling the event object that is associated with the address range.
4. Have the operating system wake and schedule the thread of the application that was waiting on the event object.
5. Have the application call **C1394GetNextRequest** to retrieve the packet into user mode.
6. Process the packet and send a response packet using **C1394SendResponse** or another transaction response function.

The above procedure however calculates the overall processing time which also includes the time needed to send the response packet.

If we want to calculate the time from the moment the request packet was sent up to the moment the application got it (the call to **C1394GetNextRequest** returned) then a slightly more complex setup is required.

In this setup the receiving application must be run on the root node (which also acts as cycle master). The receiving application will make a call to **C1394GetCycleTime** as soon as its call to **C1394GetNextRequest** returns. Because the receiving application runs on the root node, the cycle timer value read is from the same clock that is being used to put the cycle time in the cycle start packets.

So combining the output of the program with the capture log of the analyzer will allow some to make rather accurate measurements.

A sample program that performs this kind of operation is the REACT sample in the sample code. Note that this sample keeps all timestamps in an internal array and only displays the results at the end. This is done in order to put as little delay as possible in the timing. It is important to remember that console output operations (like calls to **printf**) are very slow operations compared to what a CPU can do. Putting **printf** statements between statements introduces considerable delay which makes sensitive timing measurements be very inaccurate.

A portion of the output of this program for a sample run is listed below⁴³. In this test run an instance of CMD1394.EXE on a remote node executed a FILE command that sent a series of quadlet read requests to the root node, where the REACT application was running. The delay between two successive reads from the CMD1394.EXE was such that the receiving application always had the chance to empty the request queue of the address range and fall back into the **WaitForSingleObject** call.

```
073:0312:a85    073:0313:0bc    000:0000:237
073:031d:155    073:031d:36e    000:0000:219
```

⁴³ The receiving application was running on an otherwise idle system.

The 1st timestamp is the value returned from **C1394GetCycleTime** when the **WaitForSingleObject** call returns. The 2nd timestamp is the value returned from **C1394GetCycleTime** when the **C1394GetNextRequest** call returns. The 3rd timestamp is the difference between the two.

The output of the application shows us that the application only needs between 220 and 240 cycle offsets from the time it gets notified, until the time it gets the request packet. This is a very small amount of time. 3072 cycle offsets are 125 microsec, so 240 cycle offsets are less than 10 microsec.

The corresponding packets from the capture of the 1394 bus analyzer are shown below:

12	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631202A) Timestamp(1CF9:9E2)
13	S400 ReadDQ(lbl: 0) dest_ID(FFC5) src_ID(FFC2) offset(0000: 00005150) ACK(pending) Timestamp(1CFA:2B5)
14	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631302A) Timestamp(1CFA:9E2)
15	S400 RdRespDQ(lbl: 0) dest_ID(FFC2) src_ID(FFC5) rcode(complete) quad_data(0x46697265) ACK(complete) Timestamp(1CFB:148)
16	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631402A) Timestamp(1CFB:9E2)
17	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631502A) Timestamp(1CFC:9E2)
18	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631602A) Timestamp(1CFD:9E2)
19	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631702A) Timestamp(1CFE:9E2)
20	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631802A) Timestamp(1CFF:9E2)
21	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631902A) Timestamp(1D00:9E2)
22	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631A02A) Timestamp(1D01:9E2)
23	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631B02A) Timestamp(1D02:9E2)
24	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631C02A) Timestamp(1D03:9E3)
25	S400 ReadDQ(lbl: 0) dest_ID(FFC5) src_ID(FFC2) offset(0000: 00005150) ACK(pending) Timestamp(1D04:6B2)
26	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631D02A) Timestamp(1D04:9E3)
27	S400 RdRespDQ(lbl: 0) dest_ID(FFC2) src_ID(FFC5) rcode(complete) quad_data(0x46697265) ACK(complete) Timestamp(1D05:3CD)
28	S100 CycleSt(lbl: 0) dest_ID(FFFF) src_ID(FFC5) offset(FFFF: F0000200) quad_data(time: 0xE631E02A) Timestamp(1D05:9E3)

Packet 12 is the cycle start packet for cycle 312₁₆. It is in the same cycle that the application got the notification that a request has been received.

Packet 13 was sent (1CFA:2B5 – 1CF9:9E2) = * C00+2B5-9E2 = 4D3 cycle offsets after the cycle start packet, that is at cycle time 312:02A+ 000:4D3 = 312:4FD.

The application got signalled at 312:A85 which is A85-4FD = 588 cycle offsets later (24 microsec) and got the packet at 313:0BC, which is C00+0BC-4FD = 7BF cycle offsets later which is 81 microsec.

* C00₁₆ is hex for 3072₁₀ which is the maximum value for cycle offsets.

The response packet got sent at cycle time 313:02A+(1CFB:148-1CFA:9E2) which is A93 cycle offsets (110 microsec) after the request packet was sent.

So, it took the application 24 microsec to get signalled, another 57 microsec to get the request and another 29 microsec to transmit the response packet.

A similar analysis can be performed for the read request sent with packet 25.

The above results apply in the case where the application receives requests one by one⁴⁴. When more than one packets are received into the request queue, then the signalling overhead is only paid once, and it takes the application less time to make additional calls to **C1394GetNextRequest** because the first call caches more than one packets in the application side, so the next calls don't have to switch to kernel mode in order to retrieve the request packets.

The same tests can be performed with kernel mode CSRs. Sample measurements that were performed in Unibrain's laboratory indicate that the same computer that needs 100-120 microsec to respond to a quadlet read request, requires 80-90 microsec to respond to a 20 byte block read.

This is of course an expected result, since there is more overhead involved in interacting with user mode. Another difference between a kernel mode CSR and a user mode CSR is that the reaction time of kernel mode CSRs will not be affected by system load as much as the performance of a user mode application, even if the application runs at high priority.

Asynchronous transaction requests in kernel mode are usually serviced in what is called *Dispatch Level* which is a mode of execution that can only be interrupted by interrupts. No user mode thread can be scheduled on a processor while the processor is executing code at *Dispatch Level*.

⁴⁴ The first call to **C1394GetNextRequest** returns a packet and the next returns NULL.

Accessing the Link Layer Registers

FireAPI provides a way for clients to access the adapter's link registers. This is achieved with the use of the **OID_LINK_REGISTER_ACCESS** object identifier in conjunction with **C1394QueryInformation** for reading a Link Register or **C1394SetInformation** for writing a Link Register.

The structure used in combination with the **OID_LINK_REGISTER_ACCESS** object identifier is **C1394_LINK_REGISTER_ACCESS**. This structure is defined as shown below:

```
typedef struct
{
    // Flags for the operation.
    USHORT      Flags;

    // The byte offset at which to perform the operation.
    // This can be in the range 0..4095, and must be divisible by 4,
    // so it can only take values 0,4,8,...,4092.
    USHORT      ushByteOffset;

    // On reads it contains the value that was read from the link register.
    // On writes it contains the value to be written to the link register.
    ULONG      uValue;
}
C1394_LINK_REGISTER_ACCESS, *PC1394_LINK_REGISTER_ACCESS;
```

No flags have been defined yet for these operations, so the *Flags* field should be set to zero.

Direct access to the link registers has been permitted in order to facilitate the operation of diagnostic applications and possibly allow some customization of the low level operational settings of an adapter wherever that might be required (PCI latency, FIFO sizes etc).

WARNING: Writing to a link register can cause serious trouble to the operation of the 1394 stack, which can mean anything from improper operation to a system crash. If you attempt this then you assume all responsibility for the results of the operation. Additionally, it is possible that although such an operation does not cause problem to a specific version of the drivers, it might cause problems to later versions. Please, refrain from accessing these registers unless you specifically know what you are doing.

Changing the FIFO settings

The API provides a way for clients to get informed about and modify the FIFO settings of the adapter. This is only supported on Lynx adapters. OHCI adapters have fixed FIFO settings.

This is achieved with the use of the **OID_ADAPTER_FIFO** object identifier in conjunction with **C1394QueryInformation** for finding out the current settings and options, and **C1394SetInformation** for modifying the current settings and options.

This functionality is only provided for fine-tuning or customizing the operation of the drivers for specific applications, if the default settings are proved insufficient.

For example, an isochronous intensive application might set the size of the asynchronous transmit FIFO to a small number if it knows that it will not produce asynchronous packets larger than 256 bytes. This way it can leave more FIFO space for isochronous operations, which will lessen the probability of FIFO underruns or DMA transfer errors due to PCI bus congestion.

The structure used in combination with the **OID_ADAPTER_FIFO** object identifier is **C1394_ADAPTER_FIFO_SETTINGS**. This structure is defined as shown below:

```
typedef struct
{
    // Indicates which of the substructures should be used.
    ULONG   FIFOType;

    union
    {
        struct
        {
            // Flags that describe the FIFO & driver settings,
            // and possibly control the operation.
            ULONG   Flags;

            // The total size of the FIFO in bytes.
            ULONG   uTotalSize;

            // The size used for the asynchronous and isochronous receive FIFO.
            ULONG   ReceiveSize;

            // The size of the asynchronous transmit FIFO.
            ULONG   AsynchXmitSize;

            // The size of the isochronous transmit FIFO.
            ULONG   IsochXmitSize;
        }
        LynxFIFO;
    };
};
C1394_ADAPTER_FIFO_SETTINGS, *PC1394_ADAPTER_FIFO_SETTINGS;
```

The *FIFOType* field indicates which sub-structure is valid. *FIFOType* is filled by the 1394 stack upon return of a **C1394QueryInformation(OID_ADAPTER_FIFO)** call, and it should be filled with that same value by a client that calls **C1394SetInformation(OID_ADAPTER_FIFO)**. Each adapter only supports one of the defined codes for *FIFOType*.

Currently only one sub-structure is defined, that is used on Texas Instruments PCILynx based boards. The value of *FIFOType* for this type of boards is **FIFO_TYPE_LYNX**.

The possible flags that can be set to the *LynxFIFO.Flags* field are explained in the table below.

Value	Description
FIFO_CAN_CHANGE_AT_RUNTIME	<p>Indicates that the adapter and the drivers support changing the FIFO settings while the drivers operate.</p> <p>A client can call C1394SetInformation with <code>OID_ADAPTER_FIFO</code> only if this bit is set in the information returned by C1394QueryInformation(<code>OID_ADAPTER_FIFO</code>).</p> <p>This flag is only meaningful in the information returned by C1394QueryInformation. The drivers ignore it in the C1394SetInformation call.</p>
FIFO_SUPPORTS_AUTO_ZERO_ISO_XMIT	<p>Indicates that the driver supports the feature whereby it automatically manages the FIFO setup, zeroing the size of the isochronous transmit FIFO when there are no adapter channels open that perform isochronous transmit.</p>
FIFO_AUTO_ZERO_ISO_XMIT	<p>In the information returned by C1394QueryInformation it indicates whether the driver has currently enabled or not the automatic zeroing of the isochronous transmit FIFO.</p> <p>In a C1394SetInformation call, if this bit is set to 1, then it instructs the driver to enable auto-zeroing of the xmit FIFO.</p>
FIFO_DISABLE_AUTO_ZERO_ISO_XMIT	<p>This flag is only meaningful for a C1394SetInformation call. If it is set to one, then it instructs the driver to disable auto-zeroing of the isochronous transmit FIFO.</p> <p>Only one of the FIFO_AUTO_ZERO_ISO_XMIT and FIFO_DISABLE_AUTO_ZERO_ISO_XMIT flags can be specified in a C1394SetInformation call. If none is specified, then the current mode is retained.</p>
FIFO_ISO_XMIT_ZEROED	<p>This flag can only be applied in the information returned by a C1394QueryInformation call.</p> <p>If it is set, then it indicates that currently the isochronous transmit FIFO is zeroed. This means that the value reported by the <i>IsochXmitSize</i> field is currently used in the other FIFOs according to the current policy.</p>
FIFO_FAVOUR_RECEIVE FIFO_FAVOUR_TRANSMIT FIFO_FAIR_SPLIT	<p>Indicates which is the current replacement policy of the 1394 stack when it zeroes the isochronous transmit stack.</p> <p>FIFO_FAVOUR_RECEIVE_FIFO indicates that the size freed by the isochronous transmit FIFO is added to the receive FIFO.</p> <p>FIFO_FAVOUR_TRANSMIT_FIFO indicates that the size freed by the isochronous transmit FIFO is added to the asynchronous transmit FIFO.</p> <p>FIFO_FAIR_SPLIT_FIFO indicates that the size freed by the isochronous transmit FIFO is equally divided between the asynchronous transmit FIFO and the receive FIFO.</p> <p>Only one of these flags can be specified in a call to C1394SetInformation, if the caller wants to change the replacement policy. If none of these flags is specified then the policy remains unchanged.</p>

FIFO_CHANGE_SETTINGS	<p>Can only be specified in a C1394SetInformation call. It requests the driver to read the values supplied and update the FIFO sizes accordingly.</p> <p>Note that the values supplied refer to that values to be used when there are isochronous transmit channels open. Do not specify less than 128 for any FIFO size.</p> <p>If the sum of the partial FIFO sizes is less than the total available FIFO then some FIFO space will simply get wasted. The sum of the partial FIFO sizes cannot be larger than the total available FIFO size.</p>
----------------------	--

For example, suppose that a call is made to **C1394QueryInformation(OID_ADAPTER_INFO)** and the returned structure contains the following information:

```

FIFOType = FIFO_TYPE_LYNX;
LynxFIFO.Flags = FIFO_CAN_CHANGE_AT_RUNTIME |
                 FIFO_SUPPORTS_AUTO_ZERO_ISO_XMIT |
                 FIFO_AUTO_ZERO_ISO_XMIT |
                 FIFO_FAVOUR_TRANSMIT;

LynxFIFO.uTotalSize = 4096;
LynxFIFO.ReceiveSize = 2048;
LynxFIFO.AsynchXmitSize = 1040;
LynxFIFO.IsochXmitSize = 1008;

```

This means that the adapter supports runtime FIFO changing and auto-zeroing the isochronous transmit FIFO, auto-zeroing is enabled and the current policy is to favour the asynchronous transmit FIFO. The total FIFO size is 4KB, and it is split into 2KB for receive, 1040 bytes for asynchronous transmit FIFO and 1008 bytes for isochronous transmit FIFO.

If there is no adapter channel open for isochronous transmit, then the current size of the asynchronous transmit FIFO is 2KB as well (1040+1008).

If a client wants to change the replacement policy to favour the receive FIFO, it has to make a **C1394SetInformation(OID_ADAPTER_FIFO)** call and specify:

```

FIFOType = FIFO_TYPE_LYNX;
LynxFIFO.Flags = FIFO_FAVOUR_RECEIVE;

```

If a client wants both to change the replacement policy and the default FIFO sizes, it has to make a **C1394SetInformation(OID_ADAPTER_FIFO)** call and specify:

```

FIFOType = FIFO_TYPE_LYNX;
LynxFIFO.Flags = FIFO_FAVOUR_RECEIVE | FIFO_CHANGE_SETTINGS;
LynxFIFO.ReceiveSize = new_value_1;
LynxFIFO.AsynchXmitSize = new_value_2;
LynxFIFO.IsochXmitSize = new_value_3;

```

The contents of the *LynxFIFO.uTotalSize* field are ignored in calls to **C1394SetInformation**.

The ability to control the FIFO size is very important and can be critical for high throughput applications. Developers should keep the FIFO sizes in mind when designing their systems because this will be a critical factor.

For example, by setting the receive fifo to 3.5 KB you will be able to receive at the same time two isochronous streams, each with packet size of 2048 bytes⁴⁵, from one 1394 adapter with no errors. With the default setting of 2KB this would be impossible.

⁴⁵ $2 * 8000 * 2048 = 32.768.000$ bytes/sec

Similarly, by using the default setting for the isochronous transmit FIFO, which is 1KB, and trying to transmit an isochronous stream with packets of 2KB, you may observe any of the behaviours described below:

- There is no transmission error whatsoever, no matter what is the activity on the system.
- There is no transmission error while the system is idle, but if some disk activity occurs, then transmission errors occur.
- There are random transmission errors while the system is idle (for example about 1-2 packets out of 50-100 thousand packets), and many more if there is other activity as well.
- There are periodic transmission errors while the system is idle (for example 1 packet every 90 thousand packets), and many more, random errors if there is other activity as well.
- There are many transmission errors, even when the system is idle (for example more than 1 packet fails every 500 packets).

Such patterns of behaviour have been actually observed using the same test program on different computers that were equipped with the same type of 1394 adapters.

In ALL cases, when the isochronous transmit FIFO size was increased to 2KB no errors would occur, even if the system was under severe stress (high disk and CPU activity).

Part II

FireAPI Function Reference

Initialization Functions

C1394Initialize

Initializes 1394 support for the caller.

```
STATUS_1394 C1394Initialize(void);
```

Return Values

If the function is successful then it returns **STATUS_1394_SUCCESS**. Otherwise an appropriate error status is returned according to the guidelines described in Status Codes Reference.

Remarks

An application must call this function before it calls any other **C1394xxx** function. If **C1394Initialize** is not called first, then any other **C1394xxx** call will fail. The most probable reason why this function could fail is that the 1394-class driver (**UB1394.SYS**) or the user mode API support driver (**UBUMAPI.SYS**) are not loaded.

Calling this function more than once is meaningless, but if the first call is successful, then additional calls will also return **STATUS_1394_SUCCESS**.

C1394Terminate

Terminates 1394 support for the calling application.

```
void C1394Terminate(void);
```

Remarks

An application must call this function before termination in order to perform proper 1394 cleanup. After having called this function, any other **C1394xxx** call will fail.

In Win32 environments, it is not strictly necessary to call this function when terminating, in the same way that in Win32 it is not necessary to close all object handles when terminating. If a user application terminates abnormally, then **UBUMAPI.SYS** will perform the appropriate resource cleanup and the proper operation of the 1394 stack will be unaffected.

C1394GetAdapters

Retrieves the GUIDs for the adapters installed on the system.

```
ULONG C1394GetAdapters(  
    C1394_GUID AdapterGuidArray[],  
    ULONG      uMaxArrayItems  
);
```

Parameters

AdapterGuidArray

A buffer where the GUIDs should be returned into.

uMaxArrayItems

The maximum number of items that can be stored in AdapterGuidArray.

Return Values

The number of GUIDs stored in the buffer.

Remarks

If this function returns zero, then this means that no 1394 miniport driver has been loaded.

If more than one adapters are installed on the system, and *uMaxArrayItems* is less than the number of installed adapters, then only the adapters that appear first in the enumeration will be returned. The actual results of this depend on the order in which the adapters were enumerated by the operating system.

See Also

C1394OpenAdapter

C1394OpenAdapter

Opens a handle to one of the adapter's that have been registered by miniports to the class driver.

```
STATUS_1394 C1394OpenAdapter(
    IN  PC1394_GUID          pAdapterGuid,
    IN  CLIENT_ADAPTER_HANDLE ClientAdapterHandle,
    OUT PC1394_ADAPTER_HANDLE pC1394AdapterHandle
);
```

Parameters

pAdapterGuid

A pointer to a variable of type C1394_GUID, that contains the GUID of the adapter to be opened. If NULL then the default adapter is opened.

*ClientAdapterHandle*⁴⁶

A handle that identifies the adapter to the application.

pC1394AdapterHandle

A pointer to a variable that receives the handle that identifies the adapter to the 1394 stack.

Return Values

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_NOT_FOUND	There is no adapter registered to the class driver with the GUID that is specified by the <i>pAdapterGuid</i> parameter.
STATUS_1394_ALREADY_OPEN	The client driver has already opened the adapter. The variable pointed to by <i>pClassAdapterHandle</i> receives the same C1394_ADAPTER_HANDLE value as the original call for the GUID that returned STATUS_1394_SUCCESS .
STATUS_1394_NO_MEMORY	A required memory allocation failed.
STATUS_1394_INVALID_HANDLE STATUS_1394_UNSUCCESSFUL	These return values indicate some kind of internal error or inconsistency. They should never be returned under normal circumstances.

Remarks

The class driver internally maintains a reference count for each adapter that an application opens. The application must match each successful call to **C1394OpenAdapter** with a call to **C1394CloseAdapter** if it wants to close the adapter correctly and release all associated resources.

For example consider the following scenario:

If modules A and B in an application both open the adapter with GUID X and register notifications, and at a later point module A closes its class adapter handle, then the notifications it has registered are not automatically cleared and the event handlers will still be called. In that case module A should take care to release any resources it allocated on the adapter before closing its handle.

Adapter-related resources owned by the application, for example event notifications and address range mappings, are automatically freed when an adapter is closed with **C1394CloseAdapter**. However it is suggested as good programming practice for applications to first release all their resources on an adapter before closing the adapter handle.

See Also

C1394CloseAdapter, **C1394RegisterNotification**, **C1394UnregisterNotification**, **C1394MapAddressRange**, **C1394UnmapAddressRange**

⁴⁶ All CLIENT_XXX_HANDLE types can be safely type-casted to and from the void* type.

C1394CloseAdapter

Closes a handle to a 1394 adapter.

```
void C1394CloseAdapter(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies the adapter to the 1394 driver stack.

Remarks

C1394CloseAdapter actually decrements the reference count on the adapter. The adapter is closed when the reference count drops to zero. If an application made multiple calls to **C1394OpenAdapter**, then it must match each such call with a call to **C1394CloseAdapter**.

Adapter-related resources owned by the application, for example event notifications and address range mappings, are automatically freed when an adapter gets closed (reference count drops to zero). However it is suggested as good programming practice for applications to first release all their resources on an adapter before closing the adapter handle.

See Also

C1394OpenAdapter, **C1394RegisterNotification**, **C1394UnregisterNotification**, **C1394MapAddressRange**, **C1394UnmapAddressRange**

Outgoing Asynchronous Transactions

C1394ReadNode

Performs a read transaction request to the specified node ID at the specified offset. The operation is performed synchronously, which means that when the function returns the 1394 transaction has been completed.

```
STATUS_1394 C1394ReadNode(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID         Destination,
    IN      C1394_OFFSET          Offset,
    IN      ULONG                 uNumberOfBytes,
    IN      void                  *Buffer,
    IN OUT  C1394_ACK_CODE        *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE   *pResponseCode
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the read request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the read request.

uNumberOfBytes

The number of bytes to read.

Buffer

A buffer containing at least *uNumberOfBytes* available bytes that will receive the results of the read transaction.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DRIVER_INTERNAL_ERROR	This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned. If this error ever appears, then first make sure that the UB drivers that you have installed are the correct version, before checking for any other problem.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.

STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_SPEED_LIMITATION	The size of the read request is too big for the response packet to be received by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the read request is too big for the response to be transmitted on the path from the destination node to the local node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_REQUEST	A broadcast read was requested. Only write transactions can be broadcasted.
STATUS_1394_BUS_RESET	The transaction request was cancelled due to a bus reset. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet read transaction is performed. Otherwise a block read transaction is performed.

Note that it is perfectly legal to transmit a read request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

See Also

C1394ReadNodeAsynch, **C1394WriteNode**, **C1394WriteNodeAsynch**, **C1394LockNode**, **C1394LockNodeAsynch**, **C1394TransmitPackets**

C1394WriteNode

Performs a write transaction request to the specified node ID at the specified offset. The operation is performed in a synchronous manner, which means that the function waits on an event object until the transaction is completed.

```
STATUS_1394 C1394WriteNode(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID        Destination,
    IN      C1394_OFFSET         Offset,
    IN      ULONG                uNumberOfBytes,
    IN      void                 *Buffer,
    IN OUT  C1394_ACK_CODE       *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE *pResponseCode
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the write request.

Destination

The 16-bit NodeID of the destination node. If a broadcast write is requested then the physical ID should be set to 63.

Offset

The 48-bit offset for the write request.

uNumberOfBytes

The number of bytes to write. This parameter can be zero.

Buffer

A buffer containing *uNumberOfBytes* that will be written to the *Destination* at *Offset*. If the *uNumberOfBytes* parameter is zero, then this pointer can be NULL.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Return Values

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that either the request transmission was acknowledged with <i>ack_complete</i> , or it was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code, or it was a broadcast request and it was transmitted successfully. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.
STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SPEED_LIMITATION	The size of the write request is too big to be transmitted by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the write request is too big to be transmitted on the path to the destination node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset).

STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_BUS_RESET	The transaction request was cancelled due to a bus reset. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
<i>other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet write transaction is performed. Otherwise a block write transaction is performed.

Note that it is perfectly legal to transmit a write request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

The request packet is transmitted at the speed returned by **C1394GetMaxSpeedToNode** for the specified destination. If a broadcast packet is being sent, then the transmission speed used is the *broadcast speed*, which is defined as the speed of the slowest device on the bus. If a broadcast transmission at a higher speed is required, then the function **C1394TransmitPackets** should be used. See the remarks section of that function for more information on the related issues.

See Also

C1394ReadNode, **C1394LockNode**, **C1394TransmitPackets**

C1394LockNode

Performs a lock transaction request to the specified node ID at the specified offset. The operation is performed in a synchronous manner, which means that the function waits on an event object until the response packet is received.

```

STATUS_1394 C1394LockNode(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID        Destination,
    IN      C1394_OFFSET         Offset,
    IN      C1394_EXTENDED_TCODE ExtendedTcode,
    IN      ULONG                uArgSize,
    IN      ULONGLONG            UArgValue,
    IN      ULONG                uDataSize,
    IN      ULONGLONG            UDataValue,
    OUT     void                  *DataBuffer ,
    IN OUT  C1394_ACK_CODE        *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE  *pResponseCode
);

```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the lock request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the lock request.

ExtendedTCode

The extended transaction code that describes the lock function to be performed.

uArgSize

The value of *arg_size* to be used in the lock request. This can only be 0, 4 or 8 and the permitted values also depend on the lock function to be executed. For more information see the remarks section.

UArgValue

A 64-bit unsigned long that specifies the *arg_value* to be used in the lock request. If the value of the *uArgSize* parameter is 4, then only the low 32-bits of *UArgValue* are used.

uDataSize

The value of *data_size* to be used in the lock request. This can only be 0, 4 or 8 and the permitted values also depend on the lock function to be executed. For more information see the remarks section.

UDataValue

A 64-bit unsigned long that specifies the *data_value* to be used in the lock request. If the value of the *uDataSize* parameter is 4, then only the low 32-bits of *UDataValue* are used.

DataBuffer

A pointer to a buffer of *uDataSize* bytes that receive the 32-bit or 64-bit data value returned in the lock response. This pointer can be NULL.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Return Values

Value	Description
STATUS_1394_SUCCESS	<p>The lock transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.</p> <p>It is very important to note that a successful <i>lock transaction</i> does not mean a successful <i>lock function</i>. A successful lock transaction ONLY means that a lock response has been received with the <i>resp_complete</i> response code, which in turn implies that there is valid data in the response packet (the old value of the target register). Depending on the lock function requested, the application should compare the old value with the argument value specified in the lock request in order to determine whether the <i>lock function</i> actually succeeded. On the other hand a failed <i>lock transaction</i> implies a failed <i>lock function</i>.</p>
STATUS_1394_TRANSACTION_FAILED	<p>The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.</p>
STATUS_1394_TIMEOUT	<p>The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.</p>
STATUS_1394_NOT_FOUND	<p>The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.</p>
STATUS_1394_DEVICE_BUSY	<p>The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i>, <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.</p>

STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>CI394AdapterHandle</i> is invalid. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset).
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_REQUEST	A broadcast lock was requested. Only write transactions can be transmitted as broadcasts.
STATUS_1394_BUS_RESET	The transaction request was cancelled due to a bus reset. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
<i>other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

The values specified in the *UArgValue* and *UDataValue* parameters are always transmitted in big endian format regardless of the endianness of the platform the class driver is running on. This means that on little endian platforms, the caller does not have to byte swap these values before calling the function. Similarly the data returned are automatically converted to little endian if necessary.

Anyway it is, the caller can always pass a ULONG pointer or a ULONGLONG pointer (depending on whether the call performed a 32-bit or a 64-bit operation) as the *DataBuffer* parameter (possibly with a typecast to *void**), and when the function returns use normal arithmetic to compare the old value returned with the *arg_value* provided in the call.

The code fragment below shows an example of this for a 32-bit compare-swap operation:

```
unsigned long uOldValue;
Status1394 = C1394LockNode( C1394AdapterHandle,
                           Destination,
                           0xFFFF0000230,          /* MAINT_UTILITY */
                           COMPARE_SWAP,
                           4,
                           (ULONGLONG) 0xFF003215, /* arg_value */
                           4,
                           (ULONGLONG) 0xFF0032F0, /* data_value */
                           (void*) &uOldValue,
                           NULL,
                           NULL );

/* We can only check the old value if the transaction request was successful. */
if (STATUS_1394_SUCCESS == Status1394)
{
    if (0xFF003215 == uOldValue)
        printf("COMPARE_SWAP was successful.\n");
}
}
```

If the host CPU on which the program is running is little endian, then in the lock request packet that will be sent to the remote node, the 0xFF003215 and the 0xFF0032F0 values will be both byte swapped so that they are transmitted in big endian format. The same thing will happen when the response packet is received. The old value found in the packet is expected to be in big endian format, so the 1394 stack will byte-swap it before it assigns it to the *uOldValue* variable. This way the application can do all its comparison operations with the intended values, completely forgetting about the issue of endianness.

The following code fragment displays another useful technique that can be used to make both 32-bit or 64-bit lock function calls from a single code fragment depending on the value of the *b64BitLock* control variable:

```
union
{
    UCHAR      Value[8];
    ULONG      uValue;
    ULONGLONG  UValue;
}
Old;

ULONGLONG UArgValue, UDataValue;

// Assign values to all variables that are IN parameters to the function.
...

Status1394 = C1394LockNode( C1394AdapterHandle,
                           Destination,
                           Offset1394,
                           COMPARE_SWAP,
                           (b64BitLock ? 8 : 4),
                           UArgValue,
                           (b64BitLock ? 8 : 4),
                           UDataValue,
                           Old.Value,
                           &AcknowledgeCode,
                           &ResponseCode );
```



```

if (STATUS_1394_SUCCESS == Status1394)
    if (b64BitLock)
    {
        if (Old.UValue == UArgValue)
            printf("64-bit Lock Function Succeeded.");
    }
    else if (Old.uValue == (ULONG)UArgValue)
        printf("32-bit Lock Function Succeeded.");

```

Note: Keep in mind though that the above data conversions are only performed when using functions like **C1394LockNode** and **C1394LockNodeAsynch**. When implementing a CSR that accepts lock transactions, the developer must always remember that the 32-bit or 64-bit values inside the lock request packet are always in big endian.

Caution

Developers must take some care when converting signed 32-bit integer values to unsigned 64-bit integers because the compiler might automatically perform sign extension, which might cause undesirable results.

For example consider the following program (compiled with the MSVC compiler):

```

#include <stdio.h>

void test(unsigned __int64 x)
{
    printf("0x%I64x\n", x);
}

main(void)
{
    long N;
    unsigned long U;

    N = 0x80000000;
    U = 0x80000000;

    test(N);
    test(U);
    test(0x80000000);
    return 0;
}

```

The output of this program is:

```

0xffffffff80000000
0x80000000
0x80000000

```

It is suggested that developers keep this detail in mind when performing 64-bit operations so that they won't have results other than the intended.

See Also

C1394ReadNode, **C1394WriteNode**

C1394CompareSwapNode

Performs a 32-bit or 64-bit compare swap operation.

```
STATUS_1394 C1394CompareSwapNode(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID         Destination,
    IN      C1394_OFFSET          Offset,
    IN      BOOLEAN               b32BitLock,
    IN      ULONGLONG             UArgValue,
    IN      ULONGLONG             UDataValue,
    OUT     void *                pOldValue
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the compare-swap lock transaction request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the lock request.

b32BitLock

TRUE when a 32-bit compare swap is requested, and FALSE when a 64-bit operation is requested. The aliases `Lock32` and `Lock64` have been defined in order to make the code more readable.

UArgValue

A 64-bit unsigned long that specifies the *arg_value* to be used in the lock request. If *b32BitLock* is TRUE, then only the low 32-bits of *UArgValue* are used.

UDataValue

A 64-bit unsigned long that specifies the *data_value* to be used in the lock request. If *b32BitLock* is TRUE, then only the low 32-bits of *UDataValue* are used.

pOldValue

A pointer to a buffer of 4 or 8 bytes that will receive the 32-bit or 64-bit data value returned in the lock response as the *old_value*. This pointer can be NULL if the application does not need this information.

Return Values

Value	Description
STATUS_1394_SUCCESS	The lock transaction completed normally, and the compare-swap operation was successful.
STATUS_1394_LOCK_FAILED	The lock transaction completed normally, but the compare-swap operation failed.
<i>Other</i>	Same meaning as the respective return value of C1394LockNode .

Remarks

This function is simply a wrapper around **C1394LockNode**, in order to simplify the code that needs to be written in order to perform a compare-swap function.

The function treats its arguments in the native format of the host system, but supposes that the target register is implemented in big endian format. For more details see the description of **C1394LockNode** and the sample code under [Allocating a channel number using compare swap](#).

See Also

C1394LockNode

C1394TransmitRaw

Transmits a raw packet through the adapter's asynchronous transmitter. This function executes synchronously, which means that the function waits on an event object until the transmission is completed.

```
STATUS_1394 C1394TransmitRaw(
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,
    IN void *RawPacketBuffer,
    IN ULONG RawPacketBytes,
    IN BOOLEAN bAddHeaderCRC,
    IN BOOLEAN bAddDataCRC,
    IN C1394_SPEED_CODE TransmissionSpeed
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the specified mapping is active.

RawPacketBuffer

A pointer to a buffer containing the raw packet to be transmitted.

RawPacketBytes

The number of bytes contained in *RawPacketBuffer*.

bAddHeaderCRC

A flag indicating whether a header CRC should be calculated and added into the outgoing raw packet.

bAddDataCRC

A flag indicating whether a data CRC should be calculated and added into the outgoing raw packet.

TransmissionSpeed

The speed code identifying the transmission rate to be used for this packet.

Return Values

If the packet is transmitted successfully, then **STATUS_1394_SUCCESS** is returned. Otherwise the function returns the error code indicated by the miniport.

Remarks

If either the header or data CRC should be added, then this CRC will overwrite the contents of the buffer at the appropriate offset. This means that when a packet that contains CRCs is sent then *RawPacketBytes* should contain 4 bytes for the header CRC at the appropriate offset, the necessary number of zero padding bytes after the data payload (if any) and another 4 bytes for the data CRC.

It is important to note that the 1394 stack does not *sniff* packets sent with **C1394TransmitRaw**. This means that if an application sends a valid block read request packet to a remote node, when the response packet comes, the class driver will discard it, since it is completely unaware of the request that was sent earlier. If this function is used to send a valid response to a request that was received by the class driver, then the response will be received by the remote node appropriately, but the local class driver will believe that the response has not been sent and thus it will eventually timeout the request. This means that if another request arrives before the timeout from the same node, with the same transaction label, then the class driver will reject it.

This function is primarily provided for sending PHY packets and for being able to send invalid packets to a device in order to test its proper operation. For example by using **C1394TransmitRaw** application can test a device in the following manners:

- Send 2 read transaction requests with the same transaction label. The 2nd should be rejected by the device's 1394 stack.
- Send two read responses for a given read request.
- Send a read response with less data than those requested in the read request packet.
- Send a quadlet read response packet to a 4-byte block read request and vice versa.
- Send an unsolicited read response packet to the device to see how it reacts to it.
- Send an invalid packet (block write request with actual payload greater than what is reported in the header, invalid header CRC, invalid data CRC, correct header CRC but corrupt header etc).

Practically almost any kind of test can be performed using this function.

An application can use the structures defined in the header file **1394UMAPI.H** if it wants to create an asynchronous primary packet of any type. (*Beta 2 Note: The operation of **C1394TransmitRaw** with these structures has not been yet fully tested at the time of the release, due to schedule pressure. FireAPI Developers will be notified if any update to the drivers is required for these operations to work correctly*).

See Also

C1394TransmitPackets, C1394WriteNode, C1394WriteNodeAsynch, C1394ReadNode, C1394ReadNodeAsynch, C1394LockNode, C1394LockNodeAsynch

C1394ReadNodeAsynch

Performs a read transaction request to the specified node ID at the specified offset. The operation is performed asynchronously, meaning that the function returns immediately after the read request packet has been submitted to the 1394 stack for transmission.

```

C1394_ASYNC_HANDLE C1394ReadNodeAsynch(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID         Destination,
    IN      C1394_OFFSET          Offset,
    IN      ULONG                 uNumberOfBytes,
    IN      void                  *Buffer,
    IN OUT  STATUS_1394          *pStatus1394,
    IN OUT  C1394_ACK_CODE        *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE   *pResponseCode,
    IN      void                  *Context,
    IN      HANDLE                hEvent
);

```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the read request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the read request.

uNumberOfBytes

The number of bytes to read.

Buffer

A buffer containing at least *uNumberOfBytes* available bytes that will receive the results of the read transaction.

pStatus1394

A pointer to a variable of type **STATUS_1394** that will receive the status code of the read operation. This pointer should always point to a valid address.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Context

A context value that the application wishes to associate with the operation. This value will be returned to the application by **C1394CompleteAsynch** when the operation is complete.

hEvent

The handle of the event object that should be signalled when the operation is complete.

Return Values

If any of the parameters are invalid (adapter handle, read buffer, status pointer, ack/resp pointers etc), then the function will immediately return NULL and the variable pointed to by *pStatus1394* will be set to one of the values listed below:

Value	Description
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset).
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers).

If any sort of unexpected error occurs inside the 1394 stack, then the function returns NULL, and stores **STATUS_1394_DRIVER_INTERNAL_ERROR** in the variable pointed to by *pStatus1394*.

This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned. If this error ever appears, then first make sure that the UB drivers that you have installed are the correct version, before checking for any other problem.

If the read request is successfully submitted to the 1394 stack for transmission, then the function returns a non-NULL value of type **C1394_ASYNC_HANDLE** that identifies the request, and also stores **STATUS_1394_PENDING** to the variable pointed to by *pStatus1394*.

When the event object identified by the *hEvent* parameter is set, then the application should first call **C1394CompleteAsynch** with the **C1394_ASYNC_HANDLE** that was returned by **C1394ReadNodeAsynch** in order to complete the operation. After that, the application can proceed and check the variable pointed to by *pStatus1394*, which will be set to one of the values listed in the table below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.
STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_SPEED_LIMITATION	The size of the read request is too big for the response packet to be received by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the read request is too big for the response to be transmitted on the path from the destination node to the local node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_REQUEST	A broadcast read was requested. Only write transactions can be broadcasted.
STATUS_1394_BUS_RESET	The transaction request was cancelled due to a bus reset. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet read transaction is performed. Otherwise a block read transaction is performed.

Note that it is perfectly legal to transmit a read request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

Caution: When making any kind of call that executes asynchronously, then the memory pointed to by pointers passed to this function must remain valid until the asynchronous operation completes. Make certain that this restriction is not violated, otherwise the results may be unpredictable.

See Also

C1394CompleteNodeAsynch, **C1394ReadNode**, **C1394WriteNode**, **C1394WriteNodeAsynch**, **C1394LockNode**, **C1394LockNodeAsynch**

C1394WriteNodeAsynch

Performs a write transaction request to the specified node ID at the specified offset. The operation is performed asynchronously, meaning that the function returns immediately after the write request packet has been submitted to the 1394 stack for transmission.

```

C1394_ASYNC_HANDLE C1394WriteNodeAsynch(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID         Destination,
    IN      C1394_OFFSET          Offset,
    IN      ULONG                 uNumberOfBytes,
    IN      void                  *Buffer,
    IN OUT  STATUS_1394           *pStatus1394,
    IN OUT  C1394_ACK_CODE        *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE   *pResponseCode,
    IN      void                  *Context,
    IN      HANDLE                hEvent
);

```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the write request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the write request.

uNumberOfBytes

The number of bytes to write.

Buffer

A buffer containing at least *uNumberOfBytes* available bytes that contains the data to be sent in the write request.

pStatus1394

A pointer to a variable of type **STATUS_1394** that will receive the status code of the write operation. This pointer should always point to a valid address.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Context

A context value that the application wishes to associate with the operation. This value will be returned to the application by **C1394CompleteAsynch** when the operation is complete.

hEvent

The handle of the event object that should be signalled when the operation is complete.

Return Values

If any of the parameters are invalid (adapter handle, data buffer, status pointer, ack/resp pointers etc), then the function will immediately return NULL and the variable pointed to by *pStatus1394* will be set to one of the values listed below:

Value	Description
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset).
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers).

If any sort of unexpected error occurs inside the 1394 stack, then the function returns NULL, and stores **STATUS_1394_DRIVER_INTERNAL_ERROR** in the variable pointed to by *pStatus1394*.

This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned. If this error ever appears, then first make sure that the UB drivers that you have installed are the correct version, before checking for any other problem. If the problem persists then please submit a bug report.

If the write request is successfully submitted to the 1394 stack for transmission, then the function returns a non-NULL value of type **C1394_ASYNC_HANDLE** that identifies the request, and also stores **STATUS_1394_PENDING** to the variable pointed to by *pStatus1394*.

When the event object identified by the *hEvent* parameter is set, then the application should first call **C1394CompleteAsynch** with the **C1394_ASYNC_HANDLE** that was returned by **C1394WriteNodeAsynch** in order to complete the operation. After that, the application can proceed and check the variable pointed to by *pStatus1394*, which will be set to one of the values listed in the table below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that the request transmission was either directly acknowledged with <i>ack_complete</i> or with <i>ack_pending</i> and then a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.
STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_SPEED_LIMITATION	The size of the write request is too big for the request packet to be transmitted by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the write request is too big for the request packet to be transmitted on the path from the local node to the destination node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_BUS_RESET	The transaction request was cancelled due to a bus reset. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet write transaction is performed. Otherwise a block write transaction is performed. Note that it is perfectly legal to transmit a write request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

The request packet is transmitted at the speed returned by **C1394GetMaxSpeedToNode** for the specified destination. If a broadcast packet is being sent, then the transmission speed used is the *broadcast speed*, which is defined as the speed of the slowest device on the bus. If a broadcast transmission at a higher speed is required, then the function **C1394TransmitPackets** should be used. See the remarks section of that function for more information on the related issues.

Caution: When making any kind of call that executes asynchronously, then the memory pointed to by pointers passed to this function must remain valid until the asynchronous operation completes. Make certain that this restriction is not violated, otherwise the results may be unpredictable.

See Also

C1394CompleteAsynch, **C1394ReadNode**, **C1394ReadNodeAsynch**, **C1394WriteNode**, **C1394LockNode**, **C1394LockNodeAsynch**, **C1394TransmitPackets**

C1394LockNodeAsynch

Performs a read transaction request to the specified node ID at the specified offset. The operation is performed asynchronously, meaning that the function returns immediately after the read request packet has been submitted to the 1394 stack for transmission.

```

C1394_ASYNC_HANDLE C1394LockNodeAsynch(
    IN      C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN      C1394_NODE_ID             Destination,
    IN      C1394_OFFSET              Offset,
    IN      C1394_EXTENDED_TCODE     ExtendedTcode,
    IN      ULONG                    uArgSize,
    IN      ULONGLONG                UArgValue,
    IN      ULONG                    uDataSize,
    IN      ULONGLONG                UDataValue,
    OUT     void                      *DataBuffer,
    IN OUT  STATUS_1394              *pStatus1394,
    IN OUT  C1394_ACK_CODE            *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE      *pResponseCode,
    IN      void                      *Context,
    IN      HANDLE                    hEvent
);

```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the read request.

Destination

The 16-bit NodeID of the destination node.

Offset

The 48-bit offset for the read request.

ExtendedTCode

The 48-bit offset for the read request.

uArgSize

The value of *arg_size* to be used in the lock request. This can only be 0, 4 or 8 and the permitted values also depend on the lock function to be executed. For more information see the remarks section.

UArgValue

A 64-bit unsigned long that specifies the *arg_value* to be used in the lock request. If the value of the *uArgSize* parameter is 4, then only the low 32-bits of *UArgValue* are used.

UDataSize

The value of *data_size* to be used in the lock request. This can only be 0, 4 or 8 and the permitted values also depend on the lock function to be executed. For more information see the remarks section.

UDataValue

A 64-bit unsigned long that specifies the *data_value* to be used in the lock request. If the value of the *uDataSize* parameter is 4, then only the low 32-bits of *UDataValue* are used.

DataBuffer

A pointer to a buffer of *uDataSize* bytes that receive the 32-bit or 64-bit data value returned in the lock response. This pointer can be NULL.

pStatus1394

A pointer to a variable of type STATUS_1394 that will receive the status code of the read operation. This pointer should always point to a valid address.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Context

A context value that the application wishes to associate with the operation. This value will be returned to the application by **C1394CompleteAsynch** when the operation is complete.

hEvent

The handle of the event object that should be signalled when the operation is complete.

Return Values

If any of the parameters are invalid (adapter handle, data buffer, status pointer, ack/resp pointers etc), then the function will immediately return NULL and the variable pointed to by *pStatus1394* will be set to one of the values listed below:

Value	Description
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset).
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, invalid acknowledge/response code pointers, invalid lock function parameters).

If any sort of unexpected error occurs inside the 1394 stack, then the function returns NULL, and stores **STATUS_1394_DRIVER_INTERNAL_ERROR** in the variable pointed to by *pStatus1394*.

This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned.

If this error ever appears, then first make sure that the UB drivers that you have installed have the correct version number combination, before checking for any other problem. If you have the correct set of drivers installed and this problem persists then please submit a bug report.

If the lock request is successfully submitted to the 1394 stack for transmission, then the function returns a non-NULL value of type **C1394_ASYNC_HANDLE** that identifies the request, and also stores **STATUS_1394_PENDING** to the variable pointed to by *pStatus1394*.

When the event object identified by the *hEvent* parameter is set, then the application should first call **C1394CompleteAsynch** with the **C1394_ASYNC_HANDLE** that was returned by **C1394ReadNodeAsynch** in order to complete the processing of the operation.

After that, the application can proceed and check the variable pointed to by *pStatus1394*, which will be set to one of the values listed in the table below:

Value	Description
STATUS_1394_SUCCESS	<p>The lock transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.</p> <p>It is very important to note that a successful <i>lock transaction</i> does not mean a successful <i>lock function</i>. A successful lock transaction ONLY means that a lock response has been received with the <i>resp_complete</i> response code, which in turn implies that there is valid data in the response packet (the old value of the target register). Depending on the lock function requested, the application should compare the old value with the argument value specified in the lock request in order to determine whether the <i>lock function</i> actually succeeded.</p> <p>On the other have a failed <i>lock transaction</i> implies a failed <i>lock function</i>.</p>
STATUS_1394_TRANSACTION_FAILED	<p>The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.</p>
STATUS_1394_TIMEOUT	<p>The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it.</p> <p>No information is returned through the <i>pResponseCode</i> pointer.</p>
STATUS_1394_NOT_FOUND	<p>The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.</p>
STATUS_1394_DEVICE_BUSY	<p>The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i>, <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.</p>
STATUS_1394_UNSUCCESSFUL	<p>The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i>, <i>ack_none</i>, <i>ack_pending</i>, <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated.</p> <p>If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.</p>
STATUS_1394_INVALID_REQUEST	<p>A broadcast lock was requested. Only write transactions can be broadcasted.</p>
STATUS_1394_BUS_RESET	<p>The transaction request was cancelled due to a bus reset.</p> <p>No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.</p>

<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
--------------	---

Remarks

Caution: When making any kind of call that executes asynchronously, then the memory pointed to by pointers passed to this function must remain valid until the asynchronous operation completes. Make certain that this restriction is not violated, otherwise the results may be unpredictable.

Please refer to the comments section of **C1394LockNode** for important information that also apply to the usage of **C1394LockNodeAsynch**, and also some code fragments that demonstrate how to use this function properly.

See Also

C1394CompleteNodeAsynch, C1394LockNode, C1394ReadNode, C1394ReadNodeAsynch, C1394WriteNode, C1394WriteNodeAsynch, C1394TransmitPackets

C1394PingNode

Performs a ping to the specified node ID.

```
STATUS_1394 C1394PingNode(  
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN      C1394_NODE_ID         Destination,  
    IN OUT  PC1394_PHY_PACKET     *pPhyPacket,  
    OUT     ULONG                  *pResponseTime  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the ping request.

Destination

The 16-bit NodeID of the destination node.

pPhyPacket

The phy packet returned by the destination node if the ping was successful. This parameter can be NULL if the PHY packet is not desired.

pResponseTime

A pointer to a ULONG variable that is filled with the response time from the remote node in 49.152 MHz clock counts. This parameter can be NULL if the response time is not required.

Return Values

If the ping was successful **STATUS_1394_SUCCESS** is returned otherwise the function returns an appropriate FireAPI error code.

C1394ReadPHYRegister

Performs a remote PHY register access.

```
STATUS_1394 C1394ReadPHYRegister
(
    IN  C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN  C1394_PHYSICAL_ID        PhysicalID,
    IN  UCHAR                     ReadType,
    IN  UCHAR                     PageSelect,
    IN  UCHAR                     PortSelect,
    IN  UCHAR                     RegisterOffset,
    OUT UCHAR                     *pData
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to perform the remote PHY register read.

PhysicalID

The 6-bit physical ID of the destination node that we wish to access.

ReadType

Identifies the type of remote register access. The supported types are *Base PHY Register Read* (**PHY_EXTENDED_READ_BASE_REGISTER**) and *Paged PHY Register Read* (**PHY_EXTENDED_READ_PAGED_REGISTER**).

PageSelect

When doing a *Base PHY Register Read* this parameter must be zero.

When doing a *Paged PHY Register Read* this parameter identifies the page number of the register that is to be read.

This is a 3-bit field in the remote access PHY packet, so its permitted values are 0-7.

PortSelect

When doing a *Base PHY Register Read* this parameter must be zero.

When doing a *Paged PHY Register Read* this parameter identifies the port whose paged PHY registers are going to be accessed.

This is a 4-bit field in the remote access PHY packet, so its permitted values are 0-15.

RegisterOffset

Identifies the PHY register to access.

This is a 3-bit field in the remote access PHY packet, so its permitted values are 0-7.

pData

Pointer to a UCHAR variable that will receive the byte value of the requested PHY register.

Return Values

Value	Description
STATUS_1394_SUCCESS	The remote PHY register read was completed successfully and the variable pointed to by <i>pData</i> is filled with the byte value read from the remote PHY register.
STATUS_1394_TIMEOUT	The remote access PHY packet was sent but no reply packet was received within the expected timeout (100msec). This is the value returned for example when a remote PHY register read is attempted for a PhysicalID that is not present on the bus.
STATUS_1394_INVALID_HANDLE	<i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A <i>Base PHY Register Read</i> is requested and <i>PageSelect</i> or <i>PortSelect</i> are non-zero, or <i>ReadType</i> does not have one of the supported values, or a <i>Paged PHY Register Read</i> is requested and <i>PageSelect</i> is larger than 7 or <i>PortSelect</i> larger than 15.

Remarks

The PHY registers are separated into the Base PHY registers, which provide information about the PHY chip as a whole, and the Paged PHY registers which provide information for each port separately. Currently there are two pages defined with port information: The Port Status page (0) and the Vendor Identification page (1).

The structure of the Base PHY registers and the Paged PHY registers is described in chapter 15 of the IEEE1394b standard. The binary layout of the Base PHY registers is implemented in the **C1394_PHY_BASE_REGISTERS** structure.

Register offsets and pages that are not defined by the IEEE1394b standard, read as zero.

C1394QueryPhyBaseRegs

Performs a remote PHY base register query.

```
STATUS_1394 C1394QueryPhyBaseRegs
(
    IN  C1394_ADAPTER_HANDLE    a_C1394AdapterHandle,
    IN  UCHAR                   a_BaseRegister,
    IN  UCHAR                   a_CheckBitMask,
    IN  UCHAR                   a_CompareValue,
    IN  BOOLEAN                 a_bNonZeroMatch,
    OUT ULONGLONG               *a_pMatchingPhyIDsMask
);
```

Parameters

a_C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to perform the remote PHY base register query.

a_BaseRegister

The 8-bit register to query.

a_CheckBitMask

Identifies the 8-bit mask to check with the value of the register.

a_CompareValue

A byte value to compare the result of the above check.

a_bNonZeroMatch

If TRUE *a_CompareValue* becomes optional.

a_pMatchingPhyIDsMask

It is a 64-bit integer representing a mask where each bit corresponds to a single node.

Return Values

Value	Description
STATUS_1394_SUCCESS	The query was completed successfully and the variable pointed to by <i>a_pMatchingPhyIDsMask</i> is a bit mask with each bit representing a single node.
STATUS_1394_INVALID_HANDLE	<i>C1394AdapterHandle</i> is invalid.

Remarks

C1394QueryPhyPagedRegs

Performs a remote PHY paged register query.

```

STATUS_1394 C1394QueryPhyBaseRegs
(
    IN  C1394_ADAPTER_HANDLE      a_C1394AdapterHandle,
    IN  UCHAR                     a_PagedRegister,
    IN  UCHAR                     a_CheckBitMask,
    IN  UCHAR                     a_CompareValue,
    IN  BOOLEAN                   a_bNonZeroMatch,
    IN  C1394_PORT_STATUS_ENUM    a_PortStatus,
    OUT ULONG                     a_NodesMatchingPortMaskArray[63],
    OUT ULONG                     a_NodesConnectedPortMaskArray[63]
);

```

Parameters

a_C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to perform the remote PHY paged register query.

a_PagedRegister

The 8-bit register to query.

a_CheckBitMask

Identifies the 8-bit mask to check with the value of the register.

a_CompareValue

A byte value to compare the result of the above check.

a_PortStatus

It is an enumerated value that can take these values:

```

    ConnectedOnly
    DisconnectedOnly
    AllPorts

```

This is used in determining which ports will be scanned when running the query.

a_NodesMatchingPortMaskArray

It is a fixed-size array of unsigned long values each containing a bit mask signifying the port number for which the above comparisons are true. See the remarks section for more information.

a_NodesConnectedPortMaskArray

It is a fixed-size array of unsigned long values each containing a bit mask signifying the port number that is connected. See the remarks section for more information.

Return Values

Value	Description
STATUS_1394_SUCCESS	The query was completed successfully and the two arrays contain the requested masks.
STATUS_1394_SELFID_ERROR	A critical error was found in the self ID packets.
STATUS_1394_UNSUCCESSFUL	There are no connected ports to process and ConnectedOnly was selected or there are no disconnected ports to process and DisconnectedOnly was selected for the scan.
STATUS_1394_CRITICAL_ADAPTER_ERROR	Even though AllPorts was selected, no ports to scan were found. Should never happen.
STATUS_1394_INVALID_HANDLE	<i>C1394AdapterHandle</i> is invalid.

Remarks

The user is responsible for the memory allocation and deallocation of the two arrays.

C1394TransmitPackets

Submits one or more transaction request/response packets for transmission to the 1394 stack. The call is executed asynchronously, which means that the function returns immediately as soon as the packets are passed to the 1394 stack. When the operation is completed, the application is notified through an event object.

```
C1394_ASYNC_HANDLE C1394TransmitPackets(  
    IN  C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN  PFIREAPI_TRANSACTION  *pTransactionArray,  
    IN  ULONG                  uNumberOfPackets,  
    OUT STATUS_1394           *pStatus1394,  
    IN  void                   *Context,  
    IN  HANDLE                 hEvent  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which to perform the operation.

pTransactionArray

An array of pointers to **FIREAPI_TRANSACTION** structures that contain the asynchronous operations to be submitted to the 1394 stack.

uNumberOfPackets

The number of elements in *pTransactionArray*.

pStatus1394

A pointer to a variable of type **STATUS_1394** that will receive the status code of the operation.

Context

A context value that will be returned to the application by **C1394CompleteAsynch** when the asynchronous operation is completed.

hEvent

The handle to a Win32 event object that will be signalled when the operation has been completed.

Return Values

If the packets are successfully submitted to the 1394 stack, then **STATUS_1394_PENDING** is returned through *pStatus1394* and the return value is a non-NULL handle that identifies the asynchronous operation.

If the operation is not successful (most usually because of some invalid parameter), then some other 1394 status code is returned through *pStatus1394*, and the return value of the function provides some additional information as to where exactly the error originates from. In this case the return value can be interpreted as shown in the table below:

Return Value	Description
(C1394_ASYNC_HANDLE) 0xFFFF	The error was detected by UB1394.DLL , before even individually checking out the structures pointed to by <i>pTransactionArray</i> .
(C1394_ASYNC_HANDLE) 0xFFFF0	The error was detected by the UBUMAPI.SYS kernel driver.
(C1394_ASYNC_HANDLE) 0xFFFF1	An unexpected operating-system related error occurred. In the Windows family of operating systems, calling the Win32 function GetLastError will return the operating system error code.
Any other value.	When cast to a ULONG it reflects the index in <i>pTransactionArray</i> of the FIREAPI_TRANSACTION structure that caused the error. This error was detected in UB1394.DLL , and the request was not passed to the kernel drivers at all.

Analytically, the possible 1394 status codes that can be returned on the 1st of the above occasions are:

1394 Status returned for (C1394_ASYNC_HANDLE) 0xFFFF	Description
STATUS_1394_INVALID_HANDLE	<i>C1394AdapterHandle</i> of <i>hEvent</i> are invalid.
STATUS_1394_INVALID_PARAMETER	<i>uNumberOfPacket</i> is zero, or the elements of <i>pTransactionArray</i> are not accessible.
STATUS_1394_NO_MEMORY	A required memory allocation failed.
STATUS_1394_INSUFFICIENT_RESOURCES	More than 256 packets were specified in the call. The 1394 stack will not allow more than 256 packets in one call, in order to prevent uncontrolled operating system resource consumption.

Only two 1394 error codes might be reported by UBUMAPI.SYS:

1394 Status returned for (C1394_ASYNC_HANDLE) 0xFFFF0	Description
STATUS_1394_NO_MEMORY	A kernel mode memory allocation failed.
STATUS_1394_INVALID_PARAMETER	One of the data buffers specified in the call was not accessible.

When the function returns `(C1394_ASYNC_HANDLE)0xFFFF1`, the returned 1394 status code is **STATUS_1394_DRIVER_INTERNAL_ERROR**. This value should normally never be returned.

When a parameter specified in one of the **FIREAPI_TRANSACTION** structures is invalid, then the following 1394 status codes might be returned by the **UB1394.DLL**:

1394 Status returned for (C1394_ASYNC_HANDLE) 0x0 to 0x100	Description
STATUS_1394_INVALID_PARAMETER	<ul style="list-style-type: none"> The Nth element of <i>pTransactionArray</i> was an invalid pointer. The transaction code found in the packet header was invalid. The transmission rate was invalid. The pointer to the data buffer was invalid.
STATUS_1394_INVALID_REQUEST	<ul style="list-style-type: none"> The transaction code found in the packet header identifies a stream packet. The <i>data_length</i> field in the packet header contained an invalid amount of data for the requested operation.
STATUS_1394_SPEED_LIMITATION	The transmission rate was higher than the adapter's maximum transmission rate.
STATUS_1394_SIZE_LIMITATION	The request block size was larger than the maximum asynchronous payload for the requested transmission rate.

Remarks

C1394TransmitPackets can be used to transmit asynchronous request and response packets. It is meant for use by high performance 1394 applications, that are able to produce a lot of data for transmission, and are willing to make multi-packet calls to the 1394 stack, so that the 1394 bus can be utilized as much as possible.

The 1394 bus is very fast, and when applications do not provide data to the 1394 stack fast enough, then the bus remains idle, which means that available bandwidth remains unused, although applications have data ready for transmission.

This is the exact reason why an application would care to use **C1394TransmitPackets**. This function not only does it accept multiple packets (so that it can feed the 1394 stack with a lot of data), but it executes asynchronously. This means that the function returns as soon as the request has been validated and passed to the kernel mode 1394 drivers. As soon as the function returns the application is free to prepare new data for transmission, while the 1394 stack still executes the previous request. This way the application can achieve maximum utilization of the 1394 stack, since it will minimize the amount of time that the stack does not have data to transmit (although the application has).

There is one important issue that developers should keep in mind. This is related to the *transaction label* field (*tl*) in the 1394 packet header. This field is only six bits wide, so at any moment there is a maximum of 64 asynchronous request packets that can be sent to a single destination node (without a response coming back). This means that the 1394 stack will queue the packets and transmit them as soon as transaction labels become available for a destination node.

This means that feeding the 1394 with more data will not improve performance indefinitely, but instead it will lengthen the request queue for the destination node. Application developers should experiment with various settings and then decide which operational settings are effective for their needs.

A call to **C1394TransmitPackets** is treated by the API as a single operation. The 1394 stack will set the event object identified by the *hEvent* handle, in order to notify the application that such an operation is complete when:

- The transmission of all response packets that were found in the *pTransactionArray* has been completed.
- The transmission of all request packets that were found in the *pTransactionArray* has been completed, and the response packets for these requests have been received.

Conceptually **C1394TransmitPackets** can be considered as a function that scans *pTransactionArray* and for each transaction response packet it calls **C1394SendResponse** and for each transaction request packet it calls on of **C1394WriteNode**, **C1394ReadNode** or **C1394LockNode**. When the last item in the array has been processed then the whole operation is considered complete.

The difference between the conceptual model described above and the actual implementation is that:

- The whole task is performed asynchronously by the 1394 drivers, while the application can perform other operations.
- Each transaction request/response is also processed asynchronously by the 1394 stack, which means that after the 1394 stack transmits a request packet it does not wait for its response packet to arrive before proceeding to the next request packet.

It is obvious from the above that upon completion of the operation the application should check each **FIREAPI_TRANSACTION** structure to find out the outcome of each transaction request. The semantics that apply to the 1394 status, the acknowledge code and the response code associated with each **FIREAPI_TRANSACTION** structure are the same as the one that applies to functions **C1394WriteNode**, **C1394ReadNode** and **C1394LockNode** respectively.

For transaction response packets the application should only check the 1394 status to verify that the transmission was carried out successfully.

The most usual reasons why a packet transmission might fail are bad/invalid parameters or a 1394 bus reset. Applications should always check the outcome of transmit operations.

An application is notified about the completion of a multi-packet transmit operation, through the event object identified by the *hEvent* parameter. When this event is set to the signalled state, the application should call the **C1394CompleteAsynch** function with the **C1394_ASYNC_HANDLE** returned by the call to **C1394TransmitPackets**. This is the same procedure that is followed for the other asynchronous functions as well, and is necessary for 1394-stack internal housekeeping. **C1394CompleteAsynch** will return the context value that has been associated by the application with the operation. This can be helpful for applications in order to easily maintain separate context information for each operation.

FIREAPI_TRANSACTION

The **FIREAPI_TRANSACTION** structure is defined as shown below (only the fields that are relevant to applications are shown):

```
typedef struct
{
    // <USED EXCLUSIVELY BY THE APPLICATION>
    // This array can be used by applications to store extra context information.
    void *Context[4];

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // The packet header.
    IN  C1394_PACKET_HEADER          PacketHeader;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // The buffer for the operation.
    /*
    WRITE REQUESTS & ALL RESPONSES
        If PacketHeader.data_length is less than or equal to 24, then the data
        to be sent are read from the 'Bytes' array.
        If PacketHeader.data_length is greater than 24 bytes, then the data
        buffer should be pointed to by the 'pDataBytes' field.

    READ REQUESTS
        If PacketHeader.data_length is less than or equal to 24, then the data returned
        in the read response is copied into the 'Bytes' array.
        If PacketHeader.data_length is greater than 24 bytes, then the data
        buffer should be pointed to by the 'pDataBytes' field.

    LOCK REQUESTS
        The 'Lock' field is used for the parameters of the lock-function .
        In this case the fields are treated in an endianness-neutral manner as in
        C1394LockNode: If running on a little endian processor, these values will
        be byte-swapped by the 1394 stack.
        In cases of 32 bit locks, the ULONGLONG fields are casted to/from ULONG values.
        The 1394 stack will understand if a 32-bit or 64-bit lock is requested, by checking
        the combination of data_length and ExtendedTCode in PacketHeader.
    */
    union
    {
        {
            UCHAR Bytes[24];
            void *pBytes;

            struct
            {
                ULONGLONG  UArgValue;
                ULONGLONG  UDataValue;
                ULONGLONG  UOldValue;
            }
            Lock;
        }
        Buffer;
    }
    // <FILLED BY THE APPLICATION *ONLY* FOR RESPONSE PACKETS>
    // <USED BY THE 1394 STACK>
    // Ignored for transaction request packets.
    // It should be equal to the uBusResetCount value of the C1394_PACKET
    // to which this packet responds to.
    IN  ULONG                uBusResetCount;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // The speed at which this packet will be transmitted.
    IN  C1394_SPEED_CODE     TransmissionSpeed;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION>
    // The request completion status.
    OUT STATUS_1394         Status1394;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION>
    // The acknowledge code returned when the packet was transmitted.
    OUT C1394_ACK_CODE      AcknowledgeCode;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION>
    // The response code. This field is only filled in by the 1394 stack if
    // the packet is a transaction request and a response packet was received for it.
    OUT C1394_RESPONSE_CODE ResponseCode;
}
FIREAPI_TRANSACTION, *PFIREAPI_TRANSACTION;
```

As stated earlier, the values returned for transaction request packets in the *Status1394*, *AcknowledgeCode* and *ResponseCode* fields are the same as those returned through the *pStatus1394*, *pAcknowledgeCode* and *pResponseCode* parameters of **C1394WriteNode**, **C1394ReadNode** and **C1394LockNode** respectively, depending on the type of the request. For response packets the *Status1394* field indicates whether the response packet was transmitted successfully or not.

When sending a lock transaction request, the old-value of the register is returned in the *UOldValue* field. If it is a 32-bit value, then it is cast to the `ULONGLONG` type. The 1394 stack simplifies things for the application by doing the appropriate endianness conversions so that the application can always access the fields of the *Lock* structure in the native endianness of the CPU it is running on.

When sending a lock response packet, the application should use the *Bytes* field to store the 4 or 8 bytes of value to be returned in the response. In this case the value should be stored in big endian format.

Please refer to the comments section of **C1394LockNode** for additional information on the use of lock functions, and also some code fragments that demonstrate the issues that also affect **C1394TransmitPackets**.

With regards to response packets it should be noted here that if **C1394TransmitPackets** is used to send an unsolicited response packet (send a response for which the application has not received a request packet), then this response packet will not be transmitted by the class driver, but it will be rejected with the status code **STATUS_1394_UNSOLICITED_RESPONSE**.

If an application deliberately wants to send an unsolicited response packet, then it should use the provided structures to construct the complete packet and send it with **C1394TransmitRaw**. See the remarks section of **C1394TransmitRaw** for more information on this issue.

See the section that contains the description of the **C1394_PACKET_HEADER** structure for information on the necessary fields/values for the various types of transactions.

In general it is suggested that an application fills the **C1394_PACKET_HEADER** structure with zeros before proceeding to store other information in it. Thus even if the application forgets to set some required field to zero, that will have been done implicitly.

C1394TransmitPackets is the only function that can perform a broadcast transmission at a speed higher than the 'official' broadcast speed. This is defined by the class driver as the speed of the slowest device on the bus, and is returned by **C1394GetMaxSpeedToNode** if the node ID passed as parameter has its physical ID equal to 63.

This is done because 1394 boards cannot receive nor repeat a transmission at a rate higher than their maximum capability, so transmitting a broadcast at a speed higher than that of the slowest device will result in some nodes missing the broadcast. The nodes that will miss the packet depend on the actual bus topology.

However there might be applications where the topology is well known, or the slower devices are known to be leaves and/or they are not intended to receive the broadcast. In this case an application can transmit a broadcast packet at a higher speed. By transmitting at a higher speed the application is capable of transmitting more bytes in a single broadcast packet.

See Also

C1394CompleteAsynch, **C1394ReadNodeAsynch**, **C1394WriteNodeAsynch**,
C1394LockNodeAsynch, **C1394ReadNode**, **C1394WriteNode**, **C1394LockNode**,
C1394_PACKET_HEADER

C1394AcknowledgeBusReset

Updates the internally maintained bus reset count that the 1394 stack associates with each adapter an application opens.

```
void C1394AcknowledgeBusReset(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which to perform the operation.

Remarks

The first time ever this function is called for an adapter it enables the behaviour implemented in FireAPI 1.4 and later with regards to bus resets and asynchronous transaction requests, by making the update of the adapter's bus reset count manual.

See Also

C1394CompleteAsynch

Completes the processing of an asynchronous API operation, that was initiated with a call to **C1394ReadNodeAsynch**, **C1394WriteNodeAsynch**, **C1394LockNodeAsynch** or **C1394TransmitPackets**.

```
void *C1394CompleteAsynch(C1394_ASYNC_HANDLE AsynchHandle);
```

Parameters

AsynchHandle

A handle that identifies an asynchronous API operation, that was returned by a previous call to one of **C1394ReadNodeAsynch**, **C1394WriteNodeAsynch**, **C1394LockNodeAsynch** or **C1394TransmitPackets**.

Return Values

The return value is the context value that the application had specified in the *AsynchContext* parameter in the call to one of the asynchronous functions. If *AsynchHandle* is invalid then the return value is NULL.

Remarks

After the event object associated with an asynchronous API operation has been signalled the application should call **C1394CompleteAsynch** in order to conclude with the processing of the operation and also perform some housekeeping internal to the 1394 stack.

After this function has returned the application can proceed to check the results of the operation.

See Also

C1394ReadNodeAsynch, **C1394WriteNodeAsynch**, **C1394LockNodeAsynch**, **C1394TransmitPackets**

Incoming Asynchronous Transactions

C1394MapAddressRange

Creates a new address range and maps it to an adapter or maps an already existing address range to a new adapter.

```
STATUS_1394 C1394MapAddressRange(
    IN      C1394_ADAPTER_HANDLE          C1394AdapterHandle,
    IN OUT PC1394_RANGE_HANDLE          pC1394RangeHandle,
    OUT HANDLE                          *pStartRequestProcessingEvent,
    IN      PC1394_ADDRESS_RANGE_CHARACTERISTICS pAddressRangeChars
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which to map the address range.

pC1394RangeHandle

A pointer to a variable of type **C1394_RANGE_HANDLE** that holds or will receive a handle that will identify the address range mapping on the adapter. If the destination variable contains the NULL handle value, then the class driver creates a new address range, maps it on the adapter and returns the newly created range handle to the variable pointed to by *pC1394RangeHandle*.

pStartRequestProcessingEvent

Pointer to a handle variable that will receive the handle of an event that is signaled when the first packet for this address range arrives.

pAddressRangeChars

A pointer to a structure of type **C1394_ADDRESS_RANGE_CHARACTERISTICS** that contains information on the address range. If the handle pointed to by *pC1394RangeHandle* is not NULL, then this structure should contain the same address range size as the size that was specified when the address range was first created.

Return Values

Value	Description
STATUS_1394_SUCCESS	The mapping was successfully created.
STATUS_1394_INVALID_HANDLE	<i>C1394AdapterHandle</i> is invalid or <i>ClassDriverHandle</i> is invalid, or the contents of <i>pC1394RangeHandle</i> are invalid
STATUS_1394_CONFLICT	A mapping already exists that overlaps in the 1394 address space with the requested mapping.
STATUS_1394_NO_MEMORY	A required memory allocation failed.
STATUS_1394_INVALID_PARAMETER	A pointer is invalid or one of the values specified in the C1394_ADDRESS_RANGE_CHARACTERISTICS structure are invalid

Remarks

When first creating an address range, don't forget to set to NULL the contents of the variable pointed to by *pC1394RangeHandle*, otherwise the function will fail.

The **C1394_ADDRESS_RANGE_CHARACTERISTICS** characteristics structure is defined as shown below:

```
typedef struct
{
    ////////////////////////////////////////////////////////////////////
    // The base address where the address range is to be mapped.
    // Can be different for different mappings.
    // The combination of BaseAddress and uLength must not span the
    // upper space OHCI boundary (UPPER_OHCI_SPACE_BASE = 0xFFFFF0000000),
    // nor the CSR space base (CSR_SPACE_BASE = 0xFFFFF0000000).
    // Obviously BaseAddress+uLength must be less or equal to 0xFFFFFFFF.
    //
    C1394_OFFSET    BaseAddress;

    ////////////////////////////////////////////////////////////////////
    // The address range length in bytes. Minimum value is 4.
    // The length must be the same in all mappings.
    //
    ULONG    uLength;

    ////////////////////////////////////////////////////////////////////
    // Pointer to the buffer that backs the address range.
    // This can only be non-NULL when the address range is created.
    // If it is NULL in that occasion too, then the fClientTransactions
    // is ignored and all transactions that pass the access rights test
    // are returned to the application through C1394GetNextRequest.
    //
    PVOID    pAddressRangeMemory;

    ////////////////////////////////////////////////////////////////////
    // Access Rights.
    // A binary OR of some of the ACCESS_xxx flags defined above.
    // The access rights can be different for different mappings.
    // When an additional mapping is created zero specifies access rights
    // that were specified when the range was created.
    //
    ULONG    fAccessRights;

    ////////////////////////////////////////////////////////////////////
    // Client Transaction Flags.
    // They describe which requests should be returned to the application
    // through C1394GetNextRequest, instead of being handled automatically
    // by the 1394 stack.
    // These flags are only specified when the address range is created.
    // Additional mappings should specify zero.
    //
    ULONG    fClientTransactions;

    ////////////////////////////////////////////////////////////////////
    // The maximum request queue length.
    // This is only taken into account when the range is created.
    //
    ULONG    uMaxRequestQueueItems;

    ////////////////////////////////////////////////////////////////////
    // The context value that will be stored inside the miniport packet
    // so that the client can find out from which mapping the request
    // came in.
    // This is usually different for each mapping.
    //
    CLIENT_MAPPING_HANDLE    ClientMappingHandle;
}
C1394_ADDRESS_RANGE_CHARACTERISTICS, *PC1394_ADDRESS_RANGE_CHARACTERISTICS;
```

The rules governing the type of access that is required to the memory pointed by *pAddressRangeMemory* are as follows:

- If only block read and/or quadlet read transactions are allowed on the range, then the memory pointed to by *pAddressRangeMemory* and for a length of *uLength* is checked for read access.
- If write or lock transactions are allowed, then the memory is checked for write access.

Upon successful return, *pStartRequestProcessingEvent* points to an auto-reset event that is used to notify the application each time that a new packet has arrived. The application should then repeatedly call **C1394GetNextRequest** to retrieve packets, until the function returns NULL. Note that this event is set only upon the first incoming request, so if the application waits on it again before **C1394GetNextRequest** has returned NULL, then the wait operation will either timeout or last forever⁴⁷.

This event object is created by **C1394OpenAdapter** and deleted by **C1394CloseAdapter**, so the application should not call the Win32 function **CloseHandle** with its handle, unless it has called the Win32 function **DuplicateHandle** first. Otherwise it will not be able to receive a notification about received request packets.

The value of the *uMaxRequestQueueItems* member is not used as a hard limit due to performance considerations. Under various circumstances this limit might be slightly exceeded, but this should not affect any application since this is intended only to prevent excessive memory usage in case some application is slow or something else goes wrong inside the class driver.

The possible values for *fAccessRights* can be a binary OR of the following constants:

Access Flag	Description
ACCESS_QUADLET_READ	Allows a quadlet read transaction request.
ACCESS_BLOCK_READ	Allows a block read transaction request.
ACCESS_QUADLET_WRITE	Allows a quadlet write transaction request.
ACCESS_BLOCK_WRITE	Allows a block write transaction request.
ACCESS_QUADLET_LOCK	Allows a lock transaction request with <i>arg_size</i> equal to 4.
ACCESS_OCTLET_LOCK	Allows a lock transaction request with <i>arg_size</i> equal to 8.
ACCESS_READ_REQUESTS	Binary OR of ACCESS_QUADLET_READ and ACCESS_BLOCK_READ.
ACCESS_WRITE_REQUESTS	Binary OR of ACCESS_QUADLET_WRITE and ACCESS_BLOCK_WRITE.
ACCESS_LOCK_REQUESTS	Binary OR of ACCESS_QUADLET_LOCK and ACCESS_OCTLET_LOCK.
ACCESS_ALL_REQUESTS	Binary OR of ACCESS_READ_REQUESTS and ACCESS_WRITE_REQUESTS and ACCESS_LOCK_REQUESTS.
ACCESS_OLD_UNIFIED_WRITES	Allows the delivery of a write request that was immediately acknowledged with <i>ack_complete</i> (unified transaction), even if that packet was received before the last bus reset.
ACCESS_BROADCAST	Allows broadcast requests to be delivered to the mapping. By default broadcast requests are not delivered to a mapping. This flag can only be specified if one or both of ACCESS_QUADLET_WRITE and ACCESS_BLOCK_WRITE are also specified.
ACCESS_BROADCAST_LOOPBACK	Allows loopbacked broadcast requests to be delivered to the mapping. By default loopbacked broadcast requests are not delivered to a mapping (see note below). This flag can only be specified if ACCESS_BROADCAST is also specified.

⁴⁷ Depending on the timeout value specified in the call to the Win32 functions **WaitForSingleObject** or **WaitForMultipleObjects** (or any other wrapper to these functions).

NOTE

In FireAPI broadcast transmissions are automatically loopbacked by the 1394 stack. This means that the node that is sending a broadcast packet will also receive this packet.

Often applications use broadcasts to locate or identify themselves to their peers. However it is often the case that the peer application can be running on the same node.

Broadcast loopback can be controlled on a *per address range* basis. The default access rights of an address range do not allow it to accept loopbacked broadcasts. Applications must specify the **ACCESS_BROADCAST_LOOPBACK** flag in order to be able to receive loopbacked broadcasts.

The possible values for *fClientTransactions* can be a binary OR of the following constants:

Flag	Description
CLIENT_QUADLET_READ	Passes quadlet read transaction requests.
CLIENT_BLOCK_READ	Passes block read transaction requests.
CLIENT_QUADLET_WRITE	Passes quadlet write transaction requests.
CLIENT_BLOCK_WRITE	Passes block write transaction requests.
CLIENT_QUADLET_LOCK	Passes a lock transaction request with <i>arg_size</i> equal to 4.
CLIENT_OCTLET_LOCK	Passes a lock transaction request with <i>arg_size</i> equal to 8.
CLIENT_READ_REQUESTS	Binary OR of CLIENT_QUADLET_READ and CLIENT_BLOCK_READ.
CLIENT_WRITE_REQUESTS	Binary OR of CLIENT_QUADLET_WRITE and CLIENT_BLOCK_WRITE.
CLIENT_LOCK_REQUESTS	Binary OR of CLIENT_QUADLET_LOCK and CLIENT_OCTLET_LOCK.
CLIENT_ALL_REQUESTS	Binary OR of CLIENT_READ_REQUESTS and CLIENT_WRITE_REQUESTS and CLIENT_LOCK_REQUESTS.

See Also

C1394UnmapAddressRange

C1394UnmapAddressRange

Removes an address range mapping from an adapter, and if that was the last mapping for the address range, also destroys the address range.

```
void C1394UnmapAddressRange(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    IN C1394_RANGE_HANDLE C1394RangeHandle,  
    IN C1394_OFFSET BaseOffset  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the specified mapping is active.

C1394RangeHandle

A handle identifying to the class driver an address range.

BaseOffset

A base offset to which the address range is mapped.

Remarks

There are some serious issues to be taken care of by the class driver when removing an address range. First of all the application should not have any pending transaction requests that belong to the mapping, because these requests hold a reference on the mapping object.

If an application unmaps an address range while it holds pending transaction requests then the class driver makes the address range mapping inactive but does not remove it until the application has completed the requests.

See Also

C1394MapAddressRange

C1394GetNextRequest

Pops the next application-handled request packet from an address range's request queue.

```
PC1394_PACKET C1394GetNextRequest(
    IN C1394_RANGE_HANDLE C1394RangeHandle
);
```

Parameters

C1394RangeHandle

A handle identifying to the class driver the address range whose queue should be checked.

Return Values

If the queue is empty or does not contain any items that the application should handle, the return value is NULL. Otherwise the first application-handled packet is removed and a pointer to it is returned.

Remarks

If the request is found on top of the queue that should be automatically handled by the class driver, this is done and then the next request in the queue is checked, until either the queue empties or a application -handled request is found.

It is important to note that this function will not return a packet that was queued before the last bus reset. The *uBusResetCount* field is compared against the value returned by **C1394GetBusResetCount** and will immediately complete a request packet that does not match the current bus reset count.

The only exception to this rule can be for write requests that have already been acknowledged with *ack_complete*. If the application specified the **ACCESS_OLD_UNIFIED_WRITES** flag when creating the mapping. In that case such packet are delivered regardless of their bus reset count. This might be useful in applications where the information found inside the packet is bus-reset independent and it is enough for the receiving node to correctly process the packet.

The **C1394_PACKET** structure is defined as follows:

```
typedef struct
{
    // The translated packet header.
    C1394_PACKET_HEADER    PacketHeader;

    // The class adapter handle.
    C1394_ADAPTER_HANDLE    ClassAdapterHandle;

    // The client mapping handle, that identifies the mapping
    // through which the packet was received.
    CLIENT_MAPPING_HANDLE    ClientMappingHandle;

    // The Bus Reset Count at the time the packet was received.
    ULONG                    uBusResetCount;

    // The amount of data in the whole packet.
    ULONG                    uByteCount;

    // The acknowledge code of the packet.
    C1394_ACK_CODE          AcknowledgeCode;

    // The total size of the packet buffer in bytes.
    // VALID: INCOMING & OUTGOING packets.
    ULONG                    uPacketBufferSize;

    // The buffer that contains the packet.
    // This contains both the header and payload.
    // VALID: INCOMING & OUTGOING packets.
    UCHAR                    *PacketBuffer;
}
C1394_PACKET, *PC1394_PACKET;
```

The **C1394_NODE_ID** structure is defined as follows:

```
typedef union
{
    USHORT Value;

    struct
    {
        USHORT PhysicalID:6;
        USHORT BusID:10;
    };
}
C1394_NODE_ID, *PC1394_NODE_ID;
```

The **C1394_TIMESTAMP** structure is defined as follows:

```
typedef union
{
    unsigned short Value;

    struct
    {
        unsigned short Counts:13;
        unsigned short Seconds:3;
    };
}
C1394_TIMESTAMP, *PC1394_TIMESTAMP;
```

The **C1394_OFFSET** data type is a 64-bit unsigned integer type. In Windows NT this is typedefed as ULONGLONG, so that 64-bit integer arithmetic can be used to conveniently perform any kind of operations with 1394 address space offsets.

See Also

C1394ServiceTransactionRequest, **C1394CompletePackets**

C1394SendErrorResponse

Generates and transmits a response packet for a transaction request with a given error response code.

```
void C1394SendErrorResponse(  
    IN PC1394_PACKET          pPacket,  
    IN C1394_RESPONSE_CODE    ResponseCode  
);
```

Parameters

pPacket

The transaction request packet for which to send a response. This is used so that the class driver can find out information necessary to create the response, like the destination node, transaction label etc.

ResponseCode

The response code to put in the response packet.

Remarks

This function calls **C1394SendResponse** with a NULL data buffer and a data byte count of 0. The **C1394_PACKET** structure contains the adapter handle from which the packet was received, and **C1394SendResponse** uses it to send the response packet.

See Also

C1394SendResponse

C1394SendResponse

Generates and transmits a response packet for a transaction request with a given response code and data buffer for the response data.

```
void C1394SendResponse(  
    IN PC1394_PACKET          pPacket,  
    IN C1394_RESPONSE_CODE    ResponseCode,  
    IN void                   *pResponseDataBuffer,  
    IN ULONG                   uResponseDataBytes  
);
```

Parameters

pPacket

The transaction request packet for which to send a response. This is used so that the class driver can find out information necessary to create the response, like the destination node, transaction label etc.

ResponseCode

The response code to put in the response packet.

pResponseDataBuffer

A pointer to a buffer containing *uResponseDataBytes* number of bytes that will be put in the response packet. If *uResponseDataBytes* is zero, then this parameter can be NULL.

uResponseDataBytes

The number of bytes that will be put in the data payload of the response packet.

Remarks

This function will also complete the packet pointed to by *pPacket*, by calling **C1394CompletePacket**, after it has prepared the appropriate response packet.

The memory pointed to by *pResponseDataBuffer* is available as soon as this function returns.

See Also

C1394SendErrorResponse

C1394ServiceTransactionRequest

Performs on address range memory the actions that are necessary for this request and produces and transmits the appropriate response packet. This function can only be called for address ranges whose backing memory is non-NULL (when the address range was created the *pAddressRangeMemory* field of the **C1394_ADDRESS_RANGE_CHARACTERISTICS** structure was not NULL).

```
void C1394ServiceTransactionRequest(  
    IN PC1394_PACKET pPacket  
);
```

Parameters

pPacket

A pointer to a **C1394_PACKET** structure that should be serviced by the class driver.

Remarks

C1394ServiceTransactionRequest will do any necessary updates to the address range memory, and then it will generate and transmit a response packet. It will also complete the received packet by calling **C1394CompletePacket**.

See Also

C1394MapAddressRange

C1394CompletePacket

Completes the processing of a received packet, and frees any associated resources allocated by the 1394-stack.

```
void C1394CompletePacket( IN PC1394_PACKET  pPacket );
```

Parameters

pPacket

The received packet whose processing has been completed.

C1394CompletePackets

Completes the processing of one or more received packets, and frees any associated resources allocated by the 1394-stack.

```
void C1394CompletePackets(
    IN PC1394_PACKET  *pPacketArray,
    IN ULONG          uPacketsCompleted
);
```

Parameters

pPacketArray

An array of pointers to received packets for which processing has been completed.

uPacketsCompleted

The number of miniport packet pointers in *pPacketArray*.

Remarks

After calling this function, the caller should not use the packet pointers again.

C1394CompletePacket is actually a function that calls **C1394CompletePackets** with the *uPacketsCompleted* parameter set to 1. This was made as a function instead of a macro because it had to be exported from **UB1394.DLL** and be available to languages other than C/C++. However **FireAPI.h** defined the **C1394CompletePacket_** macro as a call to **C1394CompletePackets** so C/C++ programmers can use this instead of the **C1394CompletePacket** function.

See Also

C1394GetNextRequest, **C1394SendResponse**

Device Handle Functions

C1394OpenDevice

Opens a handle to one of the 1394 devices on the local bus.

```
STATUS_1394 C1394OpenDevice(
    IN  C1394_ADAPTER_HANDLE    C1394AdapterHandle,
    IN  C1394_GUID              *pC1394Guid,
    OUT DEVICE_HANDLE           *pDeviceHandle
);
```

Parameters

C1394AdapterHandle

A handle that identifies the 1394 stack the adapter on which to try and open the device handle. Basically it identifies the 1394 bus where the device is physically located. All read and write asynchronous transactions to the device will be sent through this adapter.

pC1394Guid

A pointer to a *C1394_GUID* variable containing the GUID of the device the caller wants to open.

pDeviceHandle

A pointer to a *DEVICE_HANDLE* variable that will receive the handle to the device if the function call completes successfully.

Return Values

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_DEVICE_NOT_FOUND	There is no device with the specified GUID on the local network.
STATUS_1394_NO_MEMORY	A required memory allocation failed.
STATUS_1394_INVALID_HANDLE STATUS_1394_INVALID_PARAMETER	At least one of the provided parameters is invalid.
STATUS_1394_DRIVER_INTERNAL_ERROR	This should not happen unless there is a problem or bug in the Class Driver

Remarks

If more than one 1394 adapters are installed in the computer and each one of them is connected to a *different* 1394 bus then the **C1394AdapterHandle** parameter provided to this function identifies the 1394 bus where the device of interest is located.

If the 1394 adapters are connected to the *same* local bus, then any of the adapter handles can be used to open the device. All asynchronous traffic to the device will be routed through the adapter that was selected.

Additionally, the Class Driver will **not** automatically relocate the device handle to a different adapter in case the single 1394 bus is partitioned by disconnecting some of the cables.

For example, assume there are two 1394 adapters, adapter A and adapter B, connected directly to each other, and two cameras, C connected to adapter A and D connected to adapter B. We open a device handle to camera D from adapter A. If we break the single 1394 bus by disconnecting adapter A from B, then camera D is not accessible from A, but it is accessible from B. The Class Driver will **not** perform an automatic relocation of the device handle from adapter A to adapter B.

See Also

C1394CloseDevice

C1394CloseDevice

Closes a handle to a previously opened device.

```
void C1394CloseDevice(  
    IN  DEVICE_HANDLE    DeviceHandle,  
);
```

Parameters

DeviceHandle

A variable of type *DEVICE_HANDLE* containing the handle to be closed.

Remarks

C1394CloseDevice closes the device handle and releases any internally allocated resources. Passing an invalid value for *DeviceHandle* will not crash the application.

See Also

C1394OpenDevice

C1394ReadDevice

Sends a read transaction request to the specified device at the specified offset. The operation is performed synchronously, which means that when the function returns the 1394 transaction has been completed.

```
STATUS_1394 C1394ReadDevice(
    IN     DEVICE_HANDLE      DeviceHandle,
    IN     C1394_OFFSET      Offset,
    IN     ULONG              uNumberOfBytes,
    IN     void               *Buffer,
    IN OUT C1394_ACK_CODE     *pAcknowledgeCode,
    IN OUT C1394_RESPONSE_CODE *pResponseCode
);
```

Parameters

DeviceHandle

A handle identifying to the 1394 stack the device to read data from.

Offset

The 48-bit target offset for the read request.

uNumberOfBytes

The number of bytes to read.

Buffer

A buffer containing at least *uNumberOfBytes* writable bytes of memory that will receive the data in the read response.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DRIVER_INTERNAL_ERROR	This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned. If this error ever appears, then first make sure that the UB drivers that you have installed are the correct version, before checking for any other problem.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DEVICE_NOT_FOUND	The specified device does not exist on the local bus anymore.
STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.

STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_SPEED_LIMITATION	The size of the read request is too big for the response packet to be received by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the read request is too big for the response to be transmitted on the path from the destination node to the local node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_REQUEST	A broadcast read was requested. Only write transactions can be broadcasted.
<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet read transaction is performed. Otherwise a block read transaction is performed.

Note that it is perfectly legal to transmit a read request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

See Also

C1394ReadNode, **C1394WriteDevice**

C1394WriteDevice

Sends a write transaction request to the specified device at the specified offset. The operation is performed synchronously, which means that when the function returns the 1394 transaction has been completed.

```
STATUS_1394 C1394WriteDevice(
    IN     DEVICE_HANDLE      DeviceHandle,
    IN     C1394_OFFSET      Offset,
    IN     ULONG              uNumberOfBytes,
    IN     void                *Buffer,
    IN OUT C1394_ACK_CODE     *pAcknowledgeCode,
    IN OUT C1394_RESPONSE_CODE *pResponseCode
);
```

Parameters

DeviceHandle

A handle identifying to the 1394 stack the device to write data to.

Offset

The 48-bit target offset for the write request.

uNumberOfBytes

The number of bytes to write.

Buffer

A buffer containing *uNumberOfBytes* bytes of data that will be written to the *Destination* at *Offset*. If the *uNumberOfBytes* parameter is zero, then this pointer can be NULL.

pAcknowledgeCode

An optional pointer to a variable that will receive the acknowledge code that was returned when the transaction request packet was transmitted. The acknowledge code is only returned if the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

pResponseCode

An optional pointer to a variable that will receive the response code contained in the response packet. The response code is only returned if a response packet was received and the transaction did not complete successfully. This pointer can be NULL if the caller is not interested in this information.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully. This means that the request transmission was acknowledged with <i>ack_pending</i> and a response packet was received that contained the <i>resp_complete</i> response code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_TRANSACTION_FAILED	The transmission of the transaction request was completed successfully, and a response packet was received which indicated that the transaction did not complete successfully. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_complete</i> is returned through it, and if the <i>pResponseCode</i> pointer is provided the error response code found in the response packet is returned through it.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_OFFSET	The 1394 address space offset specified or the offset + argument size are invalid (greater than the highest 48-bit offset). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or acknowledge/response code pointers). No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DRIVER_INTERNAL_ERROR	This error generally indicates some sort of serious problem with the 1394 stack (unstable situation, internal bug etc), and should normally never be returned. If this error ever appears, then first make sure that the UB drivers that you have installed are the correct version, before checking for any other problem.
STATUS_1394_TIMEOUT	The transmission was completed successfully, <i>ack_pending</i> was returned but no response was received within the split transaction timeout. If the <i>pAcknowledgeCode</i> pointer is provided then <i>ack_pending</i> is returned through it. No information is returned through the <i>pResponseCode</i> pointer.
STATUS_1394_NOT_FOUND	The transmission was completed successfully, but no acknowledge was returned, so the miniport indicated the <i>ack_missing</i> acknowledge code. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_DEVICE_NOT_FOUND	The specified device does not exist on the local bus anymore.
STATUS_1394_DEVICE_BUSY	The transmission was completed successfully, but a busy acknowledge code was returned even after the specified retry protocol was executed. If the <i>pAcknowledgeCode</i> pointer is provided then one of the <i>ack_busy_X</i> , <i>ack_busy_A</i> or <i>ack_busy_B</i> acknowledge codes is returned through it. No response code information will be returned.

STATUS_1394_UNSUCCESSFUL	The transmission was completed successfully, but some acknowledge other than <i>ack_complete</i> , <i>ack_none</i> , <i>ack_pending</i> , <i>ack_busy_(XAB)</i> and <i>ack_missing</i> was indicated. If the <i>pAcknowledgeCode</i> pointer is provided then the function will return the acknowledge code that has been received through this pointer. No response code information will be returned.
STATUS_1394_SPEED_LIMITATION	The size of the read request is too big for the response packet to be received by the adapter at its maximum speed. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_SIZE_LIMITATION	The size of the read request is too big for the response to be transmitted on the path from the destination node to the local node. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.
STATUS_1394_INVALID_REQUEST	A broadcast read was requested. Only write transactions can be broadcasted.
<i>Other</i>	The transmission of the transaction request packet was not completed successfully due to some error on the miniport (hardware error, bus reset etc). The status returned is the same status that the miniport returned. No information is returned through the <i>pAcknowledgeCode</i> and <i>pResponseCode</i> pointers.

Remarks

If the *uNumberOfBytes* parameter equals 4, and the destination offset is quadlet aligned, then a quadlet write transaction is performed. Otherwise a block write transaction is performed.

Note that it is perfectly legal to transmit a write request of zero bytes. Some applications use such requests and their response codes as a means of communicating commands and state information.

The request packet is transmitted at the speed returned by **C1394GetMaxSpeedToNode** for the specified destination. If a broadcast packet is being sent, then the transmission speed used is the *broadcast speed*, which is defined as the speed of the slowest device on the bus. If a broadcast transmission at a higher speed is required, then the function **C1394TransmitPackets** should be used. See the remarks section of that function for more information on the related issues.

See Also

C1394WriteNode, C1394ReadDevice

C1394GetDeviceNodeId

Returns the current NodeID of a device.

```
STATUS_1394 C1394GetDeviceNodeId(
    IN  DEVICE_HANDLE          DeviceHandle,
    OUT C1394_NODE_ID         *pDeviceNodeId
);
```

Parameters

DeviceHandle

A handle identifying the destination device to the 1394 stack.

pDeviceNodeId

A pointer to a *C1394_NODE_ID* variable that will receive the current NodeID of the device.

Return Values

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_DEVICE_NOT_FOUND	There is no device with the specified GUID on the local network.
STATUS_1394_INVALID_HANDLE	At least one of the provided parameters are invalid.
STATUS_1394_DRIVER_INTERNAL_ERROR	This should not happen unless there is a problem or bug in the class Driver

See Also

C1394OpenDevice

Retry Functions

C1394MayRetryTransaction

Returns whether it makes sense to retry a failed transaction to a device.

```
BOOL C1394MayRetryTransaction(  
    IN const C1394_NODE_ID      Destination,  
    IN const C1394_OFFSET      Offset,  
    IN const STATUS_1394      Status1394,  
    IN C1394_RESPONSE_CODE     RespCode  
);
```

Parameters

Destination

The Node ID of the device that was the target of the failed transaction. This is used only for debugging purposes (for generating useful debugging messages in the debug version of UB1394.DLL).

Offset

The target offset of the failed transaction. This is used only for debugging purposes (for generating useful debugging messages in the debug version of UB1394.DLL).

Status1394

The status code of the failed transaction.

RespCode

The response code of the failed transaction, if available.

Remarks

The logic behind this function is independent of the adapter, the target node and the type of transaction. For example, if a transaction fails with the error code **STATUS_1394_NOT_FOUND** then there is absolutely no conceivable way that a retry could succeed.

The **C1394MayRetryTransaction** function does not keep internally track of any retry counters for the destination nodes and does not base its results on some number of failed retries. The number of retries is decided by the callers of **C1394MayRetryTransaction**. The only thing that **C1394MayRetryTransaction** returns is whether it technically makes sense to retry the failed transaction.

C1394RetryReadNodeInQuads

Reads a number of bytes from a device using quadlet read operations with automatically adjusted micro-delays between successive transactions.

```
STATUS_1394 C1394RetryReadNodeInQuads
(
    IN        C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN        C1394_NODE_ID         Destination,
    IN        C1394_OFFSET          Offset,
    IN        ULONG                 uNumberOfBytes,
    IN        void                  *Buffer,
    IN OUT    C1394_ACK_CODE         *pAcknowledgeCode,
    IN OUT    C1394_RESPONSE_CODE    *pResponseCode
);
```

Parameters

Same as the parameters of **C1394ReadNode**, with the restrictions that:

- *uNumberOfBytes* must be a multiple of quadlet size (4 bytes).
- *Offset* must be quadlet aligned.

Return Values

Same as the return values of **C1394ReadNode**.

Remarks

The **C1394RetryReadNodeInQuads** function inserts micro-delays between successive calls to **C1394ReadNode** in order to minimize the number of failed transaction requests due to busy acknowledges or resource conflicts. The micro-delays are dependent on the time it took the device to respond to the previous transaction request.

C1394RetryReadNodeExInQuads

Reads a number of bytes from a device using quadlet read operations with automatically adjusted micro-delays between successive transactions.

```
STATUS_1394 C1394RetryReadNodeExInQuads
(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      C1394_NODE_ID         Destination,
    IN      C1394_OFFSET          Offset,
    IN      ULONG                 uNumberOfBytes,
    IN      void                  *Buffer,
    IN      ULONG                 uClientBusResetCount,
    IN OUT  C1394_ACK_CODE        *pAcknowledgeCode,
    IN OUT  C1394_RESPONSE_CODE   *pResponseCode
);
```

Parameters

Same as the parameters of **C1394ReadNodeEx**, with the restrictions that:

- *uNumberOfBytes* must be a multiple of quadlet size (4 bytes).
- *Offset* must be quadlet aligned.

Return Values

Same as the return values of **C1394ReadNodeEx**.

Remarks

The **C1394RetryReadNodeExInQuads** function inserts micro-delays between successive calls to **C1394ReadNodeEx** in order to minimize the number of failed transaction requests due to busy acknowledges or resource conflicts. The micro-delays are dependent on the time it took the device to respond to the previous transaction request.

C1394RetryReadDeviceInQuads

Reads a number of bytes from a device using quadlet read operations with automatically adjusted micro-delays between successive transactions.

```
STATUS_1394 C1394RetryReadDeviceInQuads(  
    IN      DEVICE_HANDLE      DeviceHandle,  
    IN      C1394_OFFSET      Offset,  
    IN      ULONG              uNumberOfBytes,  
    IN      void               *Buffer,  
    IN OUT  C1394_ACK_CODE     *pAcknowledgeCode,  
    IN OUT  C1394_RESPONSE_CODE *pResponseCode  
);
```

Parameters

Same as the parameters of **C1394ReadDevice**, with the restrictions that:

- *uNumberOfBytes* must be a multiple of quadlet size (4 bytes).
- *Offset* must be quadlet aligned.

Return Values

Same as the return values of **C1394ReadDevice**.

Remarks

The **C1394RetryReadDeviceInQuads** function inserts micro-delays between successive calls to **C1394ReadDevice** in order to minimize the number of failed transaction requests due to busy acknowledges or resource conflicts. The micro-delays are dependent on the time it took the device to respond to the previous transaction request.

C1394RetryWriteDeviceInQuads

Writes a number of bytes to a device using quadlet write operations with automatically adjusted micro-delays between successive transactions.

```
STATUS_1394 C1394RetryWriteDeviceInQuads(  
    IN      DEVICE_HANDLE      DeviceHandle,  
    IN      C1394_OFFSET      Offset,  
    IN      ULONG              uNumberOfBytes,  
    IN      void               *Buffer,  
    IN OUT  C1394_ACK_CODE     *pAcknowledgeCode,  
    IN OUT  C1394_RESPONSE_CODE *pResponseCode  
);
```

Parameters

Same as the parameters of **C1394WriteDevice**, with the restrictions that:

- *uNumberOfBytes* must be a multiple of quadlet size (4 bytes).
- *Offset* must be quadlet aligned.

Return Values

Same as the return values of **C1394WriteDevice**.

Remarks

The **C1394RetryWriteDeviceInQuads** function inserts micro-delays between successive calls to **C1394WriteDevice** in order to minimize the number of failed transaction requests due to busy acknowledges or resource conflicts. The micro-delays are dependent on the time it took the device to respond to the previous transaction request.

Isochronous Processing

C1394OpenAdapterChannel

Requests the class driver to setup a new DMA channel on the adapter for use in stream operations.

```

STATUS_1394 C1394OpenAdapterChannel(
    IN  C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    OUT C1394_CHANNEL_HANDLE      *pC1394ChannelHandle,
    OUT HANDLE                     *pStartProcessingEvent,
    IN  CLIENT_CHANNEL_HANDLE      ClientChannelHandle,
    IN  FIREAPI_CHANNEL_PARAMETERS *pChannelParameters
);
    
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which to try to open a DMA channel.

pC1394ChannelHandle

A pointer to a variable that will receive the handle that will identify the adapter channel to the 1394 stack, if the operation is successful. This pointer should point to a valid memory location, otherwise the function will fail.

pStartProcessingEvent

A pointer to a handle variable that will receive the event object handle that will be used to notify the application when requests are completed. This pointer should point to a valid memory location, otherwise the function will fail.

ClientChannelHandle

A handle that will be identifying the adapter channel to the application, if the operation is successful.

pChannelParameters

A pointer to a structure that describes the type of operations that this DMA channel will be used for, and also specifies various related parameters.

Return Values

Value	Description
STATUS_1394_SUCCESS	The 1394 stack has a free DMA channel on the adapter that it can allocate for isochronous operations, and the channel has been allocated. The variable pointed to by <i>pC1394ChannelHandle</i> will be filled with a handle value that will identify the adapter channel to the 1394 stack.
STATUS_1394_INSUFFICIENT_RESOURCES	There are no free DMA channels on the adapter.
STATUS_1394_NO_MEMORY	A necessary memory allocation failed.
STATUS_1394_NOT_SUPPORTED	The required functionality is not available on the adapter.
STATUS_1394_NOT_IMPLEMENTED	The required functionality is supported by the adapter but not yet implemented by drivers.
STATUS_1394_INVALID_HANDLE	<i>ClassAdapterHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	Either <i>pClassChannelHandle</i> or <i>pChannelParameters</i> is an invalid pointer, or the values specified in the FIREAPI_CHANNEL_PARAMETERS structure are invalid.

STATUS_1394_DRIVER_INTERNAL_ERROR

An unexpected error occurred.

Remarks

Depending on the capabilities of the adapter, the class driver will check whether any DMA channels are available, and if so reserve one and set it up for the type of operation requested by the user.

The **FIREAPI_CHANNEL_PARAMETERS** structure is defined as shown below:

```
typedef struct
{
    // Identification tag. Must be set to TAG_FIREAPI_CHANNEL_PARAMETERS
    ULONG          Tag;

    // The type of adapter channel to open.
    ULONG          AdapterChannelType;

    // The type-specific parameters.
    union
    {
        //////////////////////////////////////
        // Isochronous Receive Channel.
        struct
        {
            // Flags that affect the operation of this adapter-channel.
            ULONG          fAdapterChannelOptions;

            // The maximum number of packets per isochronous request.
            // Used if the PACKETS_PER_REQUEST flag is specified in
            // fAdapterChannelOptions.
            ULONG          uMaxPacketsPerRequest;
        }
        IsochReceive;

        //////////////////////////////////////
        // Isochronous Transmit Channel.
        struct
        {
            // Flags that affect the operation of this adapter-channel.
            ULONG          fAdapterChannelOptions;

            // The maximum number of packets per isochronous request.
            // Used if the PACKETS_PER_REQUEST flag is specified in
            // fAdapterChannelOptions.
            ULONG          uMaxPacketsPerRequest;
        }
        IsochTransmit;
    };
};

FIREAPI_CHANNEL_PARAMETERS, *PFIREAPI_CHANNEL_PARAMETERS;
```

The *Tag* field must always be set to **TAG_FIREAPI_CHANNEL_PARAMETERS** otherwise the call will fail immediately and return **STATUS_1394_INVALID_PARAMETER**.

The possible values for *AdapterChannelType* are `ChannelIsochReceive` and `ChannelIsochTransmit`. Depending on the value of this field the appropriate sub-structure of the union is being used.

The currently defined values for the *fAdapterChannelOptions* field are:

Value	Description
PACKETS_PER_REQUEST	<p>The value found in the <i>uMaxPacketsPerRequest</i> should be used in order to specify the maximum number of packets per request that the client intends to use on this adapter channel.</p> <p>If this flag is not specified, then the class driver will use the default value. An application can find out about this value by specifying the OID_ISO_REQUEST_PACKETS object identifier in a call to C1394QueryInformation⁴⁸.</p> <p>There is a hard limit on the maximum value of <i>uMaxPacketsPerRequest</i>, that can be configured from the registry. Clients can find out about this value by specifying the OID_MAX_ISO_REQUEST_PACKETS object identifier in a call to C1394QueryInformation.</p>
ALLOCATE_MAX_REQUESTS	<p>When this flag is specified, the miniport will try to allocate the maximum number of requests to use for isochronous operations, if the memory is limited and the miniport fails to allocate the required memory C1394OpenAdapterChannel will fail with status STATUS_1394_NO_MEMORY. By default, if the ALLOCATE_MAX_REQUESTS flag is not specified and the miniport cannot allocate the memory required for maximum number of requests, it will try to operate by decreasing the number of requests. Currently the maximum number of requests used by FireAPI is 25.</p>

The 1394 stack needs to allocate a certain amount of memory in order to be able to prepare an isochronous request for execution.

It is suggested that applications use the **PACKETS_PER_REQUEST** flag to specify the maximum number of isochronous packets per request they are going to use, so that the 1394 stack can only allocate the necessary amount of memory.

If a client specifies a very big value in the *uMaxPacketsPerRequest* field, then the memory allocation may fail and **C1394OpenAdapterChannel** will return **STATUS_1394_NO_MEMORY**.

If an application queues a request for the adapter channel that involves more isochronous packets than the value specified when opening the adapter channel, then the call to **C1394IsochQueue** will fail with **STATUS_1394_INVALID_REQUEST**.

See Also

C1394CloseAdapterChannel, **C1394IsochQueue**, **C1394GetNextCompleteRequest**, **FIREAPI_ISOCH_REQUEST**

⁴⁸ By default FireAPI uses default=1024, max=8000.

C1394CloseAdapterChannel

Frees an adapter channel that was previously allocated with a successful call to **C1394OpenAdapterChannel**.

```
void C1394CloseAdapterChannel(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    IN C1394_CHANNEL_HANDLE C1394ChannelHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which the DMA channel is open.

C1394ChannelHandle

A handle that identifies to the 1394 stack the adapter channel to be freed.

Remarks

It is suggested that the owner of an adapter channel does not have any pending commands to the adapter channel when it closes it. If there are pending operations for an adapter channel when **C1394CloseAdapterChannel** is called for it, then the class driver will make an implicit call to **C1394IsochCancel** and cancel all the currently pending operations. After the call to **C1394CloseAdapterChannel** returns, *C1394ChannelHandle* is invalidated so the application cannot call **C1394GetNextCompleteRequest** in order to retrieve any requests that were cancelled.

Only the application that opened an adapter channel is allowed to close it. The *C1394AdapterHandle* parameter used in the call to **C1394CloseAdapterChannel** must be the one used in the call to **C1394OpenAdapterChannel** that opened the adapter channel.

See Also

C1394OpenAdapterChannel, **C1394IsochQueue**, **C1394IsochCancel**, **C1394GetNextCompleteRequest**

C1394IsochQueue

Submits to the class driver one or more isochronous operation requests for an adapter channel.

```
STATUS_1394 C1394IsochQueue(
    IN C1394_ADAPTER_HANDLE    C1394AdapterHandle,
    IN C1394_CHANNEL_HANDLE    C1394ChannelHandle,
    IN PFIREAPI_ISOCH_REQUEST  *pIsochRequestArray,
    IN ULONG                    uNumberOfRequests
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which the adapter channel is open.

C1394ChannelHandle

A handle that identifies to the 1394 stack the adapter channel on which to queue the request.

pIsochRequestArray

An array of pointers to structures of type **FIREAPI_ISOCH_REQUEST**, that describe the operations to queue for the specified adapter channel.

uNumberOfRequests

The number of elements in the *pC1394IsochRequestArray*.

Return Values

Value	Description
STATUS_1394_SUCCESS	The requests have been successfully queued in the adapter channel's operation queue.
STATUS_1394_NO_MEMORY	The operation could not be completed because a memory allocation failed.
STATUS_1394_INSUFFICIENT_RESOURCES	The class driver failed to queue the operations due to a lack of resources other than memory.
STATUS_1394_INVALID_HANDLE	A handle specified in the call was invalid.
STATUS_1394_INVALID_PARAMETER	One of the structures in the <i>pC1394IsochRequestArray</i> was invalid.
STATUS_1394_INVALID_REQUEST	One of the requests queue is not of the type allowed on the adapter channel. For example an isochronous receive command was requested on a channel opened with the <i>IsochTransmit</i> channel type.
STATUS_1394_DRIVER_INTERNAL_ERROR	An unexpected error occurred.

Remarks

If the function fails, then none of the requests has been queued for execution. That is, if any one of the requests is invalid for the specified channel, then the whole bunch is rejected.

The array pointed to by *pC1394IsochRequestArray* is returned to the caller as soon as the call returns. This means that this array can reside on the caller’s stack.

The class driver will not accept operation requests for a given adapter channel that are not compatible with the channel’s type. The table below lists the allowed requests for each type of adapter channel.

Adapter Channel Type	Allowed Operations
ChannelIsochReceive	ISOCH_OP_RCV_FIXED_PKTS ISOCH_OP_RCV_FIXED_DATA ISOCH_OP_RCV_FIXED_DATA_NH
ChannelIsochTransmit	ISOCH_OP_XMIT_FIXED_PKTS ISOCH_OP_XMIT_PKTS ISOCH_OP_XMIT_DATA

If an incompatible request is submitted to an adapter channel, then the **C1394IsochQueue** call fails with **STATUS_1394_INVALID_REQUEST**.

The *Tag* field of all **FIREAPI_ISOCH_REQUEST** structures must be set to **TAG_ISOCH_REQUEST**, otherwise **C1394IsochQueue** will fail with **STATUS_1394_INVALID_PARAMETER**.

For more information on the **FIREAPI_ISOCH_REQUEST** structure, see the respective section later in this document.

IMPORTANT NOTE: When a **C1394IsochQueue** call is successful and the requests are queued for execution, then if there were no other commands in the queue the first of the new requests may start executing immediately, before **C1394IsochQueue** actually returns to its caller. This means that if the the first request(s) complete very fast (a small request or due to a bus reset), they might even complete before **C1394IsochQueue** returns. Applications should in general be prepared to deal with this situation. If another thread is blocked waiting on the channel’s event object, then that thread might get unblocked and start executing before **C1394IsochQueue** returns (provided that **C1394IsochQueue** is about to return **STATUS_1394_SUCCESS**).

See Also

C1394IsochCancel

C1394GetNextCompleteRequest

Retrieves the next completed request from an adapter channel's *Completed Request* queue.

```
PFIREAPI_ISOCH_REQUEST C1394GetNextCompleteRequest(  
    IN C1394_CHANNEL_HANDLE    C1394ChannelHandle  
);
```

Parameters

ClassChannelHandle

A handle that identifies to the class driver the adapter channel on which to perform the operation.

Return Values

If the handle is invalid, or the adapter channel does not use a complete request queue (the **COMPLETE_DO_NOT_QUEUE** flag was set in the call to **C1394OpenAdapterChannel**), or there are no more completed requests in the queue, then the function returns NULL. Otherwise the function returns a pointer to the next FIREAPI_ISOCH_REQUEST structure that is completed.

Remarks

See Also

C1394IsochQueue

C1394IsochCancel

Cancels one or more isochronous operations that have been queued for a specific adapter channel.

```
void C1394IsochCancel(  
    IN C1394_ADAPTER_HANDLE    C1394AdapterHandle,  
    IN C1394_CHANNEL_HANDLE    C1394ChannelHandle,  
    IN ULONG                    fCancelOptions,  
    IN ULONG                    uRequestIndex  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which the adapter channel is open.

C1394ChannelHandle

A handle that identifies to the 1394 stack the adapter channel on which to perform the operation.

fCancelOptions

A 32-bit value that identifies various options for the operation. See the notes below for the supported options.

uRequestIndex

A request index (*uRequestIndex* field of **FIREAPI_ISOCH_REQUEST**) needed for the operation.

Remarks

Currently the only defined flag for *fCancelOptions* is *IsochCancelAll*.

If the flag *IsochCancelAll* is specified, then the class driver immediately aborts all queued isochronous operations on the adapter channel. The value of the *uRequestIndex* parameter is ignored in this case.

See Also

C1394IsochQueue

VersaPHY Functions

C1394VPReadNode

This function is analogous to the **C1394ReadNode** function that is provided for standard asynchronous 1394 transactions.

```
STATUS_1394 C1394VPReadNode(
    IN      C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN      C1394_VERSAPHY_BUS_ID     VPBusID,
    IN      C1394_PHYSICAL_ID         PhysicalID,
    IN      C1394_TRANSACTION_LABEL   TLabel,
    IN      BYTE                       BlockNum,
    IN      BYTE                       PRegOff,
    IN      BYTE                       BytesToRead,
    IN      ULONG                      ExpectedResponsePackets,
    IN OUT  void                      *ResponsePacketBuffer
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the read request.

VPBusID

The 6-bit BusID of the destination node. **C1394_VPLOCAL_BUSID** is defined as zero. A value different than zero specifies a remote 1394 bus (not currently supported).

PhysicalID

The 6-bit PhysicalID of the destination node.

TLabel

The 6-bit transaction label to be used for the read request.

The values 0 to 63 can be used if the application wants to manage the TLabel itself.

The value **C1394_VPANY_TLABEL** lets the Class Driver select the transaction label that will be used. The Class Driver will increment the TLabel from 0 to 63 and then back to zero in a circular fashion.

The value **C1394_VPPHYID_TLABEL** is defined so that the adapter's PhyID is used as the transaction label. This will help facilitate debugging/tracing, since the VersaPHY transaction request packets do not contain the id of the originating node, but only that of the destination.

BlockNum

The 8-bit block number that will be specified in the *Blk_Number* field of the VersaPHY read request packet.

PRegOff

The 4-bit block offset that will be specified in the *PReg_Off* field of the VersaPHY read request packet.

BytesToRead

The number of bytes to read (8-bit value).

ExpectedResponsePackets

The number of response packets that are expected for this transaction. This may depend on the type of VersaPHY device, that is if it is 8-bit or 16-bit, and the value of *PRegOff*.

ResponsePacketBuffer

A buffer containing at least `ExpectedResponsePackets* sizeof(C1394_PHY_PACKET)` available bytes that will receive the read response packets for the read transaction.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or BusID>63, or PhysicalID>62, or TLabel>63, or pRegOff>15).
STATUS_1394_INVALID_REQUEST	The request targets the local bus and the node identified by PhysicalID has the LinkOn bit set to 1, so it can't be a VersaPHY device.
STATUS_1394_TIMEOUT	The transmission was completed successfully, but the expected responses were not received within the allotted amount of time.
STATUS_1394_BUS_RESET	The transaction request was cancelled because a bus reset occurred before all the expected responses had been received.

Remarks

One request packet may have multiple response packets depending on the value of *BytesToRead* the value of *PRegOff* and the type of VersaPhy device (8-bit or 16-bit).

If the *TLabel* parameter equals **C1394_PHYID_TLABEL** then the sending node's PhyID is specified as the *TLabel*. This is done in order to facilitate debugging/tracing when examining 1394 bus analyzer logs, since the VersaPhy packets only contain the physical ID of the target device and not the source. Putting the source physical ID in the transaction label makes it easier to analyze the VersaPhy traffic on the bus, especially if more than one PCs are connected to it.

Read/Write requests to a given Physical ID are serialized. If multiple threads/apps try to send a VersaPHY PhyID request to the same Physical ID then the requests are queued and sent one by one after each transaction completes (all expected responses come in or there is a timeout). This is done so that independent applications on the same PC don't accidentally target the same device with concurrent transactions at the same time. It is expected that most VersaPHY devices will not be able to handle multiple concurrent transaction requests.

When a transaction response arrives the Class Driver performs matching based solely on the following fields of the response packet:

- BusID
- Physical ID of the VersaPhy device
- Transaction label

With regards to parameter validation, the value of *BlockNum* in the response packets **is ignored at this time**, since *PRegOff+BytesToRead* is allowed to exceed 16 and in this case the behavior of the devices with regards to the *BlockNum* value they insert in the response packets is not clearly defined yet.

With regards to parameter validation, the value of *PRegOff* in the response packets **is ignored at this time**, due to possible differences in implementation between 8-bit and 16-bit devices.

When the Class Driver receives *ExpectedResponsePackets* matching packets it will consider the transaction successful and the function call will return.

The value of the standard *SPLIT_TIMEOUT* register is used for timing out. The transaction is considered to timeout if the expected number of response packets is not received within the allotted amount of time.

The buffer pointed to by *ResponsePacketBuffer* will be zero-filled upon entry to the function and each incoming response will be copied as it arrives. So when a transaction times out the application can check the contents of the *ResponsePacketBuffer* and discover the number of response packets that were actually received.

The data being read are allowed to cross a 16-byte block boundary, that is *PRegOff+Bytes* may be larger than 16.

When a bus reset occurs all pending VersaPhy physical ID transactions (the one being executed and all others that are possibly queued for the same physical ID) are canceled and completed with the status code **STATUS_1394_BUS_RESET**.

C1394VPWriteNode

This function is analogous to the C1394WriteNode function that is provided for standard asynchronous 1394 transactions.

```
STATUS_1394 C1394VPWriteNode(
    IN      C1394_ADAPTER_HANDLE    C1394AdapterHandle,
    IN      C1394_VERSAPHY_BUS_ID    VPBusID,
    IN      C1394_PHYSICAL_ID        PhysicalID,
    IN      C1394_TRANSACTION_LABEL  TLabel,
    IN      BYTE                      BlockNum,
    IN      BYTE                      PRegOff,
    IN      USHORT                    WriteData,
    IN      ULONG                     ExpectedResponsePackets,
    IN OUT  void                      *ResponsePacketBuffer
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the write request.

VPBusID

The 6-bit BusID of the destination node. **C1394_VPLOCAL_BUSID** is defined as zero. A value different than zero specifies a remote 1394 bus.

PhysicalID

The 6-bit PhysicalID of the destination node.

TLabel

The 6-bit transaction label to be used for the read request.

The values 0 to 63 can be used if the application wants to manage the TLabel itself.

The value **C1394_VPANY_TLABEL** lets the Class Driver select the transaction label that will be used. The Class Driver will increment the TLabel from 0 to 63 and then back to zero in a circular fashion.

The value **C1394_VPPHYID_TLABEL** is defined so that the adapter's PhyID is used as the transaction label. This will help facilitate debugging/tracing, since the VersaPHY transaction request packets do not contain the id of the originating node, but only that of the destination.

BlockNum

The 8-bit block number that will be specified in the *Blk_Number* field of the VersaPHY write request packet.

PRegOff

The 4-bit block offset that will be specified in the *PReg_Off* field of the VersaPHY write request packet.

WriteData

A 16-bit data value that will be written to the device. This value is provided in native (little-endian) format and will be formatted appropriately before sending to the device.

This means that $(WriteData \gg 8)$ will be transmitted at byte 5 of the VersaPHY packet and $(WriteData \& 0xFF)$ will be transmitted at byte 6.

ExpectedResponsePackets

The number of response packets that are expected for this transaction. This may depend on the type of VersaPHY device, that is if it is 8-bit or 16-bit, and the value of *PRegOff*.

ResponsePacketBuffer

A buffer containing at least *ExpectedResponsePackets*sizeof(C1394_PHY_PACKET)* available bytes that will receive the write response packets for the write transaction.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The transaction was completed successfully.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (invalid data buffer, or BusID>63, or PhysicalID>62, or TLabel>63, or PRegOff>15, or ExpectedResponsePackets is zero).
STATUS_1394_INVALID_REQUEST	The request targets the local bus and the node identified by PhysicalID has the LinkOn bit set to 1, so it can't be a VersaPHY device.
STATUS_1394_TIMEOUT	The expected response packets were not received within the allotted amount of time.
STATUS_1394_BUS_RESET	The transaction request was cancelled because a bus reset occurred before the expected response packets were received.

Remarks

If the *TLabel* parameter equals **C1394_VPPHYID_TLABEL** then the sending node's PhyID is specified as the TLabel. This is done in order to facilitate debugging/tracing when examining 1394 bus analyzer logs, since the VersaPhy packets only contain the physical ID of the target device and not the source. Putting the source physical ID in the transaction label makes it easier to analyze the VersaPhy traffic on the bus, especially if more than one PCs are connected to it.

Read/Write requests to a given Physical ID are serialized. If multiple threads/apps try to send a VersaPHY PhyID request to the same Physical ID then the requests are queued and sent one by one after each transaction completes (all expected responses come in or there is a timeout). This is done so that independent applications on the same PC don't accidentally target the same device with concurrent transactions at the same time. It is expected that most VersaPHY devices will not be able to handle multiple concurrent transaction requests.

When a transaction response arrives the Class Driver performs matching based solely on the following fields of the response packet:

- BusID
- Physical ID of the VersaPhy device
- Transaction label
- Block Number

With regards to parameter validation, the value of *PRegOff* in the response packets **is ignored at this time**, due to possible differences in implementation between 8-bit and 16-bit devices.

When the Class Driver receives *ExpectedResponsePackets* matching packets it will consider the transaction successful and the function call will return.

The value of the standard **SPLIT_TIMEOUT** register is used for timing out. The transaction is considered to timeout if the expected number of response packets is not received within the allotted amount of time.

The buffer pointed to by *ResponsePacketBuffer* will be zero-filled upon entry to the function and each incoming response will be copied as it arrives. So when a transaction times out the application can check the contents of the *ResponsePacketBuffer* and discover the number of response packets that were actually received.

When a bus reset occurs, all pending VersaPhy physical ID transactions (the one being executed and all others that are possibly queued for the same physical ID) are completed with the status code **STATUS_1394_BUS_RESET**.

C1394VPSendPacket

This function is analogous to the C1394TransmitRaw function.

```
STATUS_1394 C1394VPSendPacket(  
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN      void                  *VPPacketBuffer  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter through which to transmit the VersaPhy packet.

VPPacketBuffer

A buffer containing a properly formatted VersaPhy packet (8 bytes) that is to be transmitted on the bus.

Remarks

This function permits the sender to send any VP packet to the bus.

The following apply:

- No validity checks are performed on the packet contents.
- The CRC, if valid, is expected to be already calculated by the sender using function **C1394CalculateCRC8**.
- The 8-byte data are expected in big-endian format (ready for transmission).

C1394VPChannelOpen

This function connects the application to the VersaPhy channel specified by the VersaPHY label so that the application can listen to traffic from/for this channel.

```
STATUS_1394 C1394VPChannelOpen(
    IN    C1394_ADAPTER_HANDLE           C1394AdapterHandle,
    IN    C1394_VERSAPHY_LABEL          VersaPhyLabel,
    IN    C1394_VPCHANNEL_OPTIONS       *pVPChannelOptions,
    OUT   HANDLE                         *pStartProcessingEventHandle,
    OUT   C1394_VPCHANNEL_HANDLE        *pC1394VPChannelHandle
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which to open the VersaPhy channel.

VersaPhyLabel

The 14-bit VersaPhy label that the application wants to listen to.

Applications have to perform device discovery themselves in order to discover the VersaPhy labels of devices on the 1394 bus. There is no API provided function that performs this task.

pVPChannelOptions

A pointer to a structure that contains the operational options for this VersaPHY channel.

pStartProcessingEventHandle

A pointer to an event handle where the *StartProcessing* event for this channel will be returned.

This is an auto-reset event. The handle will be closed by the 1394 stack when the VersaPhy channel is closed.

pC1394VPChannelHandle

A pointer to a variable that will receive the handle of the VersaPhy channel.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394AdapterHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (VersaPhyLabel>14-bit, or <i>pStartProcessingEventHandle</i> is invalid, or <i>pC1394VPChannelHandle</i> is invalid).
STATUS_1394_OUT_OF_MEMORY	A memory allocation failed.

Remarks

The processing model for VersaPhy channels follows the one defined for address ranges and isochronous channels.

Initially the channel is considered *Not Busy*. When the first VersaPhy packet arrives then the channel is marked as *Busy* and the *StartProcessing* event is signaled (this is an auto-reset event). The application is then expected to wake and start calling **C1394VPChannelGetNextPacket** until the function returns

FALSE which means there are no more packets to process. Then the channel is marked as *Not Busy* again and the next packet that will arrive will cause the *StartProcessing* event to get signaled and the cycle to start over.

If a packet arrives while the channel is *Busy* then the packet is simply added at the end of the queue and the *StartProcessing* event is not signaled as the application is already *awake* and active doing processing.

The channel that receives all VersaPHY traffic will be referred to as the *Master Channel* while all other channels will be referred to as *VPLabel Channels*.

VPLabel Channels

Each *VPLabel* channel is private to the application that opened it. This means that multiple applications can open the same channel and then listen to and process the same traffic. Every *VPLabel* channel gets its own private queue with incoming packets, so the speed of processing by one application does not affect the others that might be listening on the same *VPLabel* channel.

A separate kernel buffer is allocated from non-paged pool for the packet queue of each *VPLabel* channel. VersaPHY packet reception in this buffer occurs in round-robin fashion and when the queue gets full new incoming packets are discarded.

The size of the reception buffer for a *VPLabel* channel is controlled through the *pVPChannelOptions* parameter.

In the following paragraph a pattern is suggested for processing the VersaPHY packets from user mode applications as efficiently as possible.

Suggested User Mode VersaPHY Packet Processing

To improve the chances of not dropping packets applications can use the following pattern:

- Start a high priority thread that listens to the channel.
- When the *StartProcessingEvent* gets signaled the thread starts a tight loop that pumps VersaPHY packets from the kernel queue into a process-owned queue in user space, until the kernel queue is empty.
- This high-priority listener thread should do NO processing on these packets, but instead signal another worker thread to start processing the process-owned queue.

Given the above pattern and the fact that VersaPHY devices are usually not very intensive in producing unsolicited VersaPHY packets, the value of 4 (1024 packets) as the queue size should be enough for most applications.

Needless to say the user mode application should also set a limit on its queue size, but this can be rather big (several MB). Alternatively the application can store the packets to disk using a memory mapped file that uses a sliding window and thus never have problems with memory allocation.

VPLabel Read/Write Transactions

All *VPLabel* read/write transactions for the same *VPLabel* value are internally routed through a single queue in the Class Driver, no matter which application they are originating from. This is done in order to prevent concurrent transactions requests from being sent to the VersaPHY device, since most devices may be of a simple nature and thus unable to handle multiple concurrent transactions correctly.

When a *VPLabel* response packet is received, it is checked against the currently pending transaction for this *VPLabel* and all related processing takes place before the Class Driver starts to examine the *VPLabel* channel queues that are currently open on this *VPLabel*. This means that the pending VPLabel read/write transactions will always be properly processed, even if the *VPLabel* channel queue is full.

Each application opening a *VPLabel* channel handle has control over the following options:

- Should outgoing *VPLabel* transaction requests and their matching responses (if any) originating from the VPLabel handle itself be recorded to the *VPLabel* channel queue?
- Should outgoing *VPLabel* transaction requests and the matching responses (if any) from other VPLabel handles on the same VPLabel value be recorded to the *VPLabel* channel queue?
- Should packets with bad CRC be inserted in the queue?
- The size of the kernel packet receive queue.

If the application does not wish to record outgoing VPLabel transactions then the VPLabel channel packet queue will only record unsolicited responses from the VersaPHY device. However, if there are other PCs on the same 1394 bus communicating with the VersaPHY device then the VPLabel channel packet queue will record their VPLabel transactions.

C1394 VPLABEL CHANNEL OPTIONS

This structure contains the parameters that control the operation of a VPLabel channel. The structure is defined as follows:

```
typedef struct
{
    // Identification tag. Must be set to TAG_VPLABEL_CHANNEL_OPTIONS
    ULONG        Tag;

    // Should outgoing transactions from the VPLabel handle be recorded?
    UCHAR        bRecordOutgoingFromSelf;

    // Should outgoing transactions from handles on the same VPLabel value be
    recorded?
    UCHAR        bRecordOutgoingToVPLabel;

    // Should packets with bad CRC be accepted?
    UCHAR        bRecordPacketsWithBadCRC8;

    // The size of the kernel packet queue in packets.
    // It will be rounded up to the next 4K page boundary.
    USHORT       ushMaxPacketsInQueue;
}
C1394_VPLABEL_CHANNEL_OPTIONS, *PC1394_VPLABEL_CHANNEL_OPTIONS;
```

Please note that the kernel buffer for the VPLabel channels gets allocated from non-paged memory which is a precious system resource and should not be abused because then the overall performance of the operating system will be degraded.

Do not use big values for `ushMaxPacketsInQueue` just because they seem to work fine; take the time to optimize your application so that it pulls the VersaPHY packets to user mode efficiently and then experiment until you find the minimum safe value for `ushMaxPacketsInQueue`.

The maximum value for the queue size of a VPLabel channel is **MAX_VPLABEL_QUEUE_LENGTH** which is defined as 64K which gives a 1MB buffer.

C1394VPChannelClose

This function disconnects the application from the specified VersaPhy channel.

```
void C1394VPChannelClose(  
    IN    C1394_ADAPTER_HANDLE      C1394AdapterHandle,  
    IN    C1394_VP_CHANNEL_HANDLE   C1394VPChannelHandle  
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the VersaPhy channel is open.

C1394VPChannelHandle

A handle identifying to the 1394 stack the VersaPhy channel.

Remarks

All VersaPhy channels opened by an application are automatically closed if the application crashes or calls **C1394Terminate**.

C1394VPChannelGetNextPacket

This function retrieves the next VersaPhy packet from the specified VersaPhy channel.

```

BOOL C1394VPChannelGetNextPacket(
    IN    C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN    C1394_VPCHANNEL_HANDLE    C1394VPChannelHandle,
    IN    C1394_VERSAPHY_PACKET_INFO *pVPPacketInfo
);

```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the VersaPhy channel is open.

C1394VPChannelHandle

A handle identifying to the 1394 stack the VersaPhy channel.

pVPPacketInfo

A pointer to a buffer that will receive information about the next VersaPhy packet.

Remarks

FALSE is returned to indicate that the queue is empty and there are no more packets to process. As is the case with address ranges and isochronous channels, this will clear the channel's *Busy* flag, so that the event associated with the channel will be signaled again when the next packet arrives.

For reasons of simplicity this function does not return a status code. If the specified VersaPhy channel handle or VersaPhy packet buffer are invalid then FALSE will be returned and a debug message will be written to the kernel debugger.

The **C1394_VERSAPHY_PACKET_INFO** structure is defined as shown below:

```

typedef struct
{
    C1394_PHY_PACKET    VersaPhyPacket;

    // The high performance counter ticks in 100 nsec units and is 64-bit.
    // To come to the point to use the high bits several thousands of years
    // must pass, so we will use the high bits to encode any extra information
    // we want to return.
    union
    {
        ULONGLONG    TimeStamp:62;
        ULONGLONG    bValidCRC8:1;
        ULONGLONG    bIncoming:1;
    };
};
C1394_VERSAPHY_PACKET_INFO, *PC1394_VERSAPHY_PACKET_INFO;

```

All packets are returned in big endian format. To find out about the type of each packet use the **C1394VPGetPacketType** function.

To swap a big-endian packet so that you can use one of the **_LITTLE** structures, use code as shown below:

```

PC1394_VERSAPHY_PACKET_VPLABEL_WRITE_REQUEST_LITTLE pPacketLE;
ULONGLONG OctletLE = *(ULONGLONG*)pPhyPacket->Bytes;
SWAP_ENDIAN_64(OctletLE);
pPacketLE = (PC1394_VERSAPHY_PACKET_VPLABEL_WRITE_REQUEST_LITTLE) &OctletLE;

```


C1394VPChannelRead

This function performs a read transaction using VersaPhy Label addressing.

```
STATUS_1394 C1394VPChannelRead(
    IN      C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN      C1394_VPCHANNEL_HANDLE    C1394VPChannelHandle,
    IN      C1394_TRANSACTION_LABEL   TLabel,
    IN      UCHAR                      LRegOff,
    IN      ULONG                      Data,
    IN      UCHAR                      NumResponses,
    IN      void                       *VPPacketBuffer
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the VersaPhy channel is open.

C1394VPChannelHandle

A handle identifying to the 1394 stack the VersaPhy channel.

TLabel

The 6-bit transaction label to be used for the read request. The value **C1394_VPPHYID_TLABEL** is defined so that the adapter's PhyID is used as the transaction label. This will help facilitate debugging/tracing.

LRegOff

The 4-bit value to be used in the *LReg_Off* field of the read request.

Data

The 24-bit value to be used in the *Profile_Defined_Addr/Data* field of the read request. *Data* is specified in native (little-endian) 32-bit format.

NumResponses

The number of read response packets to expect for this transaction request.

VPPacketBuffer

Pointer to a buffer where a maximum of *NumResponses* VersaPhy packets will be stored.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394VPChannelHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (<i>TLabel</i> >6-bit, or <i>LRegOff</i> >4-bit, or <i>Data</i> >24-bit or <i>VPPacketBuffer</i> is invalid).
STATUS_1394_TIMEOUT	The expected number of responses was not received before the timeout period expired.

Remarks

This function transmits the requested read transaction targeting the VP Label of the channel. Since the operation of read/writes with VersaPhy Label addressing is dependent upon the specific profile being implemented by the device, these functions provide a generic interface for managing this kind of transactions.

The application issuing the read is expected to be aware of the profile being implemented so it should know what the value passed in the *Data* parameter means and thus the number of responses that are expected.

The function will block on the VersaPhy channel, waiting for the specified number of response packets to arrive from the given VersaPhy label and with the specified transaction label.

If the **SPLIT_TIMEOUT** period elapses before the responses are received then the operation will return with a timeout status code. If this is a condition that is expected to occur often, then the application should zero fill the memory pointed to by *VPPacketBuffer* and then manually detect the number of VersaPhy packets that were received by checking the packets for non-zero contents.

The VersaPhy packets that form the reply to the read transaction will not be inserted to the VersaPhy Label queue and thus:

1. The StartProcessing event object will not be signaled (user mode).
2. The matching response packets will not be returned through the **C1394VPChannelGetNextPacket** function.

In similarity to PhyID operations, all read/write transactions using VersaPhy Label addressing that target the same VPLabel are serialized and executed one after the other.

C1394VPChannelWrite

This function performs a write transaction using VersaPhy Label addressing.

```
STATUS_1394 C1394VPChannelWrite(
    IN      C1394_ADAPTER_HANDLE      C1394AdapterHandle,
    IN      C1394_VPCHANNEL_HANDLE    C1394VPChannelHandle,
    IN      C1394_TRANSACTION_LABEL    TLabel,
    IN      UCHAR                      LRegOff,
    IN      ULONG                      Data,
    IN      UCHAR                      NumResponses,
    IN      void                       *VPPacketBuffer
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the VersaPhy channel is open.

C1394VPChannelHandle

A handle identifying to the 1394 stack the VersaPhy channel.

TLabel

The 6-bit transaction label to be used for the read request. The value **C1394_VPPHYID_TLABEL** is defined so that the adapter's PhyID is used as the transaction label. This will help facilitate debugging/tracing.

LRegOff

The 4-bit value to be used in the *LReg_Off* field of the read request.

Data

The 24-bit value to be used in the *Profile_Defined_Addr/Data* field of the read request. *Data* is specified in native (little-endian) 32-bit format.

NumResponses

The number of write response packets to expect for this transaction request.

VPPacketBuffer

Pointer to a buffer where a maximum of *NumResponses* VersaPhy packets will be stored.

Return Values

The possible return values are listed below:

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_INVALID_HANDLE	The handle specified by <i>C1394VPChannelHandle</i> is invalid.
STATUS_1394_INVALID_PARAMETER	A parameter is invalid (<i>TLabel</i> >6-bit, or <i>LRegOff</i> >4-bit, or <i>Data</i> >24-bit or <i>VPPacketBuffer</i> is invalid).
STATUS_1394_TIMEOUT	The expected number of responses was not received before the timeout period expired.

Remarks

See **C1394VPReadNode**.

Control & Information Functions

C1394BusReset

Issues a software initiated long or short bus reset.

```
STATUS_1394 C1394BusReset(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    IN BOOLEAN bShortReset  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which to initiate the bus reset.

bShortReset

Indicates whether a short or long bus reset should be initiated.

Return Values

If the operation is completed successfully, then **STATUS_1394_SUCCESS** is returned.

If a bus reset has occurred in the last 2 seconds then **STATUS_1394_TIMEOUT** is returned. If a short bus reset is required and the adapter is not capable of this operation then

STATUS_1394_INVALID_REQUEST is returned.

If the adapter handle is invalid then **STATUS_1394_INVALID_HANDLE** is returned.

Otherwise the appropriate error status should be returned according to the guidelines described in Status Code Reference.

Remarks

If there are any transmit requests pending at the time the function is called, then they are aborted with the **STATUS_1394_BUS_RESET** status.

The 1394 standard specifies that at least 2 seconds should elapse after a bus reset before a node can initiate a bus reset through software (P1394a 2.1 paragraph 9.13 (modification of P1394a 2.0 paragraph 9.10)). The 1394 class driver checks for this condition and when it occurs it returns

STATUS_1394_TIMEOUT.

Moreover the 1394 class driver makes certain that two or more independent applications do not request a bus reset in less than 2 seconds apart from each other. If this happens, only the first call to

C1394BusReset will be actually performed, while the others will return **STATUS_1394_TIMEOUT**.

See Also

C1394IsBusResetInProgress, **C1394GetBusResetCount**

C1394IsBusResetInProgress

Indicates whether a bus reset is in progress on the specified adapter.

```
BOOLEAN C1394IsBusResetInProgress(  
    C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter for which to return the bus reset count.

Return Values

If the handle provided is invalid, then FALSE is returned.

Remarks

This function can be used in combination with **C1394GetBusResetCount** in order to synchronize with bus reset events. See the remarks section of **C1394GetBusResetCount** for more details.

See Also

C1394BusReset, **C1394GetBusResetCount**

C1394GetBusResetCount

Returns the number of bus resets that have occurred on the specified adapter.

```
ULONG C1394GetBusResetCount(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter for which to return the bus reset count.

Return Values

If the adapter handle is invalid then 0xFFFFFFFF is returned.

Remarks

An application uses this function as a help to determine whether a bus reset has occurred while a piece of code that is affected by bus resets is executing.

Before entering the critical piece of code the application should get the bus reset count and then call **C1394IsBusResetInProgress**. If FALSE is returned the application can enter the critical code. When this code is exited the application should call **C1394GetBusResetCount** once more. If the count returned is different than the original count then a bus reset occurred while the code was executing and it might still be in progress.

Applications can also register a bus reset notification, but this is not quite as suitable for critical checks, because there is inevitably a small delay between the time the bus reset occurs and the time the application's notification handler is called.

See Also

C1394BusReset, **C1394IsBusResetInProgress**

C1394QueryInformation

Returns information about the adapter’s capabilities, characteristics and operational settings.

```

STATUS_1394 C1394QueryInformation(
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN      OID_1394              ObjectIdentifier1394,
    IN OUT  PVOID                 Buffer,
    IN      ULONG                 uBufferLength,
    OUT     ULONG                 *puBytesWritten OPTIONAL,
    OUT     ULONG                 *puBytesRequired OPTIONAL
);
    
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose information is to be queried.

ObjectIdentifier1394

The identifier of the information item that is to be queried.

Buffer

A buffer where information is returned. In a few cases it also contains input parameters.

uBufferLength

The size in bytes of the memory pointed to by *Buffer*.

puBytesWritten

An optional pointer to a ULONG variable that receives the size in bytes of the data returned. If the caller does not care for this information it can set this parameter to NULL.

puBytesRequired

An optional pointer to a ULONG variable that receives the required size in bytes of *Buffer* for the call to succeed. If this information is not needed then the caller can set this parameter to NULL. This information is returned only if the pointer is not NULL and the call fails due to the buffer size being too small.

Return Values

If the call is completed successfully then **STATUS_1394_SUCCESS** is returned. Otherwise the appropriate error status should be returned according to the guidelines described in the Status Code Reference.

Remarks

The object identifiers defined are listed in the table below. The type assumed for *Buffer* for each identifier is listed in another table that follows.

Object Identifier	Description
Adapter Capabilities Group	
OID_ISOCHRONOUS_CAPABLE	Indicates whether the adapter is capable of isochronous operations.
OID_CYCLE_MASTER_CAPABLE	Indicates whether the adapter is capable of being the cycle master.

OID_CYC_CLK_ACC	Specifies the adapter's master clock accuracy in parts per million. A value between 0 and 100 is returned. If the <i>cmc</i> bit is set in <i>Bus_Info_Block</i> , then this value is copied into the <i>cyc_clk_acc</i> field of <i>Bus_Info_Block</i> . Otherwise that field contains all 1's.
OID_POWER_MANAGEMENT_CAPABLE	Indicates whether the adapter is capable of power management.
OID_SHORT_BUS_RESET_CAPABLE	Indicates whether the adapter's PHY is capable of performing an arbitrated (short) bus reset.
OID_PING_CAPABLE	Indicates whether the adapter is capable of performing ping operations.
OID_BYTE_ALIGNMENT_CAPABLE	Indicates whether the adapter is capable of accepting requests for 1394 offsets aligned on byte boundaries.
OID_WORD_ALIGNMENT_CAPABLE	Indicates whether the adapter is capable of accepting requests for 1394 offsets aligned on word (2-bytes) boundaries.
OID_MAP_PHYSICAL_ADDRESS_CAPABLE	Indicates whether the adapter is capable of supporting physical address range mappings.
OID_ISO_RECEIVE_DMA_CONTEXTS	Indicates the number of isochronous receive DMA contexts available in the adapter.
OID_ISO_TRANSMIT_DMA_CONTEXTS	Indicates the number of isochronous transmit DMA contexts available in the adapter.
Bus_Info_Block Group	
OID_CONFIGURATION_ROM	Returns the contents of the 20 bytes that comprise the <i>Bus_Info_Block</i> , starting at offset FFFFF0000400 ₁₆ . The contents are returned as if they were transmitted on the cable in big endian and then possibly byte swapped upon reception according to the setting specified by OID_BYTE_SWAP .
OID_CONFIG_ROM_CRC_CONTROL	Returns the contents of the quadlet found at address FFFFF0000400 ₁₆ . This quadlet contains the <i>info_length</i> , <i>crc_length</i> and <i>rom_crc_value</i> fields, which are described in paragraph 8.3.2.5.3. of 1394-1995. The contents are returned as if they were transmitted on the cable in big endian and then possibly byte swapped upon reception according to the setting specified by OID_BYTE_SWAP .
OID_ISOCHRONOUS_CAPABLE_BIT	Returns the value of the <i>isc</i> bit.
OID_CYCLE_MASTER_CAPABLE_BIT	Returns the value of the <i>cmc</i> bit.
OID_IRM_CAPABLE_BIT	Returns the value of the <i>irmc</i> bit.
OID_SERIAL_BUS_MANAGER_CAPABLE_BIT	Returns the value of the <i>bmc</i> bit.
OID_POWER_MANAGEMENT_CAPABLE_BIT	Returns the value of the <i>pmc</i> bit.
OID_GENERATION_COUNT	Returns the value of the <i>generation</i> field.

<p>OID_MAX_REC</p>	<p>Returns the value of the <i>max_rec</i> field (4-bits). The maximum asynchronous payload that can be accepted by this node in a block write transaction request or sent in a block read response is 2^{\max_rec+1}.</p> <p>The value of zero means that the maximum payload is not specified, and the values E_{16} and F_{16} are reserved.</p>
<p>Control & Operational Settings</p>	
<p>OID_ADAPTER_CHECK_FOR_HUNG</p>	<p>Indicates whether the adapter appears to be hung. If the adapter is hung then TRUE is returned in the output buffer, otherwise FALSE is returned.</p>
<p>OID_ADAPTER_FIFO</p>	<p>Returns information about the current FIFO settings of the adapter. This information describes the total amount of FIFO available on the adapter, and the way into which is partitioned between receive, transmit, asynchronous and isochronous operations.</p> <p>These settings can be configured dynamically through a call to C1394SetInformation. For more information see the <i>Changing FIFO settings</i> section at the end of the document.</p>
<p>OID_DATA_CRC_INDICATED</p>	<p>Indicates whether the data CRC is included in the received packets (which have data payload) that the class driver is indicating to applications. If it is included then it is the last quadlet of the indicated packet.</p>
<p>OID_HEADER_CRC_INDICATED</p>	<p>Indicates whether the header CRC is included in the received packets that the class driver is indicating to applications. If not, then the data payload (if any) follows immediately after the header quadlets in the receive buffer (see 1394-1995 paragraph 6.2.1 for the general format of primary packets).</p>
<p>OID_PCI_LATENCY</p>	<p>Returns the current PCI latency value for the 1394 adapter. PCI latency can range from 0 to 255.</p> <p>This value is configurable, both at run-time through C1394SetInformation and at boot-time through the appropriate registry setting.</p>
<p>OID_QUERY_SW_BYTE_SWAP</p>	<p>Indicates whether the miniport driver will have to do a software byte swap if the class driver asks for a data buffer to be transmitted normally or byte-swapped.</p> <p>Upon calling C1394QueryInformation the <i>Buffer</i> parameter should contain a BOOLEAN value⁴⁹. This value is TRUE if the data buffer would be transmitted byte-swapped and FALSE if it would be transmitted normally.</p> <p>This item helps an application check if the adapter that it is using is of limited capability and cannot support per packet byte-swap settings (for asynchronous packets). In that case the application might want to select to transmit the data in the form that will not cause extra overhead due to the software-performed byte swapping.</p> <p>Note that if the adapter hardware supports per-packet transmit byte-swap settings, then FALSE is always returned in the output buffer, since software byte swaps are never needed.</p>

⁴⁹ * ((BOOLEAN *)*Buffer*) must be either TRUE or FALSE.

OID_RECEIVE_BUFFER_SIZE	Returns the size in bytes of the reception buffer that the miniport uses for receiving asynchronous traffic. This value is registry configurable, and is read by the 1394 stack at load time.
OID_DEF_AR_PACKET_TRANSFER	Returns the default maximum number of asynchronous packets that will be transferred from kernel mode to user mode in each call to C1394GetNextRequest . New address ranges are initialized with this value. Applications can either modify the global setting or modify this setting separately for specific address ranges by using C1394SetInformation and the OID_AR_PACKET_TRANSFER identifier.
OID_AR_PACKET_TRANSFER	Returns the current value of the maximum number of asynchronous packets that will be transferred from kernel mode to user mode in each call to C1394GetNextRequest for the specific address range.
OID_BUS_RESET_EXCEPTIONS	Maintained after the release of ubCore 5.50 for backwards compatibility reasons. Always indicates that UB1394.DLL will NEVER raise a SEH exception when a transaction request fails because of a bus reset.
OID_ISO_REQUEST_PACKETS	Return the default maximum number of isochronous packets that an adapter channel will be able to accept with a single isochronous request. This value is the one that the class driver will use by default if a client does not specify the PACKETS_PER_REQUEST flag when opening an adapter channel. This value is registry configurable, and is read by the 1394 stack at load time.
OID_MAX_ISO_REQUEST_PACKETS	Returns the overall maximum number of isochronous packets that an adapter channel can accept in a single request. Clients that specify the PACKETS_PER_REQUEST flag when opening an adapter channel, cannot put a greater value in the <i>uMaxPacketsPerRequest</i> field. This value is registry configurable, and is read by the 1394 stack at load time.
OID_DMA_LIMIT	Returns the maximum size in bytes of a single DMA operation that the operating system can perform on the current hardware platform. This is usually around 2GB for 32-bit systems, and may get limited to 1MB for 64-bit systems. This value is a hard limit on the size of an isochronous operation. If you want to transmit/receive more bytes than that you will have to use multiple isochronous requests, so that each is less than the DMA limit.

OID_MULTIDMA_MODE	<p>Returns the current DMA Multiplexing Mode. Supported values are:</p> <pre>typedef enum { MultiDMAOnLastContext = 0, MultiDMADisabled = 1, MultiDMAForced = 2, } MultiDmaOperation;</pre> <p>See the documentation in the isochronous operations section for more details.</p>
Identification Information Group	
OID_GUID	The adapter's GUID. The C1394_GUID type is defined as a structure containing a string with 8 bytes, so effectively the GUID is returned in big endian.
OID_PRODUCT_ID	The adapter's Product ID. The C1394_PRODUCT_ID type is defined as a structure containing a string with 3 bytes, so effectively the product ID is returned in big endian.
OID_VENDOR_ID	The adapter's Vendor ID. The C1394_VENDOR_ID type is defined as a structure containing a string with 3 bytes, so effectively the product ID is returned in big endian.
OID_1394_COMPLIANCE	<p>Returns whether the adapter is 1394-1995 or P1394a compliant. The possible return values are:</p> <ul style="list-style-type: none"> • COMPLIANCE_NONE • COMPLIANCE_1394_1995 • COMPLIANCE_1394A • COMPLIANCE_1394B
OID_OHCI_VERSION	<p>The OHCI version to which the adapter complies. The major version is in the high word and the minor version is in the low word.</p> <p>COMPLIANCE_NONE indicates a non OHCI-compliant adapter.</p> <p>COMPLIANCE_PARTLY_OHCI (defined as 0xFFFFFFFF) indicates an adapter that only supports a subset of the OHCI standard.</p>
Link Layer Information Group	
OID_LINK_SPEED	<p>Returns the maximum speed capability of the LINK on this adapter. This information is made available to remote nodes by the class driver, through the <i>link_spd</i> field in <i>Bus_Info_Block</i> (P1394a p9.18).</p> <p>The possible return values are {S100, S200, S400, S800, S1600, S3200}.</p>
OID_LINK_LAYER_ENABLE_STATUS	<p>Indicates whether the link layer is enabled on the adapter.</p> <p>This corresponds to bit <i>linkEnable</i> in the HCControl registers (OHCI paragraph 5.7).</p>
OID_ISOCHRONOUS_ENABLE_STATUS	Indicates whether the adapter has enabled isochronous operations.

OID_POSTED_WRITES_ENABLE_STATUS	Indicates whether the adapter has enabled posted writes (OHCI paragraph 3.3.3). This corresponds to bit <i>postedWriteEnable</i> in the <i>HCControl</i> registers (OHCI paragraph 5.7). <i>Note: In the Windows™ implementation physical writes are not enabled at all, so this should return 0.</i>
OID_LPS_BIT	Indicates whether the communication between the Link and the PHY is enabled. This corresponds to bit <i>LPS</i> in the <i>HCControl</i> register (OHCI paragraphs 5.7, 5.7.3).
OID_ABDICATE_BIT	Returns the value of the <i>abdicate</i> bit in the <i>bus_depend</i> field of the <i>STATE_CLEAR</i> register (P1394a paragraph 9.23).
OID_BYTE_SWAP	Indicates whether data should be byte swapped by the adapter upon reception of asynchronous transaction requests/responses. The byte swap setting for incoming channel data is specified separately for each channel. See the discussions at the beginning of this document for a description of the issues involved with byte swapping. This corresponds to the reverse of the <i>noByteSwapData</i> bit in the <i>HCControl</i> register (OHCI paragraphs 5.7, 5.7.1). The start up default for all adapters is not to byte swap data; the miniport driver should by default assume a big endian representation. The endianness setting is controlled through the class driver's registry settings.
OID_LINK_REGISTER_ACCESS	Reads a quadlet value from one of the standard registers of the link layer. For more information on this capability see the section <i>Changing the Link Registers</i> of this document.
OID_CYCLE_START_AVAILABLE	Returns whether there is any cycle start activity on the bus. This means that either the adapter is generating these cycle start packets itself (which means it is root and cycle master), or it is receiving cycle start packets from the bus. When the adapter is the root node on its bus, the identifier OID_CYCLE_MASTER_STATUS can be used to find out whether the adapter is also acting as the cycle master or not.
OID_DUAL_PHASE_RETRY_SUPPORTED	Indicates whether the adapter supports the dual phase retry protocol.
OID_CURRENT_RETRY_PROTOCOL	Returns the current retry protocol in use on the adapter. The possible return values are: <ul style="list-style-type: none"> • SINGLE_PHASE_RETRY_PROTOCOL • DUAL_PHASE_RETRY_PROTOCOL
OID_DUAL_PHASE_RETRY_TIMEOUT	Returns the timeout used in the dual phase retry protocol (if it is supported by the adapter).

OID_SINGLE_PHASE_RETRY_COUNTS	<p>Returns the single phase retry counts:</p> <ol style="list-style-type: none"> Asynchronous Response Retries Asynchronous Request Retries <p>This information is found in the <i>ATRetries</i> register (OHCI 1.0 Paragraph 5.4). If the adapter is not OHCI compliant then the equivalent information is returned.</p>
PHY Information Group	
OID_NODE_ID	Returns the 16-bit Node ID of the adapter. The value is returned using the C1394_NODE_ID structure which uses bit fields over a USHORT. So the value returned is always in the native endianness of the machine that the driver runs on.
OID_ROOT_STATE	Indicates whether the adapter is the root node on its attached network. This can be found at the <i>R</i> bit in the standard PHY registers (P1394a Paragraph 6.1).
OID_FORCE_ROOT	Returns the value of the <i>force_root</i> variable for the adapter.
OID_GAP_COUNT	Returns the value of the <i>gap_count</i> variable for the adapter. This can be found as the <i>Gap_count</i> field in the standard PHY registers (P1394a Paragraph 6.1).
OID_POWER_STATUS_BIT	Returns the value of the <i>PS</i> bit (cable power active) in the standard PHY registers (P1394a Paragraphs 6.1, 7.3).
OID_ROOT_HOLD_OFF_BIT	Returns the value of the <i>RHB</i> bit in the standard PHY registers (P1394a Paragraph 6.1).
OID_SELF_ID_PACKETS	Returns a copy of the self-ID packets that the adapter received during the last bus reset. These self-ID packets will always contain the self-ID packet(s) of the local adapter.
OID_CONTENDER_BIT	<p>Returns the value that the adapter will use for the <i>c</i> bit in its self-ID packets it is going to transmit after the next bus reset.</p> <p>This is available as the <i>Contender</i> bit in the extended PHY registers (P1394a Paragraph 6.1).</p>
OID_CYCLE_MASTER_STATUS	Indicates whether the node is performing cycle master tasks. This is indicated through the <i>cmstr</i> bit in the STATE_CLEAR. <i>bus_depend</i> field (1394-1995 Paragraph 8.3.2.2.1).
OID_PHY_ENHANCEMENTS_ENABLE_STATUS	<p>Returns the value of the <i>aPHYEnhanceEnable</i> bit in the <i>HCControl</i> register (OHCI 1.0 Paragraph 5.7).</p> <p>It indicates whether the PHY enhancements are enabled.</p>
OID_PHY_SPEED	Returns the maximum speed capability of the PHY on this adapter. The possible return values are {S100, S200, S400, S800, S1600, S3200}.
OID_MAX_REPEATER_DELAY	<p>Returns the value of the <i>Delay</i> field in the extended PHY registers (P1394a Paragraph 6.1), which indicates the worse case repeater delay expressed as:</p> <p style="text-align: center;">$144 + Delay * 20 \text{ nsec}$</p>

OID_JITTER	Returns the value of the <i>Jitter</i> field in the extended PHY registers (P1394a Paragraph 6.1), which indicates the difference between the fastest and the slowest repeater delay expressed as: $(Jitter+1)*20$ nsec
OID_POWER_CLASS	Returns the power class of the adapter. This is the value transmitted in the adapter's self-ID packets. It is also located in the <i>Pwr</i> field of the extended PHY registers (P1394a Paragraph 6.1).
OID_ARBITRATION_ACCELERATION_STATUS	Returns the value of the <i>Enab_accel</i> bit in the extended PHY registers (P1394a Paragraph 6.1), which indicates whether the arbitration accelerations are enabled.
OID_MULTI_SPEED_PACKET_CONCATENATE_STATUS	Returns the value of the <i>Enab_multi</i> bit in the extended PHY registers (P1394a Paragraph 6.1), which indicates that multi speed packet concatenation is enabled.
OID_PHY_REGISTER_PAGE	Returns the contents of the requested PHY register page. These registers are accessible through the <i>Page_Select</i> and <i>Port_Select</i> fields of the extended PHY register map (P1394a Paragraph 6.1).
OID_SUSPEND_RESUME_FAULT	Indicates whether the <i>Fault</i> bit in the Port Status register page (PHY Register Page 0) for the specified port is set.
OID_TOTAL_PORTS	Returns the number of ports found on the adapter. This is available as the <i>Total_ports</i> field in the extended PHY registers (P1394a Paragraph 6.1).
OID_PORT_STATUS	Returns the status of the specified port. The possible values returned through <i>OutputBuffer</i> are: <ul style="list-style-type: none"> • PortStatusInvalid: The port number is not valid on this adapter. • PortStatusUnknown: The miniport cannot provide per port status information. • PortStatusConnected: The port is connected. • PortStatusDisconnected: The port is disconnected. • PortStatusDisabled: The port is disabled. • PortStatusSuspended: The port is suspended.
OID_CHILD_PORT	Indicates whether the specified port is a child port. If the port specified is invalid then the function should return STATUS_1394_INVALID_PARAMETER .
OID_PORT_NEGOTIATED_SPEED	Returns the maximum speed negotiated between this PHY port and its immediately connected port.
OID_ASYNCHRONOUS_REQUEST_FILTER_STATUS	Returns in <i>OutputBuffer</i> the contents of the <i>AsynchRequestFilter</i> register (OHCI 1.0 Paragraph 5.13.1).
OID_ADAPTER_CONNECTED	Indicates whether any of the adapter's ports are physically connected. Although this information can be derived by examining the port status information, some adapter's may not be able to provide that kind of information, while they might be able to indicate whether or not any port at all is connected.

Channel Information Group	
OID_CHANNEL_MASK	<p>Returns the current value of the channel mask associated with an adapter channel (used for stream operations).</p> <p>This 64-bit mask informs the drivers about the isochronous channel numbers that the client intends to use through the adapter channel. The class driver will only allow receive/transmit requests for the channel numbers set in this mask.</p> <p>Clients can control this mask by specifying the OID_CHANNEL_MASK oid in a call to C1394SetInformation.</p>
OID_CHANNEL_REQUEST_INDEX	<p>Returns the value that will be used as the request index for the next request to be submitted to the class driver.</p>
Core CSR Registers Group	
OID_CSR_BUS_MANAGER_ID	<p>Returns the contents of the BUS_MANAGER_ID register (1394-1995 paragraph 8.3.2.3.6). <i>See note below for OID_CSR_xxx.</i></p>
OID_CSR_CYCLE_TIME	<p>Returns the contents of the CYCLE_TIME register (1394-1995 paragraph 8.3.2.3.1). <i>See note below for OID_CSR_xxx.</i></p>
OID_CSR_BANDWIDTH_AVAILABLE	<p>Returns the contents of the BANDWIDTH_AVAILABLE register (1394-1995 paragraph 8.3.2.3.7). <i>See note below for OID_CSR_xxx.</i></p>
OID_CSR_CHANNELS_AVAILABLE	<p>Returns the contents of the CHANNELS_AVAILABLE register (1394-1995 paragraph 8.3.2.3.8). <i>See note below for OID_CSR_xxx.</i></p>
OID_CSR_BUS_TIME	<p>Returns the contents of the BUS_TIME register (1394-1995 paragraph 8.3.2.3.2). <i>See note below for OID_CSR_xxx.</i></p>
OID_CSR_BUSY_TIMEOUT	<p>Returns the contents of the BUSY_TIMEOUT register (1394-1995 paragraph 8.3.2.3.5). <i>See note below for OID_CSR_xxx.</i></p>
Bus Topology Information Group	
OID_NODE_COUNT	<p>Returns the number of nodes that are connected to the bus. NOTE: Requires the bus number as input parameter.</p>
OID_PHYSICAL_NODES	<p>Returns a 64-bit value indicating the physical IDs of the nodes that are physically present on the specified bus. NOTE: Requires the bus number as input parameter.</p>
OID_LINK_ON_NODES	<p>Returns a 64-bit value indicating the physical IDs of the nodes on the specified bus that have their link layer active. These are the nodes that had the L-bit set in the self-ID packets that were transmitted after the last bus reset. NOTE: Requires the bus number as input parameter.</p>
OID_CONTENDER_NODES	<p>Returns a 64-bit value indicating the physical IDs of the nodes on the specified bus that had the c-bit (contender) set in the self-ID packets that were transmitted after the last bus reset. NOTE: Requires the bus number as input parameter.</p>
OID_HOP_COUNT	<p>Returns the bus hop count (the maximum distance between any two nodes) . NOTE: Requires the bus number as input parameter.</p>

OID_SPEED_TABLE	Returns a copy of the class driver's speed table for the specified bus. This table contains the maximum transmission rate between any pair of nodes on that bus. NOTE: Requires the bus number as input parameter.
OID_SELFID_ANALYSIS_ERROR	Returns a mask with the error codes that were detected by the class driver during the selfID packet analysis. When there is no problem, the value returned is equal to SELFID_OK. For a complete list of the defined flags, see section SelfID Analysis Errors at the end of this document. NOTE: Requires the bus number as input parameter.
OID_TOPOLOGY_ANALYSIS_ERROR	Returns a mask with the error codes that were detected during the topology analysis by the class driver. When there is no problem, the value returned is equal to TOPOLOGY_OK. For a complete list of the defined flags, see section Topology Analysis Errors at the end of this document. NOTE: Requires the bus number as input parameter.
OID_BUS_TOPOLOGY	Returns information about the node on the bus, including information on the tree structure of the bus. NOTE: Requires the bus number as input parameter.
Statistics Group	
OID_STATISTICS_ENABLE_STATUS	Indicates whether statistics collection is enabled on the adapter.
OID_ASYNCHRONOUS_STATISTICS	Returns the adapter's asynchronous statistics.
OID_STREAM_STATISTICS	Returns the adapter's stream statistics (totals for all channels, both asynchronous and isochronous).
OID_CHANNEL_STATISTICS	Returns the adapter's statistics for the specified channel.
OID_PHY_STATISTICS	Returns the adapter's PHY statistics.

Note that the OIDs in the Core CSR Registers Group have been introduced solely for the purpose of making possible synchronous access to these registers if they are available on the local host.

The output parameter requirements for each type are listed in the table that follows.

Object Identifier	Buffer	UBufferLength
OID_ABDICATE_BIT OID_ARBITRATION_ACCELERATION_STATUS OID_ADAPTER_CHECK_FOR_HUNG OID_ADAPTER_CONNECTED OID_BYTE_ALIGNMENT_CAPABLE OID_BYTE_SWAP OID_CHILD_PORT OID_CYCLE_MASTER_CAPABLE OID_CYCLE_MASTER_CAPABLE_BIT OID_CYCLE_MASTER_STATUS OID_CYCLE_START_AVAILABLE OID_CONTENDER_BIT OID_DATA_CRC_INDICATED OID_DUAL_PHASE_RETRY_SUPPORTED OID_FORCE_ROOT OID_HEADER_CRC_INDICATED OID_IRM_CAPABLE_BIT OID_ISOCHRONOUS_CAPABLE OID_ISOCHRONOUS_CAPABLE_BIT OID_ISOCHRONOUS_ENABLE_STATUS OID_LINK_LAYER_ENABLE_STATUS OID_LPS_BIT OID_MULTI_SPEED_PACKET_CONCATENATE_STATUS OID_PHY_ENHANCEMENTS_ENABLE_STATUS OID_PING_CAPABLE OID_POSTED_WRITES_ENABLE_STATUS OID_POWER_MANAGEMENT_CAPABLE OID_POWER_MANAGEMENT_CAPABLE_BIT OID_POWER_STATUS_BIT OID_ROOT_HOLD_OFF_BIT OID_ROOT_STATE OID_SERIAL_BUS_MANAGER_CAPABLE_BIT OID_SHORT_BUS_RESET_CAPABLE OID_STATISTICS_ENABLE_STATUS OID_WORD_ALIGNMENT_CAPABLE OID_MAP_PHYSICAL_ADDRESS_CAPABLE	BOOLEAN *	sizeof(BOOLEAN)
OID_1394_COMPLIANCE OID_CONFIG_ROM_CRC_CONTROL OID_CURRENT_RETRY_PROTOCOL OID_CYC_CLK_ACC OID_CYCLE_TIME_CHANGE_THRESHOLD OID_DEF_AR_PACKET_TRANSFER OID_DMA_LIMIT OID_JITTER OID_GAP_COUNT OID_GENERATION_COUNT OID_HOP_COUNT OID_ISO_RECEIVE_DMA_CONTEXTS OID_ISO_REQUEST_PACKETS OID_ISO_TRANSMIT_DMA_CONTEXTS OID_LINK_SPEED OID_MAX_ISO_REQUEST_PACKETS OID_MAX_REC OID_MAX_REPEATER_DELAY OID_MULTIDMA_MODE OID_NODE_COUNT OID_OHCI_VERSION OID_PCI_LATENCY OID_PHY_SPEED OID_POWER_CLASS OID_PORT_STATUS OID_RECEIVE_BUFFER_SIZE OID_SELFID_ANALYSIS_ERROR OID_TOTAL_PORTS	ULONG *	sizeof(ULONG)
OID_PHYSICAL_NODES OID_LINK_ON_NODES OID_CONTENDER_NODES	ULONGLONG *	sizeof(ULONGLONG)
OID_SPEED_TABLE	void *	SPEED_TABLE_SIZE
OID_BUS_TOPOLOGY	PC1394_BUS_TOPOLOGY_INFO	sizeof(C1394_BUS_TOPOLOGY_INFO)
OID_BUS_RESET_EXCEPTIONS	PC1394_BR_EXCEPTION_STATUS	sizeof(C1394_BR_EXCEPTION_STATUS)
OID_CONFIGURATION_ROM	void *	20 bytes
OID_NODE_ID	PC1394_NODE_ID	sizeof(C1394_NODE_ID)
OID_GUID	PC1394_GUID	sizeof(C1394_GUID)

OID_PRODUCT_ID	PC1394_PRODUCT_ID	sizeof(C1394_PRODUCT_ID)
OID_VENDOR_ID	PC1394_VENDOR_ID	sizeof(C1394_VENDOR_ID)
OID_DUAL_PHASE_RETRY_TIMEOUT	PC1394_DUAL_PHASE_TIMEOUT	sizeof(C1394_DUAL_PHASE_TIMEOUT)
OID_SINGLE_PHASE_RETRY_COUNTS	PC1394_SINGLE_PHASE_RETRIES	sizeof(C1394_SINGLE_PHASE_RETRIES)
OID_SELF_ID_PACKETS	void *	variable
OID_ASYNCHRONOUS_REQUEST_FILTER_STATUS	PREGISTER_64	sizeof(REGISTER_64)
OID_CSR_BUS_TIME OID_CSR_BANDWIDTH_AVAILABLE OID_CSR_BUS_MANAGER_ID OID_CSR_BUSY_TIMEOUT OID_CSR_CYCLE_TIME	PREGISTER_32	sizeof(REGISTER_32)
OID_CSR_CHANNELS_AVAILABLE	PREGISTER_64	sizeof(REGISTER_64)
OID_ASYNCHRONOUS_STATISTICS	<i>Not defined yet</i>	<i>Not defined yet</i>
OID_STREAM_STATISTICS	<i>Not defined yet</i>	<i>Not defined yet</i>
OID_PHY_STATISTICS	<i>Not defined yet</i>	<i>Not defined yet</i>

The following object identifiers use the *OutputBuffer* parameter to point to a structure that contains parameters that **C1394QueryInformation** should use. The function results are stored in this same structure.

Object Identifier	Buffer	UbufferLength
OID_PORT_STATUS OID_CHILD_PORT OID_SUSPEND_RESUME_FAULT OID_PORT_NEGOTIATED_SPEED	PC1394_PORT_INFORMATION	sizeof(C1394_PORT_INFORMATION)
OID_BUS_TOPOLOGY OID_CONTENDER_NODES OID_HOP_COUNT OID_LINK_ON_NODES OID_NODE_COUNT OID_PHYSICAL_NODES OID_SELFID_ANALYSIS_ERROR OID_SPEED_TABLE	PC1394_BUS_ID	sizeof(C1394_BUS_ID)
OID_AR_PACKET_TRANSFER	C1394_AR_PACKET_TRANSFER	sizeof(C1394_AR_PACKET_TRANSFER)
OID_QUERY_SW_BYTE_SWAP	BOOLEAN *	sizeof(BOOLEAN)
OID_LINK_REGISTER_ACCESS	PC1394_LINK_REGISTER_ACCESS	sizeof(C1394_LINK_REGISTER_ACCESS)
OID_PHY_REGISTER_PAGE	PC1394_PHY_REGISTER_PAGE	sizeof(C1394_PHY_REGISTER_PAGE)
OID_ADAPTER_FIFO	PC1394_ADAPTER_FIFO_SETTINGS	sizeof(C1394_ADAPTER_FIFO_SETTINGS)
OID_CHANNEL_MASK	PCHANNEL_MASK_STRUCT	sizeof(CHANNEL_MASK_STRUCT)
OID_CHANNEL_REQUEST_INDEX	PCHANNEL_RQ_INDEX_INFO	sizeof(CHANNEL_RQ_INDEX_INFO)
OID_CHANNEL_STATISTICS	<i>Not yet defined.</i>	<i>Not yet defined.</i>

Note: Some of the object identifiers listed earlier might correspond to items that are not directly available on an adapter, for example fields in the extended PHY register map. If the miniport driver can obtain equivalent information in some other adapter-specific way, then it will provide support for that object identifier (in essence emulate the feature).

Note: There are some pairs of OIDs in the form XXX_CAPABLE and XXX_CAPABLE_BIT. This is done in order to allow maximum flexibility. For example the adapter might be XXX_CAPABLE but an application might want it to temporarily turn off the XXX_CAPABLE_BIT in the *Bus_Info_Block*.

Many of the object identifiers listed above, are for simple data types, like BOOLEAN and ULONG. For ease of use an application can use the **C1394QueryULONGInformation** and **C1394QueryBooleanInformation** functions to query any item of ULONG or BOOLEAN type that does not need input information. These functions are actually calling **C1394QueryInformation** with the appropriate parameters and return status code checks.

See Also

C1394QueryULONGInformation, C1394QueryBooleanInformation

C1394QueryBooleanInformation

Wraps a call to **C1394QueryInformation** for any information item which returns a BOOLEAN value.

```
BOOLEAN C1394QueryBooleanInformation(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    IN OID_1394              Oid1394  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose information is to be queried.

Oid1394

The identifier of the information item that is to be queried.

Return Values

If the internal call to **C1394QueryInformation** is successful, then the data value returned by **C1394QueryInformation** is also returned by this function. If the call to **C1394QueryInformation** fails for any reason (invalid parameter, item not supported, invalid data type) then the value returned is FALSE.

Remarks

This function is provided for convenience when calling **C1394QueryInformation** for standard object identifiers that are known to be supported/available. In that case the call to **C1394QueryInformation** never fails (provided the parameters are OK), so an application can use this function in order to simplify its code.

Since this function returns no status code, it has no way to indicate an error. For that reason if the call to **C1394QueryInformation** fails then FALSE is returned.

Note that this function does not check if the object identifier provided is one that returns BOOLEAN information, but simply passes it on to **C1394QueryInformation**. It is the responsibility of the caller to provide one of the BOOLEAN information object identifiers.

See Also

C1394QueryInformation

C1394QueryULONGInformation

Wraps a call to **C1394QueryInformation** for any information item which returns a ULONG.

```
ULONG C1394QueryULONGInformation(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    IN OID_1394              Oid1394,  
    IN ULONG                 ErrorReturnValue  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose information is to be queried.

Oid1394

The identifier of the information item that is to be queried.

ErrorReturnValue

The identifier of the information item that is to be queried.

Return Values

If the internal call to **C1394QueryInformation** is successful, then the data value returned by **C1394QueryInformation** is also returned by this function. If the call to **C1394QueryInformation** fails for any reason (invalid parameter, item not supported, invalid data type) then the value specified by the *ErrorReturnValue* parameters is returned.

Remarks

This function is provided for convenience when calling **C1394QueryInformation** for standard object identifiers that are known to be supported/available. In that case the call to **C1394QueryInformation** never fails (provided the parameters are OK), so an application can use this function in order to simplify its code.

Since this function returns no status code, the only way to indicate an error is by returning a special value.

Note that this function does not check if the object identifier provided is one that returns ULONG information, but simply passes it on to **C1394QueryInformation**. It is the responsibility of the caller to provide one of the ULONG information object identifiers.

IMPORTANT NOTE: This function will most probably fail for any object identifiers that require input parameters (**OID_NODE_COUNT**, **OID_HOP_COUNT**, etc), because the caller has no way of specifying these parameters (which means that most probably a random value would be used). For such OIDs use **C1394QueryInformation**.

See Also

C1394QueryInformation

C1394SetInformation

This function is used to control various operational settings of the drivers.

```

STATUS_1394 C1394SetInformation(
    IN  C1394_ADAPTER_HANDLE  ClassAdapterHandle,
    IN  OID_1394              ObjectIdentifier1394,
    IN  PVOID                 InputBuffer,
    IN  ULONG                 uInputBufferLength
)
;

```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter on which to perform the operation.

ObjectIdentifier1394

The identifier of the information item that is to be set.

InputBuffer

A pointer to a structure that is going to be used as the input buffer to the operation.

uInputBufferLength

The size of the input buffer in bytes.

Return Values

Value	Description
STATUS_1394_SUCCESS	The operation was completed successfully.
STATUS_1394_INVALID_HANDLE	Either <i>ClassAdapterHandle</i> or a handle specified in the oid-specific parameters is invalid.
STATUS_1394_INVALID_BUFFER_SIZE	The <i>uInputBufferLength</i> parameter is smaller than required for the operation.
STATUS_1394_INVALID_PARAMETER	On of the <i>ObjectIdentifier1394</i> , <i>InputBuffer</i> , or <i>uInputBufferLength</i> parameters, or an oid-specific parameter is invalid.
STATUS_1394_NO_MEMORY	A necessary memory allocation failed.
STATUS_1394_CONFLICT	The operation failed because there was a logical conflict (for example an attempt to enable a channel number that is already enable on a different adapter channel).
STATUS_1394_NOT_SUPPORTED	The required functionality is not available on the adapter.
STATUS_1394_NOT_IMPLEMENTED	The required functionality is supported by the adapter but not yet implemented by drivers.
STATUS_1394_DRIVER_INTERNAL_ERROR	An unexpected error occurred.

Depending on the semantics associated with each object identifier, there might be additional return values that only apply to the particular identifier. These are described in the remarks section.

Remarks

The object identifiers defined are listed in the table below. The type assumed for *Buffer* for each identifier is listed in another table that follows.

Object Identifier	Description
Bus_Info_Block Group	
OID_SERIAL_BUS_MANAGER_CAPABLE_BIT	Sets the value of the <i>bmc</i> bit, as this appears in the configuration ROM.
OID_POWER_MANAGEMENT_CAPABLE_BIT	Sets the value of the <i>pmc</i> bit.
Link Layer Information Group	
OID_LINK_REGISTER_ACCESS	Writes a quadlet value to one of the standard registers of the link layer. For more information on this capability see the section <i>Changing the Link Registers</i> of this document.
Control & Operational Settings	
OID_ADAPTER_FIFO	Sets the current FIFO settings of the adapter. This can control the total amount of FIFO used on the adapter, and the way into which is partitioned between receive, transmit, asynchronous and isochronous operations. For more information see the <i>Changing FIFO settings</i> section at the end of the document.
OID_PCI_LATENCY	Sets the current PCI latency value for the 1394 adapter. PCI latency can range from 0 to 255. This value is configurable, both at run-time through C1394SetInformation and at boot-time through the appropriate registry setting of the miniport driver.
OID_DEF_AR_PACKET_TRANSFER	Sets the default maximum number of asynchronous packets that will be transferred from kernel mode to user mode in each call to C1394GetNextRequest . New address ranges are initialized with this value. Applications can modify this setting separately for specific address ranges by using the identifier OID_AR_PACKET_TRANSFER .
OID_AR_PACKET_TRANSFER	Sets the maximum number of asynchronous packets that will be transferred from kernel mode to user mode in each call to C1394GetNextRequest for a specific address range.
OID_BUS_RESET_EXCEPTIONS	Not Supported after the release of ubCore 5.50.
OID_MULTIDMA_MODE	Sets the current DMA Multiplexing Mode. Supported values are: <pre>typedef enum { MultiDMAOnLastContext = 0, MultiDMADisabled = 1, MultiDMAForced = 2, } MultiDmaOperation;</pre> See the documentation in the isochronous operations section for more details on each more. The current MultiDMA mode can only be changed when there are no active adapter channels.

Channel Information Group	
OID_CHANNEL_MASK	Set the current value of the channel mask associated with an adapter channel (used for stream operations). This 64-bit mask informs the drivers about the isochronous channel numbers that the client intends to use through the adapter channel. The class driver will only allow receive/transmit requests for the channel numbers set in this mask.
OID_CHANNEL_MASK_ENABLE	Enables for use with an adapter channel the bits specified in the channel mask provided. The channel numbers that correspond to bits that are zero in the provided mask will not be affected. Effectively this OID does a binary OR of the provided mask and the current value of the adapter channel's channel mask. This OID is meant for easier control of the channel mask.
OID_CHANNEL_MASK_DISABLE	Disables for use with an adapter channel the bits specified in the channel mask provided. The channel numbers that correspond to bits that are zero in the provided mask will not be affected. Effectively this OID does a binary AND of the provided mask's complement and the current value of the adapter channel's channel mask. This OID is meant for easier control of the channel mask.

The input parameter requirements for each OID are listed in the table that follows.

Object Identifier	InputBuffer	Buffer Length
OID_POWER_MANAGEMENT_CAPABLE_BIT OID_SERIAL_BUS_MANAGER_CAPABLE_BIT	BOOLEAN *	sizeof(BOOLEAN)
OID_ADAPTER_FIFO	PC1394_ADAPTER_FIFO_SETTINGS	sizeof(PC1394_ADAPTER_FIFO_SETTINGS)
OID_PCI_LATENCY	ULONG *	sizeof(ULONG)
OID_LINK_REGISTER_ACCESS	PC1394_LINK_REGISTER_ACCESS	sizeof(PC1394_LINK_REGISTER_ACCESS)
OID_DEF_AR_PACKET_TRANSFER	ULONG *	sizeof(ULONG)
OID_MULTIDMA_MODE	ULONG *	sizeof(ULONG)
OID_AR_PACKET_TRANSFER	C1394_AR_PACKET_TRANSFER	sizeof(C1394_AR_PACKET_TRANSFER)
OID_CHANNEL_MASK	CHANNEL_MASK_STRUCT	sizeof(CHANNEL_MASK_STRUCT)
OID_CHANNEL_MASK_ENABLE	CHANNEL_MASK_STRUCT	sizeof(CHANNEL_MASK_STRUCT)
OID_CHANNEL_MASK_DISABLE	CHANNEL_MASK_STRUCT	sizeof(CHANNEL_MASK_STRUCT)

The *power management capable* bit and the *serial bus manager capable* bit are “sticky”. Once set, they remain set until they are cleared, and they are not affected by 1394 bus resets, or the termination of the application that set them.

Additionally, there is no way to prevent another client driver or application from setting/clearing these bits. They are meant for use by 1394 management software and are not intended for general application use.

The **CHANNEL_MASK_STRUCT** type is defined as shown below:

```
typedef struct
{
```

```

// The handle of the adapter channel.
C1394_CHANNEL_HANDLE    ChannelHandle;

// The 64-bit mask containing the channel numbers to enable for this channel.
ULONGLONG    UEnabledChannelNumbers;
}
CHANNEL_MASK_STRUCT, *PCHANNEL_MASK_STRUCT;

```

The mask set with the channel applies to the specified adapter channel, and are also ORed-in the mask of enabled channel numbers that the class driver maintains for all the adapter channels. A given isochronous channel number can only be enabled for a single adapter channel.

The channel numbers that are enabled for an adapter channel, are **not cleared** by bus resets. They are only cleared when either the application explicitly requests so with another **C1394SetInformation** call or when the adapter channel is closed.

If the *UEnabledChannelNumbers* field contains a 1 in bit position N (0..63), then channel N is enabled for this adapter channel. This means that the application can queue isochronous requests to the adapter channel that use this channel number.

For example, to enable channel numbers 0, 11 and 39 the *UEnabledChannelNumbers* field should be set to the value shown below:

```
((ULONGLONG)1)<<0) | ((ULONGLONG)1)<<11) | ((ULONGLONG)1)<<39)
```

Note that enabling and disabling channel numbers is performed through a single operation. This means that if you have enabled channels 0, 11 and 39 as shown above, and later on you also want to enable 42, then you will have to specify the value:

```
((ULONGLONG)1)<<0) | ((ULONGLONG)1)<<11) | ((ULONGLONG)1)<<39) | ((ULONGLONG)1)<<42)
```

If you only specify the value `((ULONGLONG)1)<<42)`, then 0, 11 and 39 would become disabled.

Additionally note that, enabling/disabling channel numbers can only be performed when there are no pending isochronous requests on an adapter channel. Otherwise the operation will fail, and **C1394SetInformation** will return `STATUS_1394_DEVICE_BUSY`.

The reason for doing this is that a change in the channel mask only affects any new isochronous requests that get submitted to the class driver after the channel mask is set.

For example, if channel 0 is enabled and there are 5 queued requests for isochronous receive with this channel number, then if the application disables channel 0 and enables another channel number, the pending requests for channel number 0 will not be cancelled by the 1394 stack.

If at that time, another application tries to enable channel zero for transmit/receive on another adapter channel, and queues a request, then the adapter hardware may get confused over this situation and even hung.

See Also

C1394QueryInformation, **C1394OpenAdapterChannel**, **C1394IsochCancel**

C1394GetAdapterMaxRec

Returns the value of the *max_rec* field in the *Bus_Info_Block* of the specified adapter.

```
ULONG C1394GetAdapterMaxRec(
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose *max_rec* value is requested.

Return Values

If the adapter handle is invalid then the return value is 0xFFFFFFFF.

Remarks

The *max_rec* field in the *Bus_Info_Block* of configuration ROM puts an additional limit to the maximum payload size of incoming block write requests, and outgoing block read responses. The maximum data payload is equal to 2^{max_rec+1} bytes, and the maximum values are shown in the table below (copied from P1394A 2.1, paragraph 9.23 (P1394a 2.0 paragraph 9.18)).

<i>max_rec</i>	Maximum Payload (bytes)
0	Not Specified
1	4
2	8
3	16
4	32
5	64
6	128
7	256
8	512
9	1024
A ₁₆	2048
B ₁₆	4096
C ₁₆	8192
D ₁₆	16384
E ₁₆ and F ₁₆	Reserved

Note that this limit applies only to asynchronous payloads. Isochronous operations are not affected by the *max_rec* field.

The value of *max_rec* can be also derived by calling **C1394QueryInformation**, and specifying the **OID_MAX_REC** object identifier. The **C1394GetAdapterMaxRec** function is available for ease of use, since the class driver has this information cached for each adapter, and can return it even if there is a bus reset on the adapter at the time of the call. Additionally this function yields better performance as it avoids making passing the call to the miniport who is controlling the adapter.

See Also

C1394QueryInformation, **C1394GetAdapterSpeed**, **C1394GetMaxPayloadForMaxRec**, **C1394GetMaxPayloadForSpeed**, **C1394GetMaxSpeedToNode**

C1394GetAdapterGUID

Retrieves the GUID of the specified adapter.

```
void C1394GetAdapterGUID(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle,  
    OUT PC1394_GUID pGUID  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose GUID is requested.

pGUID

A pointer to a **C1394_GUID** structure where the GUID will be stored.

Return Values

If the memory pointed to by *pGUID* is not accessible then the function will fail silently, that is it will return without performing any actions.

If the adapter handle is invalid then the GUID returned will be all zeros.

Remarks

The function retrieves the adapter's GUID without a switch to kernel mode, so it is a more efficient way to retrieve the GUID of an adapter handle than calling **C1394QueryInformation** with **OID_GUID**.

See Also

C1394QueryInformation

C1394GetAdapterNodeID

Returns the current NodeID of the specified adapter.

```
C1394_NODE_ID C1394GetAdapterNodeID(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose current NodeID is requested.

Return Values

If the adapter handle is invalid or there is a bus reset in progress then the NodeID returned has its BusID field equal to 1023 and the PhysicalID field equal to 63 (which results in NodeID.Value being equal to (unsigned short)0xFFFF).

Remarks

Calling this function is functionally equivalent to calling **C1394QueryInformation** with the **OID_NODE_ID** object identifier, but using **C1394GetAdapterNodeID** is easier to use and yields better performance since the class driver maintains this information and does not have to pass the call to the miniport driver that controls the adapter.

See Also

C1394QueryInformation, **C1394IsBusResetInProgress**

C1394GetAdapterSpeed

Returns the maximum transmission speed of the specified adapter.

```
C1394_SPEED_CODE C1394GetAdapterSpeed(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose maximum transmission speed is requested.

Return Values

If the adapter handle is invalid then the return value is `SPEED_CODE_INVALID`.

Remarks

The function returns the minimum of the adapter's link and PHY speed. The same information can be derived by calling **C1394QueryInformation** twice, specifying the **OID_LINK_SPEED** and **OID_PHY_SPEED** object identifiers, and taking the minimum of the returned values.

This function is available for ease of use, since the class driver has this information cached for each adapter, and additionally it yields better performance as it avoids making any calls to the miniport who is controlling the adapter.

See Also

C1394QueryInformation, **C1394GetAdapterMaxRec**, **C1394GetMaxPayloadForSpeed**, **C1394GetMaxSpeedToNode**

C1394GetCycleTime

Returns the value of the CYCLE_TIME register.

```
ULONG C1394GetCycleTime(
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter whose cycle timer value is requested.

Return Values

If *C1394AdapterHandle* is an invalid adapter handle then 0 is returned. Otherwise the cycle timer value is returned as a ULONG in the machine's native endianness.

Remarks

The structure of the CYCLE_TIME register is described in 1394-1995 paragraph 8.3.2.3.1.

The return value can be assigned to the *uValue* member of the **C1394_CYCLE_TIME_REGISTER** structure, so that the structure bit fields can be used to parse the information in the cycle timer. This structure is defined as shown below:

```
typedef union
{
    ULONG    uValue;

    struct
    {
        ULONG CycleOffset:12;
        ULONG CycleCount:13;
        ULONG SecondCount:7;
    };
}
C1394_CYCLE_TIME_REGISTER, *PC1394_CYCLE_TIME_REGISTER;
```

The 12-bit *CycleOffset* field shall be updated on each tick of the local 24.576 MHz PHY clock, with the exception that an increment from the value 3071 shall cause a wraparound to zero and shall carry into the *CycleCount* field.

The value is the fractional part of the isochronous cycle of the current time, in units that are counts of the 24.576 MHz clock.

The 13-bit *CycleCount* field shall increment on each carry from the *CycleOffset* field, with the exception that an increment from the value 7999 shall cause a wraparound to zero and shall carry into the *second_count* field.

The value is the fractional part of the second of the current time, in units of 125 microsec.

The 7-bit *SecondCount* field shall increment on each carry from the *CycleCount* field, with the exception that an increment from the value 127 shall cause a wraparound to zero.

For information on how to calculate the difference between two values of the CYCLE_TIME register see the related sample in the introductory section of this document.

See Also

C1394GetIRMNodeID

Returns the 16-bit node identifier of the node currently elected as the Isochronous Resource Manager (IRM).

```
C1394_NODE_ID C1394GetIRMNodeID(  
    IN          C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN          C1394_BUS_ID          BusID  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the class driver the adapter through which to look for the bus specified.

BusID

The 10-bit bus identifier of the bus of interest.

Return Values

If the bus specified is unknown, or its IRM is unknown to the class driver or there is no IRM on the bus then the node ID (1023,63) is returned (RetVal.Value==0xFFFF). The same value is returned if *C1394AdapterHandle* is invalid.

Remarks

The node selected as the IRM is the node with the greater physical ID that had its *contender* bit set in the self ID packets sent after the last bus reset.

Unless there was some error in the self ID packets, the class driver always has accurate information for the local 1394 bus (BUS_ID==1023). Since the class driver is IRM-capable, and always sets the *contender* bit in the local node's self ID packets, there will always be an active IRM on the bus. A return value of (1023,63) for the local 1394 bus identifier, indicates that there was some serious error in the self ID packets, which prevented the class driver from determining which node is the IRM.

See Also

C1394GetAdapterNodeID, C1394GetRootNodeID

C1394GetRootNodeID

Returns the 16-bit node identifier of the node currently elected as the Isochronous Resource Manager (IRM).

```
C1394_NODE_ID C1394GetRootNodeID(  
    IN      C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN      C1394_BUS_ID          BusID  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the class driver the adapter through which to look for the bus specified.

BusID

The 10-bit bus identifier of the bus of interest.

Return Values

If the bus specified is unknown, or its root node is unknown to the class driver then the node ID (1023,63) is returned (RetVal.Value==0xFFFF). The same value is returned if *C1394AdapterHandle* is invalid.

Remarks

Unless there was some error in the self ID packets, the class driver always has accurate information for the local 1394 bus (BUS_ID==1023). A return value of (1023,63) for the local 1394 bus identifier (**LOCAL_1394_BUS_ID**), indicates that there was some serious error in the self ID packets, which prevented the class driver from determining which nodes are present on the bus.

See Also

C1394GetAdapterNodeID, **C1394GetIRMNodeID**

C1394GetMaxPayloadForMaxRec

Returns the maximum payload that can be transmitted with an asynchronous packet to a node that has the specified *max_rec* value in its configuration ROM.

```
ULONG C1394GetMaxPayloadForMaxRec( ULONG max_rec );
```

Parameters

max_rec

The value to calculate the payload for.

Return Values

If *max_rec* is 0, E₁₆ or F₁₆ (reserved values), or any other invalid value, then 0 is returned.

Remarks

This function is actually implemented as a macro for better performance.

See Also

C1394GetMaxPayloadForSpeed

C1394GetMaxPayloadForSpeed

Returns the maximum payload that can be transmitted with an asynchronous packet at the specified speed.

```
ULONG C1394GetMaxPayloadForSpeed( C1394_SPEED_CODE SpeedCode );
```

Parameters

SpeedCode

The speed code to calculate the result for.

Return Values

If a speed code other than S100, S200, S400, S800, S1600 or S3200 is specified, then the return value is 0.

Remarks

This function is actually implemented as a macro for better performance.

C1394GetMaxSpeedBetweenNodes

Returns the maximum transmission rate that can be used in transmissions between any of the nodes specified, through the adapter specified.

```
C1394_SPEED_CODE C1394GetMaxSpeedBetweenNodes (
    IN C1394_ADAPTER_HANDLE AdapterHandle,
    IN C1394_NODE_ID *Nodes,
    IN ULONG uNumberOfNodes
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter through which to make the calculation.

Nodes

An array containing the 16-bit node identifiers of the nodes among which to calculate the maximum transmission speed. If NULL, then all nodes present on the local 1394 bus are implied. If non-NULL, then this array should contain at least one item.

uNumberOfNodes

The number of items in the *Nodes* array. The minimum legal value is 1. If *Nodes* is NULL then this parameter is ignored.

Return Values

Any node that does not belong to the local 1394 bus is ignored in the calculations⁵⁰.

Any destination 16-bit node ID, identifying a device on the local 1394 bus that is not physically present, will also be ignored.

If the adapter handle specified is invalid, or there is a bus reset in progress on the adapter, or the number of items that will be used in the calculation are zero, then **SPEED_CODE_INVALID** is returned. Otherwise the maximum transmission rate that can be used between any two devices in the *Nodes* array is returned.

Remarks

This function is usually called in order to calculate the transmission speed of an isochronous channel that should be received by all nodes in a given set.

Calculating the maximum speed between a subset of the nodes on the local bus is not an expensive operation because the class driver already has calculated the maximum transmission rate between all pairs of nodes. Since the bus topology is a tree (no loops) the maximum speed between a set of nodes is calculated using the formula below:

$$MaxSpeedBetween\{N_1, N_2, \dots, N_n\} = Min\{ Speed(N_1, N_2), Speed(N_1, N_3), \dots, Speed(N_1, N_n)\}$$

If a client wants to find the maximum transmission speed from the local node to another node on the bus, then it should better use the **C1394GetMaxSpeedToNode** function for better performance.

If **C1394GetMaxSpeedBetweenNodes** is called with a single argument then it returns the actual speed capabilities of the specified node on the bus. This is the same value that **C1394GetNodeSpeed** returns.

See Also

C1394GetMaxSpeedToNode, **C1394GetMaxPayloadForMaxRec**, **C1394GetMaxPayloadForSpeed**, **C1394GetAdapterSpeed**

⁵⁰ This behaviour will be refined in future releases, when the 1394 bus bridge proposal achieves a more complete and stable status.

C1394GetMaxSpeedToNode

Returns the maximum transmission speed that can be used in a transmission to the destination node through the adapter specified.

```
C1394_SPEED_CODE C1394GetMaxSpeedToNode(
    IN C1394_ADAPTER_HANDLE AdapterHandle,
    IN C1394_NODE_ID Destination
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter through which to make the calculation.

Destination

The 16-bit node identifier of the destination node for which to return the maximum transmission speed.

Return Values

If the adapter handle specified is invalid, or there is a bus reset in progress on this adapter, or the destination node is on a remote 1394 bus⁵¹, or the destination node is not physically present on the local 1394 bus, then **SPEED_CODE_INVALID** is returned.

If the destination node is on the local 1394 bus, then the class driver will return the speed code of the slowest device on the path from the local node to the destination node.

Remarks

If an application wants to calculate the maximum payload that it can physically transmit to a destination node then it should use the return value of this function as a parameter to

C1394GetMaxPayloadForSpeed.

However this refers to maximum physical transmission capability. The destination node may not be able to handle maximum sized packets in its transaction layer. The maximum payload that a node can accept (provided that it can be physically transmitted to it) is defined by the value of the *max_rec* field in its configuration ROM. This value is supposed to stay constant between bus resets⁵² so applications can cache this value for various destination nodes, once they have retrieved it.

The result of this function can also be calculated by calling **C1394GetMaxSpeedBetweenNodes** using an array containing the node identifier of the local node and the destination node. However this would be extremely inefficient, since **C1394GetMaxSpeedBetweenNodes** has to run a relatively complex graph algorithm in order to return its results.

The maximum speed between the local node and each other node on the local bus, is a piece of information that will be requested too often, so the class driver performs this calculation after each bus reset and caches these results for better performance.

See Also

C1394GetMaxPayloadForMaxRec, **C1394GetMaxPayloadForSpeed**, **C1394GetAdapterSpeed**

⁵¹ This behaviour will be refined in future releases, when the 1394 bus bridge proposal achieves a more complete and stable status.

⁵² A change to any value in the configuration ROM should be followed by a bus reset.

C1394GetNodeSpeed

Returns the speed code of the maximum transmission rate that a node is capable of achieving.

```
C1394_SPEED_CODE C1394GetNodeSpeed(  
    IN C1394_ADAPTER_HANDLE  C1394AdapterHandle,  
    IN C1394_NODE_ID         NodeID  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter from which to retrieve the information.

NodeID

The 16-bit node identifier of the node whose transmission rate capabilities are being queried.

Return Values

If *C1394AdapterHandle* is invalid or the node specified by *NodeID* is not present on the bus, then **SPEED_CODE_INVALID** is returned. Otherwise the class driver looks up the node's information in its internal speed table and returns the speed code of the node's maximum transmit rate.

Remarks

The speed capabilities of a node on the bus might differ from the transmission rate at which the local node can communicate with that node, because slower devices might be on the path between the two nodes. Use the **C1394GetMaxSpeedToNode** function to retrieve that information.

C1394GetNodeSpeed is meant simply for convenience and clarity of code. It simply calls **C1394GetMaxSpeedBetweenNodes** with a single argument.

See Also

C1394GetMaxSpeedBetweenNodes, **C1394GetMaxSpeedToNode**

C1394GetPhyPacketType

Returns the exact type of the specified PHY packet.

```
C1394_PHY_PACKET_TYPE C1394GetPhyPacketType(
    PC1394_PHY_PACKET    pPhyPacket
);
```

Parameters

pPhyPacket

A pointer to a PHY packet whose type is to be determined.

Return Values

The return values are taken from the **C1394_PHY_PACKET_TYPE** enumeration. See the remarks section for more details.

Remarks

The **C1394_PHY_PACKET_TYPE** is defined as shown below:

```
typedef enum
{
    PhyPacketInvalid,
    PhyPacketSelfID0,
    PhyPacketSelfID1,
    PhyPacketSelfID2,
    PhyPacketSelfID3,
    PhyPacketLinkOn,
    PhyPacketConfiguration,
    PhyPacketPing,
    PhyReadBaseRegister,
    PhyReadPagedRegister,
    PhyBaseRegisterContents,
    PhyPagedRegisterContents,
    PhyPacketRemoteCommand,
    PhyPacketRemoteConfirmation,
    PhyPacketResume
}
C1394_PHY_PACKET_TYPE;
```

For each possible PHY packet type, **C1394GetPhyPacketType** will make certain that it has the correct structure (the second quadlet is the complement of the first) and that all (if any) reserved fields have a zero value. If any field is not valid then *PhyPacketInvalid* is returned.

This function correctly identifies all the extended PHY packet types defined by P1394A.

See Also

C1394TransmitRaw

C1394GetTransactionType

A useful macro that returns the generic transaction type (request, response, stream, none, invalid) of the specified transaction code.

```
ULONG C1394GetTransactionType(  
    C1394_TRANSACTION_CODE TransactionCode  
);
```

Parameters

TransactionCode

The transaction code whose type is to be determined.

Return Values

The return values are shown in the table below:

Return Value	Description
TRANSACTION_TYPE_NONE	Invalid or reserved transaction code.
TRANSACTION_TYPE_REQUEST	Request packet transaction code (block write request, quadlet write request, lock request, block read request, quadlet read request)
TRANSACTION_TYPE_RESPONSE	Response packet transaction code (write response, lock response, block read response, quadlet read response)
TRANSACTION_TYPE_STREAM	Stream packet.

Remarks

This macro is useful for quick switching depending on the generic type of the transaction request in hand. It is the most efficient way of deciding the generic transaction type since it is implemented using an internal lookup table.

See Also

C1394GetExpectedResponseCode, **C1394IsTransactionCodeLegal**, **C1394IsResponseCodeLegal**

C1394GetExpectedResponseCode

Returns the expected transaction response code for a transaction request.

```
C1394_TRANSACTION_CODE C1394GetExpectedResponseCode(  
    IN C1394_TRANSACTION_CODE TransactionRequestCode  
);
```

Parameters

TransactionRequestCode

The transaction request code for which to return the appropriate transaction response code.

Return Values

If *TransactionRequestCode* is a valid transaction code and indeed a transaction request code, then the appropriate transaction response code is returned. Otherwise `0xFF` is returned.

Remarks

This macro should be used when generating response packets, so that the response code is the correct for the transaction request in question.

It is the most efficient way of determining the correct response code, since it is implemented using an internal lookup table.

See Also

C1394GetTransactionType, **C1394IsTransactionCodeLegal**, **C1394IsResponseCodeLegal**

C1394IsResponseCodeLegal

Indicates whether the value passed as a parameter is a valid transaction code.

```
BOOLEAN C1394IsResponseCodeLegal(  
    IN C1394_RESPONSE_CODE ResponseCode  
);
```

Parameters

ResponseCode

The value to be checked for validity.

Return Values

If *ResponseCode* specifies a valid response code, then the return value is TRUE, otherwise it is FALSE.

Remarks

This macro is the most efficient way for checking the validity of a response code value, since it is implemented using an internal lookup table.

See Also

C1394GetExpectedResponseCode, **C1394GetTransactionType**, **C1394IsTransactionCodeLegal**

C1394IsTransactionCodeLegal

Indicates whether the value passed as a parameter is a valid transaction code.

```
BOOLEAN C1394IsTransactionCodeLegal(  
    IN C1394_TRANSACTION_CODE TransactionCode  
);
```

Parameters

TransactionCode

The value to be checked for validity.

Return Values

If *TransactionCode* specifies a valid transaction code, then the return value is TRUE, otherwise it is FALSE.

Remarks

This macro is the most efficient way for checking the validity of a transaction code value, since it is implemented using an internal lookup table.

See Also

C1394GetExpectedResponseCode, C1394GetTransactionType, C1394IsResponseCodeLegal

Event Notification Functions

C1394RegisterNotification

Registers a notification for the specified 1394-event type.

```
STATUS_1394 C1394RegisterNotification(
    IN  C1394_ADAPTER_HANDLE           C1394AdapterHandle,
    IN  C1394_EVENT_TYPE               EventType,
    IN  PVOID                           pNotificationSettings,
    IN  PVOID                           Context,
    IN  C1394_EVENT_INDICATION_HANDLER EventIndicationHandler
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which to register the notification.

EventType

The code of the 1394-event of interest.

pNotificationSettings

A pointer to an event type specific structure that contains additional parameters for the registration of the notification.

Context

A pointer value that will be passed as additional caller-specified context to the event notification handler.

EventIndicationHandler

A callback function that can be called to handle this event. This function pointer can be NULL.

Return Values

If the function is successful, then **STATUS_1394_SUCCESS** is returned. Otherwise the appropriate error code is returned and **pNotificationHandle* is set to NULL.

Remarks

Applications receive 1394-event notifications, through a Win32 event object. Each application that has called **C1394Initialize** has an associated auto-reset event object. When a 1394-event should be indicated to the application and the application's 1394-event queue is empty, then the Win32 event object is set. The application should test this object, and when it finds that it is signalled, then it should repeatedly call **C1394GetAsynchEvent** until **STATUS_1394_NOT_FOUND** is returned. This is the indication that there are no more 1394-event notifications for this application.

If the application has specified a valid function pointer in the *EventIndicationHandler* parameter then **C1394GetAsynchEvent** will automatically call this function when it retrieves a 1394-event of the associated type, without returning to the application. If NULL is specified as the *EventIndicationHandler* then **C1394GetAsynchEvent** returns **STATUS_1394_SUCCESS** and an information record is returned to the application. The application should then act upon this event as it needs to, and then call **C1394GetAsynchEvent** again until **STATUS_1394_NOT_FOUND** is returned.

Note that the Win32 event object used for asynchronous notification signalling is auto-reset and is set only upon the first 1394-event to be inserted in the queue. This means that once this event is signalled, the application should not wait on it again until **C1394GetAsynchEvent** returns **STATUS_1394_NOT_FOUND**. Otherwise the waiting thread will either timeout, or wait for ever⁵³.

⁵³ Depending on the timeout value specified in the call to the Win32 functions **WaitForSingleObject** or **WaitForMultipleObjects** (or any other wrapper to these functions).

An application can get the handle of the asynchronous notification event object by calling function **C1394GetAsynchEventHandle**. This event object is created by **C1394Initialize** and deleted by **C1394Terminate**, so the application should not call the Win32 function **CloseHandle** with its handle, unless it has called the Win32 function **DuplicateHandle** first. Otherwise it will not be able to receive asynchronous event notifications.

The **C1394_EVENT_INDICATION_HANDLER** type is defined as follows:

```
typedef void (*C1394_EVENT_INDICATION_HANDLER) (
    IN CLIENT_ADAPTER_HANDLE ClientAdapterHandle,
    IN C1394_EVENT_TYPE EventType,
    IN PC1394_EVENT_PARAMETERS_STRUCT pEventParameters,
    IN PVOID Context
);
```

Each application can only have a single notification handler for each 1394-event type. If **C1394RegisterNotification** is called twice for the same 1394-event type, then the second call overrides the settings of the first call.

The possible values for *EventType* that applications can specify, together with their meanings are shown in the table that follows.

NOTE: In the current version of FireAPI only the *EventPhyBusResetStart*, *EventPhyBusResetComplete* and *EventMiniportAdapterRemoved* events are enabled for user mode applications.

EventType	Meaning
EventPhyBusResetStart	A bus reset has started on the local network or a remote network.
EventPhyBusResetComplete	The bus reset on the local network has completed.
EventMiniportAdapterRemoved	The 1394 adapter has been removed from the system. This can happen if the 1394 driver stack is unloaded for a specified adapter.
EventPhyConfigTimeout	The network topology includes a loop and a configuration timeout has occurred.
EventPhyCablePowerFail	There is loss of cable power.
EventPhyPortStatusChanged	There is a change in the status of some port. A separate indication is provided for each port whose status has changed. If the port is being resumed on a boundary node then that node should cause a bus reset soon so that the new bus topology is discovered. In that case the port status change notification is delivered prior to the bus reset notification. As stated in P1394a paragraph 3.5.4 in some occasions resumption of a suspended port causes all other suspended ports to be also resumed. In that case the notifications for each port are provided separately. Not all miniport drivers support this feature, so trying to install a notification for this event type might return STATUS_1394_NOT_SUPPORTED .
EventPhyPortSuspendResumeFault	There was a fault generated during a port suspend/resume handshake. This corresponds to the <i>Fault</i> field in the PHY register page 0 (P1394a paragraph 6.1 – Table 6.2). Not all miniport drivers support this feature, so trying to install a notification for this event type might return STATUS_1394_NOT_SUPPORTED .

EventPhyPHYPacket	A PHY packet has been received. This is the indication that clients drivers should use to perform remote PHY register reads and remote commands. They should register an event notification with C1394RegisterNotification and then use C1394TransmitRaw or its wrapper, C1394TransmitPHYPacket , to queue the packet for transmission. When the packet is actually transmitted and the response PHY packet arrives, then the event notification callback will be called to process the response. The PhyPHYPacket indication is not provided during the bus reset phase, when self-ID packets are being sent.
EventPhyPingResponse	A response to a ping request has arrived. Although this is a self-ID packet that will be indicated with the PhyPHYPacket event, the PhyPingResponse indication contains the time that it took the self-ID packet to arrive in response to the ping PHY packet that was sent by a previous call to C1394PingNode .
EventLinkCycleTooLong	The last isochronous cycle was too long. More than ISOCHRONOUS_CYCLE_TIME passed and a subaction gap was not detected.
EventLinkCycleStart	A cycle start packet is received. This event should only be indicated for the first cycle start packet that is received after a cycle lost event, or the first cycle start packet that is ever received.
EventLinkCycleLost	A cycle start packet was not received in the expected time. <i>Note: Many adapters may indicate this condition as an interrupt on each cycle. This event is only indicated the first time it occurs after a cycle start packet is received.</i>
EventLinkDuplicateChannel	More than one isochronous packets with the same channel number were detected on the network during the last isochronous cycle.
EventLinkBusOccupancyViolation	A node held control of the bus for too long.
EventLinkUnknownTransactionCode	A valid packet (CRC-wise) was received with an unknown transaction code.
EventLinkHeaderCRCError	A packet was received that did not pass the header CRC check.
EventMiniportAdapterHardwareError	A recoverable hardware error occurred on the adapter.
EventMiniportAdapterHung	An unrecoverable error has occurred on the adapter and its operation has stopped.
EventMiniportAdapterReset	The miniport has reset the adapter in order to recover from some serious error condition.
EventConfigGapCountChange	A PHY configuration packet was received that changed the current value of <i>gap_count</i> .
EventConfigCycleMasterActivated	The adapter has been activated as the cycle master.
EventConfigCSRChange	A CSR or an address range has been changed.
EventConfigGUIDsResolved	The class driver has completed the resolution of GUIDs for the specified bus, which means that clients can proceed and call the C1394ResolveGUID and C1394GetNodeGUID functions.
EventConfigDeviceRemoval	A device to which there was an open handle has been removed from the bus.
EventIrmChannelAllocated	A channel has been allocated.
EventIrmChannelFreed	A channel has been freed.

EventIrmBandwidthAllocated	Bandwidth has been allocated.
EventIrmBandwidthFreed	Bandwidth has been freed.
EventIrmUnexpectedChannel	An isochronous packet was received for a channel that is free in the CHANNELS_AVAILABLE register.

See Also**C1394OpenAdapter, C1394CloseAdapter, C1394UnregisterNotification**

C1394UnregisterNotification

Unregisters a notification that was registered with a previous call to **C1394RegisterNotification**.

```
void C1394UnregisterNotification (
    IN  C1394_ADAPTER_HANDLE  C1394AdapterHandle,
    IN  C1394_EVENT_TYPE      EventType
);
```

Parameters

C1394AdapterHandle

A handle identifying to the 1394 stack the adapter on which the operation should take place.

EventType

The code of the 1394-event of interest.

Remarks

All registered 1394-event notifications are automatically freed when an application closes an adapter handle with **C1394CloseAdapter** (in the same way that address range mappings also get freed). However it is suggested as the proper practice to unregister all notifications from an adapter before closing its handle.

Applications are assured that they will no more receive notifications about a given 1394-event from the moment the call to **C1394UnregisterNotification** returns, even if an instance of the 1394-event had already been queued in the application's 1394-event queue before the application called **C1394UnregisterNotification**.

See Also

C1394RegisterNotification, **C1394GetAsynchEventHandle**, **C1394GetAsynchEvent**, **C1394CloseAdapter**

C1394GetAsynchEvent

Returns information about an asynchronous 1394 event notification and/or calls the associated event callbacks that the application has installed with **C1394RegisterNotification**.

```
STATUS_1394 C1394GetAsynchEvent (  
    IN  C1394_ADAPTER_HANDLE          C1394AdapterHandle,  
    OUT PC1394_EVENT_PARAMETERS_STRUCT pEventInformation  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter from which to retrieve the next asynchronous event notification.

pEventNotification

A pointer to a structure of type **C1394_EVENT_PARAMETERS_STRUCT** that will be filled with information about the event, like the event code, and possibly other information if available.

Return Values

If there is an event notification in the application's event queue for this adapter, then the function returns **STATUS_1394_SUCCESS**.

If there are no more events, then the function returns **STATUS_1394_NOT_FOUND**.

Otherwise the function returns an error code according to the guidelines specified in Status Code Reference.

Remarks

Applications retrieve their asynchronous 1394-event notifications in a similar fashion to the processing of transaction requests for address ranges. When the first event is queued, the application's event object for this adapter's asynchronous events, is set.

Then the application should repeatedly call **C1394GetAsynchEvent** until **STATUS_1394_NOT_FOUND** is returned.

See function **C1394RegisterNotification** for more information.

In cases where the application installs event notification callbacks for all the asynchronous events that it has registered, then **C1394GetAsynchEvent** will always return **STATUS_1394_NOT_FOUND**.

However it must be called because it internally performs a loop that empties the asynchronous event queue and calls the appropriate callbacks.

NOTE: While developing you might have successfully called **C1394RegisterNotification** for an event, but your event callback never gets called. This symptom is a common error, and it is the result of forgetting to call **C1394GetAsynchEvent**.

See Also

C1394GetAsynchEventHandle, **C1394RegisterNotification**, **C1394UnregisterNotification**

C1394GetAsynchEventHandle

Returns the handle to the Win32 event object that is used for notifying an application about the asynchronous 1394-events of its interest.

```
HANDLE C1394GetAsynchEventHandle(  
    IN C1394_ADAPTER_HANDLE C1394AdapterHandle  
);
```

Parameters

C1394AdapterHandle

A handle that identifies to the 1394 stack the adapter from which to retrieve the Win32 event handle.

Return Values

If the parameter is valid, then a non-NULL handle is returned. Otherwise NULL is returned.

Remarks

Applications retrieve their asynchronous 1394-event notifications in a similar fashion to the processing of transaction requests for address ranges. When the first 1394-event notification is queued, the application's event object for this adapter's asynchronous events is set. This is precisely the Win32 event object returned by **C1394GetAsynchEventHandle**.

See function **C1394RegisterNotification** for more information.

See Also

C1394GetAsynchEvent, **C1394RegisterNotification**, **C1394UnregisterNotification**

C1394GetAddAdapterEventHandle

Returns the handle to the Win32 event object that is used for notifying an application about the addition of a 1394 adapter in the system.

```
HANDLE C1394GetAddAdapterEventHandle(void);
```

Parameters

none

Return Values

The function always returns a valid handle.

Remarks

The event returned is set when a new adapter is added to the system. WDM drivers can be loaded and unloaded on demand by the user without a reboot of the system. User mode FireAPI applications should use this event in order to be signaled when an adapter is added to the system. 1394 pci adapters are added to the system at boot time and are not removed or added again unless there is a specific request by the user for the drivers to be unloaded and reloaded. If the 1394 adapter is not added and removed from the system frequently for some other reasons there is not an imperative need to listen to this event.

The event is created by **C1394Initialize** and if its creation fails then **C1394Initialize** also fails. That's why **C1394GetAddAdapterEventHandle** always returns a valid handle, as long as FireAPI has been initialized. The event handle is closed by **C1394Terminate** so there is no need for the user to close this handle.

The event is initially not signalled. When an application starts up it should open all available adapters and then monitor this event for the addition of new adapters. When the event is signalled the application should retrieve all adapters by calling **C1394GetAdapters** and then compare the GUIDs reported to the GUIDs of the adapters that it has already opened and find out which are the new adapters. You can easily retrieve the GUID of any open adapter handle by calling **C1394GetAdapterGUID**.

The reason that notifications for new adapters do not use the standard mechanism provided by **C1394RegisterNotification** is that in order to call **C1394RegisterNotification** you need to have an adapter handle available, and you obviously can't have an adapter handle for an adapter that has not yet been added to the system.

See Also

C1394GetAsynchEvent, **C1394RegisterNotification**, **C1394UnregisterNotification**

Miscellaneous Functions

C1394AddBigEndian32

C1394AddBigEndian64

Adds two big endian values and returns the 32-bit or 64-bit result in big endian.

```
void C1394AddBigEndian32(  
    OUT void *pResult,  
    IN  void *pOp1,  
    IN  void *pOp2  
);  
  
void C1394AddBigEndian64(  
    OUT void *pResult,  
    IN  void *pOp1,  
    IN  void *pOp2  
);
```

Parameters

pResult

Pointer to a 32-bit or 64-bit variable that will receive the result in big endian format.

pOp1

Pointer to a 32-bit or 64-bit variable that contains the first operand (in big endian).

pOp2

Pointer to a 32-bit or 64-bit variable that contains the second operand (in big endian).

Remarks

Both operands are assumed to already be in big endian format when this function is called.

See Also

SwapEndian32, **SwapEndian64**, **BlockSwapEndian32**

C1394CalculateCRC16

Runs the CRC-16 algorithm and calculates a 16-bit CRC value for the specified data.

```
void C1394CalculateCRC16(  
    IN void *pData,  
    IN ULONG uDoublets,  
    OUT UCHAR CRC[2]  
);
```

Parameters

pData

A pointer to the memory area containing the data over which to calculate the CRC value.

uDoublets

The size of the data expressed in doublets (1 doublet = 16-bits = 2 bytes).

CRC

An array of two bytes into which the 16-bit CRC value will be returned.

Remarks

The CRC-16 algorithm is implemented as described in clause 8.1.5 in the CSR Architecture IEEE standard.

Both the input data and the resulting 16-bit CRC are treated as big endian. This is the reason why the CRC parameter is declared as a 2-byte array, and not an unsigned short integer (who would be platform-endianess dependent).

If *uDoublets* is zero, then the resulting CRC will also be zero.

See Also

C1394CalculateLinkCRC

C1394CalculateCRC8

Runs the CRC-8 algorithm over a block of memory and calculates an 8-bit CRC value.

```
BYTE C1394CalculateCRC8(  
    IN void *Buffer,  
    IN ULONG Bytes  
);
```

Parameters

Buffer

A pointer to the memory area containing the data over which to calculate the CRC-8 value.

Bytes

The number of bytes to include in the calculation.

Remarks

The CRC-8 algorithm is implemented as described in annex C of the VersaPhy 1.0 standard.

This function expects the *Buffer* parameter to be valid and accessible for at least *Bytes* bytes. If this is not true then an access violation will occur in user mode applications and a zero CRC will be returned for kernel mode applications (kernel breakpoint if the debug drivers are being used).

When used to calculate the CRC of a VersaPhy packet, the packet must be in big-endian format and the value of *Bytes* specified must be 7.

C1394CalculateLinkCRC

Runs the 32-bit CRC algorithm that is used by the 1394 Link Layer in the generation and checking of header and data CRCs.

```
void C1394CalculateLinkCRC(  
    IN void      *pData,  
    IN ULONG     uQuadlets,  
    OUT UCHAR    CRC[ 4 ]  
);
```

Parameters

pData

A pointer to the memory area containing the data over which to calculate the CRC value.

uQuadlets

The size of the data expressed in quadlets (1 quadlet = 32-bits = 4 bytes).

CRC

An array of 4 bytes into which the 32-bit CRC value will be returned.

Remarks

The CRC algorithm is implemented as described in clause 6.4 of IEEE 1394-1995.

Both the input data and the resulting 32-bit CRC are treated as big endian. This is the reason why the CRC parameter is declared as a 4-byte array, and not an unsigned long integer (who would be platform-endianess dependent).

If *uQuadlets* is zero, then the resulting CRC will also be zero.

See Also

C1394CalculateCRC16

C1394DebugPrint

Sends a formatted string to the kernel debugger.

```
void C1394DebugPrint(char *format, ...);
```

Parameters

format

A printf-like format specifier.

Remarks

This function is an intelligent wrapper around the Win32 function **OutputDebugString**. If you would like to include the kernel debugger messages only in debug builds, then use the **KdPrint** macro which resolves to **C1394DebugPrint** if **_DEBUG** is defined, and to a no-op otherwise.

See Also

Part III

FireAPI Structures & Macros Reference

C1394_PACKET_HEADER

The **C1394_PACKET_HEADER** structure is defined as follows:

```
typedef struct
{
    // Valid in all cases.
    ULONG          uHeaderBytes;
    ULONG          data_length;
    C1394_TRANSACTION_CODE  TransactionCode;

    // Valid for asynchronous primary packets.
    C1394_NODE_ID  Source;
    C1394_NODE_ID  Destination;
    C1394_OFFSET   Offset;
    C1394_EXTENDED_TCODE  ExtendedTCode;
    C1394_TRANSACTION_LABEL  TransactionLabel;
    C1394_RESPONSE_CODE  ResponseCode;

    // Valid only for stream packets.
    C1394_CHANNEL  Channel;
    C1394_TAG      Tag;
    C1394_SY_CODE  SyCode;
}
C1394_PACKET_HEADER, *PC1394_PACKET_HEADER;
```

Incoming Packets & C1394_PACKET_HEADER

For each type of packet that can be indicated by the class driver to applications, the fields of this structure that will be filled by the class driver are described below:

Read Request for Data Quadlet

uHeaderBytes:	Either 12 or 16, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be 4.
TransactionCode:	TCODE_QUADLET_READ_REQUEST
Source, Destination:	Will be filled in.
Offset:	Will be filled in.
TransactionLabel:	Will be filled in.

Read Response for Data Quadlet

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be 4.
TransactionCode:	TCODE_QUADLET_READ_RESPONSE
Source, Destination:	Will be filled in.
TransactionLabel:	Will be filled in.
ResponseCode:	Will be filled in.

Read Request For Data Block

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Must be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_BLOCK_READ_REQUEST
Source, Destination:	Will be filled in.
Offset:	Will be filled in.
ExtendedTCode:	Zero.
TransactionLabel:	Will be filled in.

Read Response for Data Block

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_BLOCK_READ_RESPONSE
Source, Destination:	Will be filled in.
ExtendedTCode:	Zero.
TransactionLabel:	Will be filled in.
ResponseCode:	Will be filled in.

Write Request for Data Quadlet

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be 4.
TransactionCode:	TCODE_QUADLET_WRITE_REQUEST
Source, Destination:	Will be filled in.
Offset:	Will be filled in.
TransactionLabel:	Will be filled in.

Write Request for Data Block

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_BLOCK_WRITE_REQUEST
Source, Destination:	Will be filled in.
Offset:	Will be filled in.
ExtendedTCode:	Zero.
TransactionLabel:	Will be filled in.

Write Response

uHeaderBytes:	Either 12 or 16, depending on whether the adapter is also indicating the header CRC quadlet.
TransactionCode:	TCODE_WRITE_RESPONSE
Source, Destination:	Will be filled in.
TransactionLabel:	Will be filled in.
ResponseCode:	Will be filled in.

Lock Request

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Must be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_LOCK_REQUEST
Source, Destination:	Will be filled in.
Offset:	Will be filled in.
ExtendedTCode:	Zero.
TransactionLabel:	Will be filled in.

Lock Response

uHeaderBytes:	Either 16 or 20, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_LOCK_RESPONSE
Source, Destination:	Will be filled in.
ExtendedTCode:	Will be filled in.
TransactionLabel:	Will be filled in.
ResponseCode:	Will be filled in.

Stream Block

uHeaderBytes:	Either 4 or 8, depending on whether the adapter is also indicating the header CRC quadlet.
data_length:	Will be filled in from the packet's <i>data_length</i> field.
TransactionCode:	TCODE_STREAM_BLOCK
Channel:	Will be filled in.
Tag:	Will be filled in.
SyCode:	Will be filled in.

Outgoing Packets & C1394_PACKET_HEADER

For each type of packet that can be passed to the class driver by an application, the fields of this structure that must be filled by the application are described below:

Read Request for Data Quadlet

TransactionCode:	TCODE_QUADLET_READ_REQUEST
data_length:	Must be 4.
Source, Destination:	Must be filled in.
Offset:	Must be filled in.
TransactionLabel:	Either filled in automatically by the class driver, or by the application if it is using a pre-allocated label.

Read Response for Data Quadlet

TransactionCode:	TCODE_QUADLET_READ_RESPONSE
data_length:	Must be 4.
Source, Destination:	Must be filled in.
TransactionLabel:	Must be filled in.
ResponseCode:	Must be filled in.

Read Request For Data Block

data_length: Must be filled with the value that should go into the packet's *data_length* field.
 TransactionCode: **TCODE_BLOCK_READ_REQUEST**
 Source, Destination: Must be filled in.
 Offset: Must be filled in.
 ExtendedTCode: Must be filled in with zero.
 TransactionLabel: Either filled in automatically by the class driver, or by the application if it is using a pre-allocated label.

Read Response for Data Block

data_length: Must be filled with the value that should go into the packet's *data_length* field.
 TransactionCode: **TCODE_BLOCK_READ_RESPONSE**
 Source, Destination: Must be filled in.
 ExtendedTCode: Must be filled in with zero.
 TransactionLabel: Must be filled in.
 ResponseCode: Must be filled in.

Write Request for Data Quadlet

TransactionCode: **TCODE_QUADLET_WRITE_REQUEST**
data_length: Must be 4.
 Source, Destination: Must be filled in.
 Offset: Must be filled in.
 TransactionLabel: Either filled in automatically by the class driver, or by the application if it is using a pre-allocated label.

Write Request for Data Block

data_length: Must be filled with the value that should go into the packet's *data_length* field.
 TransactionCode: **TCODE_BLOCK_WRITE_REQUEST**
 Source, Destination: Must be filled in.
 Offset: Must be filled in.
 ExtendedTCode: Must be filled in with zero.
 TransactionLabel: Either filled in automatically by the class driver, or by the application if it is using a pre-allocated label.

Write Response

TransactionCode: **TCODE_WRITE_RESPONSE**
 Source, Destination: Must be filled in.
 TransactionLabel: Must be filled in.
 ResponseCode: Must be filled in.

Lock Request

<code>data_length:</code>	Must be filled with the value that should go into the packet's <i>data_length</i> field.
<code>TransactionCode:</code>	TCODE_LOCK_REQUEST
<code>Source, Destination:</code>	Must be filled in.
<code>Offset:</code>	Must be filled in.
<code>ExtendedTCode:</code>	Must be filled in.
<code>TransactionLabel:</code>	Either filled in automatically by the class driver, or by the application if it is using a pre-allocated label.

Lock Response

<code>data_length:</code>	Must be filled with the value that should go into the packet's <i>data_length</i> field.
<code>TransactionCode:</code>	TCODE_LOCK_RESPONSE
<code>Source, Destination:</code>	Must be filled in.
<code>ExtendedTCode:</code>	Must be filled in.
<code>TransactionLabel:</code>	Must be filled in.
<code>ResponseCode:</code>	Must be filled in.

Stream Block

<code>data_length:</code>	Must be filled with the value that should go into the packet's <i>data_length</i> field.
<code>TransactionCode:</code>	TCODE_STREAM_BLOCK
<code>Channel:</code>	Must be filled in.
<code>Tag:</code>	Must be filled in.
<code>SyCode:</code>	Must be filled in.

FIREAPI_ISOCH_REQUEST

This structure describes an isochronous operation request that should be executed by the 1394 stack on an adapter channel.

```
typedef struct
{
    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // Identification tag. Must be set to TAG_FIREAPI_ISOCH_REQUEST.
    IN ULONG          Tag;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE MINIPORT>
    // The op-code of the operation to execute.
    IN ULONG          uOperationCode;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION>
    // The completion status of the request. Initially it contains the value
    // STATUS_1394_PENDING.
    OUT STATUS_1394   Status;

    // <FILLED BY THE 1394 STACK>
    // The adapter handle on which the request will be executed.
    IN C1394_ADAPTER_HANDLE      C1394AdapterHandle;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION UPON COMPLETION OF THE COMMAND>
    // The client channel handle of the channel. This is filled by the
    // 1394 stack inside C1394IsochQueue. Any value stored by the client
    // will be overwritten upon entry to C1394IsochQueue.
    IN CLIENT_CHANNEL_HANDLE     ClientChannelHandle;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE 1394 STACK>
    // The channel handle of the channel on which the request will be executed.
    IN C1394_CHANNEL_HANDLE      C1394ChannelHandle;

    // <FILLED BY THE 1394 STACK>
    // <USED BY THE APPLICATION AND THE 1394 STACK>
    // This value contains a unique id for each request submitted to the miniport.
    // The first operation has ID zero, and the rest get the ID of the previous
    // request plus one. At some point in space & time this will wrap.
    // This ID can be only be used by applications after they retrieve
    // the structure from C1394GetNextCompleteRequest.
    OUT ULONG          uRequestIndex;

    // <USED EXCLUSIVELY BY APPLICATIONS>
    // Space for context information used by the application.
    void              *Context[4];

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // Command options that should be handled by the 1394 stack.
    // This involves both timeout and bus reset related options.
    IN ULONG          fOptions;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // The event object to be signaled when this command completes.
    // This is used only when the COMPLETE_SET_EVENT flag is set in fOptions.
    IN HANDLE         CompletionEventHandle;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // The timeout value in msec for this command.
    // Receive commands may time out because the sender stopped sending data.
    // Transmit commands may time out because the cycle master stopped sending
    // cycle starts.
    // A value of zero means that no timeout is used for this command.
    IN USHORT         ushTimeoutMsec;

    // <FILLED BY THE APPLICATION>
    // <USED BY THE 1394 STACK>
    // If the number of bus resets that occur while this command is running
    // exceeds this number, then the command is aborted.
    IN UCHAR          uchBusResetLimit;
}
```

```

// <FILLED BY THE APPLICATION>
// <USED BY THE 1394 STACK>
// If the number of cycle lost event that occur while this command is running
// exceeds this number, then the command is aborted.
IN  UCHAR   uchCycleLostLimit;

// <FILLED BY THE 1394 STACK>
// <USED BY THE APPLICATION>
// The number of bus resets that occurred while this command was running.
OUT UCHAR   uchBusResets;

// <FILLED BY THE 1394 STACK>
// <USED BY THE APPLICATION>
// The number of cycle lost events that occurred while this command was running.
OUT UCHAR   uchCyclesLost;

// A union of structures describing the operation-specific parameters
// of all supported operations.
union
{
    C1394_ISOCH_XMIT_DATA           XmitData;
    C1394_ISOCH_XMIT_PKTS          XmitPkts;
    C1394_ISOCH_XMIT_FIXED_PKTS    XmitFixedPkts;

    C1394_ISOCH_RCV_PKTS           RcvPkts;
    C1394_ISOCH_RCV_FIXED_PKTS     RcvFixedPkts;
    C1394_ISOCH_RCV_FIXED_DATA     RcvFixedData;
    C1394_ISOCH_RCV_FIXED_DATA_NH  RcvFixedDataNH;

    C1394_ISOCH_IDLE_CYCLES        IdleCycles;
};
}
FIREAPI_ISOCH_REQUEST, *PFIREAPI_ISOCH_REQUEST;

```

The *Tag* field must be filled by applications with the value **TAG_FIREAPI_ISOCH_REQUEST**. This is used by the 1394 stack as a first means of validating a pointer to a **FIREAPI_ISOCH_REQUEST** structure.

The possible values for the *uOperationCode* field are shown in the table that follows in the next page.

Value	Description
ISOCH_OP_RCV_FIXED_PKTS	<p>An isochronous receive operation, where complete isochronous packets (header quadlet + data) are received in a virtually contiguous buffer.</p> <p>A maximum size M (in quadlets) is specified for the isochronous packets to be received. The Nth packet received in the buffer is stored at offset (M+1)*4*N.</p> <p>If a packet smaller than M quadlets is received, then simply a couple of bytes stay unused. If a packet larger than M quadlets appears on the isochronous stream, then the packet will either be ignored or will be partially received depending on the operation-specific flags set for this operation. For more details see the description of the C1394_ISOCH_RCV_FIXED_PKTS structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>RcvFixedPkts</i> field.</p>
ISOCH_OP_RCV_FIXED_DATA	<p>Similar to ISOCH_OP_RCV_FIXED_PKTS, with the difference that the headers and the data are received into two separate buffers. This way the data appear <u>contiguously</u> in memory.</p> <p>A maximum size is specified for the isochronous packets to be received. For more details see the description of the C1394_ISOCH_RCV_FIXED_DATA structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>RcvFixedData</i> field.</p> <p>This method is known to present problems on some 64-bit machines with lots of memory (data corruption during DMA). It is suggested that the ISOCH_OP_RCV_FIXED_DATA_NH method is used instead.</p>
ISOCH_OP_RCV_FIXED_DATA_NH	<p>Similar to ISOCH_OP_RCV_FIXED_DATA, with the difference that there is no header buffer. The isochronous packet header (1 quadlet) is discarded and the data are received into the data buffer. The data appear <u>contiguously</u> in memory.</p> <p>A maximum size is specified for the isochronous packets to be received. For more details see the description of the C1394_ISOCH_RCV_FIXED_DATA_NH structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>RcvFixedDataNH</i> field.</p>
ISOCH_OP_XMIT_DATA	<p>An isochronous transmit operation, where the caller provides two buffers. The first is the header buffer, which can contain $H \geq 1$ quadlets for each packet. The second is the data buffer which contains the rest of the payload bytes for each packet. For more details see the description of the C1394_ISOCH_XMIT_DATA structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>XmitData</i> field.</p>
ISOCH_OP_XMIT_FIXED_PKTS	<p>An isochronous transmit operation, where complete isochronous packets (header quadlet + data) are laid out in a virtually contiguous buffer, each one starting at a multiple of a fixed offset.</p> <p>A maximum size M (in quadlets) is specified for the isochronous packets to be transmitted. The Nth packet is stored in the buffer at offset (M+1)*4*N. For more details see the description of the C1394_ISOCH_XMIT_FIXED_PKTS structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>XmitFixedPkts</i> field.</p>

<p>ISOCH_OP_XMIT_PKTS</p>	<p>An isochronous transmit operation, where complete isochronous packets (header quadlet + data) are layed out in a virtually contiguous buffer, one after the other, with each packet starting at the next quadlet boundary. For more details see the description of the C1394_ISOCH_XMIT_PKTS structure below.</p> <p>When this code is specified, then the operation-specific parameters are found in the <i>XmitPkts</i> field.</p>
---------------------------	--

Table 5. FireAPI Isochronous Operation Codes

Request Index

The *uRequestIndex* field is a per-adapter-channel unique value, assigned by the 1394 stack to each request being queued for an adapter channel.

The value assigned for a request is the value of the previous request incremented by one (wrapping back to zero after 2^{32} operations). This value is used as an extra identifier for an isochronous request, and it is used in implementing various operations that are synchronization-sensitive (like cancelling or timing out queued operations).

Note that this value is stored in the structure by the 1394 stack, and the value used for the first command of each adapter channel is zero.

Any value written in that field by an application will be overwritten by the 1394 stack.

Applications should either keep track of the number of requests that they have so far submitted to the 1394 stack, or specify **OID_CHANNEL_RQ_INDEX** in a call to **C1394QueryInformation** in order to find out the value that will be used by the class driver for the next isochronous command that will be submitted.

Request Timeouts

Timing-out is requested for an isochronous operation if the *ushTimeoutMsec* field is non zero. This field is expressed in milliseconds. This value identifies an upper limit in the amount of time that it should take for the command to execute. If this interval elapses and the command is not complete, then the 1394 stack completes this command with the status **STATUS_1394_TIMEOUT**.

There are a number of reasons why a command could time out:

- The cycle master hungs and stops generating cycle start packets. This will cause all isochronous talkers to stop transmitting.
- An isochronous talker hungs and stops generating isochronous traffic.
- An isochronous talker was physically removed from the bus.

If timing is requested for an isochronous request, then the related flags in the *fOptions* field are shown in the table below. These flags are mutually exclusive (only one of them can be specified). The default timeout option is **TIMEOUT_START_NEXT**⁵⁴.

If *ushTimeoutMsec* is zero then the related bits are cleared by the 1394 stack.

⁵⁴ If *ushTimeoutMsec* is non-zero and no **TIMEOUT_XXX** flag is specified, then the 1394 stack will OR the **TIMEOUT_START_NEXT** flag in the *fOptions* field of the request.

Value	Description
TIMEOUT_START_NEXT	Cancel the current request with status STATUS_1394_TIMEOUT and immediately continue with the next one.
TIMEOUT_START_NEXT_NO_TIMING	Cancel the current request with status STATUS_1394_TIMEOUT and immediately continue with the next one, but without timing it out (assume <i>ushTimeoutMsec</i> is zero). Request timing will resume after the next request is completed.
TIMEOUT_FLUSH_QUEUE	Cancel all remaining requests. If any new requests get submitted for this channel, then time them accordingly to their <i>TimeoutOptions</i> .

Table 6. Isochronous Request Timeout Options

Bus Reset Handling

Another issue, also related to timing, that has to be faced by the 1394 stack is what happens to an isochronous request when a bus reset occurs. A bus reset causes a temporary disruption in isochronous traffic, which may in some cases be longer than anticipated, and cause an isochronous operation to timeout. Similarly it could be desirable for an application to cancel or restart an isochronous command if a bus reset occurs while it is executing.

The options that can be specified in the *fOptions* field with regards to bus reset handling are shown in the table below. These flags are mutually exclusive (only one of them can be specified). The default option is **BR_START_NEXT**⁵⁵.

Value	Description
BR_FLUSH_QUEUE	Cancel all requests in the queue, with status STATUS_1394_BUS_RESET .
BR_START_NEXT	Cancel the current request with status STATUS_1394_BUS_RESET and start the next (if there is one) as soon as the bus reset completes. This is the default option if no BR_XXX flag is specified.
BR_RESTART	Restart the current operation anew, after the bus reset completes, possibly overwriting any data received so far. The value of the <i>uchBusResets</i> field will be incremented by one, regardless of whether the BUS_RESET_LIMIT flag is specified or not. If a non-zero value is specified in <i>ushTimeoutMsec</i> then timing is restarted as well.
BR_CONTINUE	Do nothing. Continue with the current request and continue timing as if nothing happened (only applicable if a non-zero value is specified in <i>ushTimeoutMsec</i>). The value of the <i>uchBusResets</i> field will be incremented by one, regardless of whether the BUS_RESET_LIMIT flag is specified or not.
BR_RESTART_TIMING	Continue with the current request but restart the timing of the operation anew, as soon as bus reset completes (only applicable if a non-zero value is specified in <i>ushTimeoutMsec</i>). The value of the <i>uchBusResets</i> field must be incremented by one, regardless of whether the BUS_RESET_LIMIT flag is specified or not.

Table 7. Isochronous Request Bus Reset Options

⁵⁵ If no BR_XXX flag is specified, then the 1394 stack will OR the BR_START_NEXT flag in the *fOptions* field of the request.

Other Options

The following flags can also be specified in the *fOptions* field of an isochronous request:

Value	Description
COMPLETE_SET_EVENT	Set the associated event object when this command is completed.
BUS_RESET_LIMIT	If the number of bus resets that occurred during the processing of this command exceeds the value specified by the field <i>uchBusResetLimit</i> then abort the command with status STATUS_1394_BR_LIMIT . This flag can not be used in combination with BR_FLUSH_QUEUE or BR_START_NEXT .
CYCLE_LOST_LIMIT ⁵⁶	If the number of cycle lost interrupts that occurred during the processing of this command exceeds the value specified by the field <i>uchCycleLostLimit</i> then abort the command with status STATUS_1394_CYCLOST_LIMIT .

⁵⁶ This flag will be implemented in future versions of FireAPI.

Isochronous Completion Status

The *Status* field reflects the completion status of the request. When the request is submitted with **C1394IsochQueue** this field is set to **STATUS_1394_PENDING**. Upon return it is set as follows:

Value	Description
STATUS_1394_SUCCESS	The requested operation was completed successfully, without any errors. <u>This status code will only be returned when there was no error during the execution of the request.</u>
STATUS_1394_ABORTED	The operation was aborted prematurely due to a class driver request.
STATUS_1394_BUS_RESET	The operation was aborted because a bus reset occurred and either the request had specified the flag BR_START_NEXT in the <i>fOptions</i> field, or a previous request had the BR_FLUSH_QUEUE flag set.
STATUS_1394_BR_LIMIT	The operation was aborted because the number of bus resets exceeded the number specified by <i>uchBusResetLimit</i> .
STATUS_1394_DMA_LIMIT	The operation was aborted by the 1394 stack because it exceeds the maximum size of DMA operations as set by the operating system.
STATUS_1394_TIMEOUT	The operation was completed due to a timeout.
STATUS_1394_DUPLICATE_CHANNEL	The channel number specified in the operation-specific parameters is already used by another adapter channel for the same type of activity (transmit or receive).
STATUS_1394_NO_MEMORY	The operation could not be completed because a memory allocation failed.
STATUS_1394_CRC_ERROR	The reception of one or more packets failed with a CRC error.
STATUS_1394_FIFO_OVERRUN	The DMA channel failed to receive one or more packets because of a FIFO overrun. This situation may occur on very busy systems, when the adapter did not have the chance to move data from the isochronous receive FIFO into main memory fast enough. As a result the receive FIFO was completely filled and incoming data was lost. Note that this operation is done automatically by the 1394 adapter on the PCI so it has little to do with interrupt latency and interrupt handling. The most likely cause of this error is a PCI bus that is too loaded with traffic. Note that OHCI adapters cannot distinguish between a FIFO overrun error (which means that the <i>data_length</i> in the header of the received packet does not match the received packet size) and a data CRC error. This means that if there is an actual data CRC error (usually one or more corrupted iso packets on the bus) you will still get the same error code. If you are troubleshooting this type of error you are advised to use a 1394 Bus Analyzer and check whether the iso stream is valid throughout or it contains erroneous packets.

STATUS_1394_FIFO_UNDERRUN	<p>The DMA channel failed to transmit one or more packets because of a FIFO underrun.</p> <p>This situation occurs usually on very busy systems, when the adapter did not have the chance to move data from main memory to the isochronous transmit FIFO fast enough. As a result the transmit FIFO was emptied and an invalid packet was transmitted.</p> <p>Note that this operation is done automatically by the 1394 adapter on the PCI so it has little to do with interrupt latency and interrupt handling. The most likely cause of this error is a PCI bus that is too loaded with traffic.</p>
STATUS_1394_DMA_ERROR + xxx	<p>The operation was aborted because an error other than the above occurred in the isochronous DMA channel.</p> <p>The actual error code returned is the STATUS_1394_DMA_ERROR constant plus the OHCI-level event code for the operation. Some event codes have been remapped to specific STATUS_1394 errors as follows:</p> <p>evt_long_packet (0x02) → STATUS_1394_LONG_PACKET</p> <p>evt_overrun (0x05) → STATUS_1394_FIFO_OVERRUN</p> <p>evt_underrun (0x04) → STATUS_1394_FIFO_UNDERRUN</p> <p>ack_data_error (0x1D) → STATUS_1394_CRC_ERROR</p> <p>Note that the ack_data_error condition is often reported instead of evt_overrun as the adapter runs out of isochronous receive FIFO and truncates the incoming packet so the packet appears to have wrong length and invalid CRC.</p>
STATUS_1394_LONG_PACKET	<p>At least one isochronous data packet was received whose data length (payload) was larger than the maximum payload of the isochronous request.</p>
STATUS_1394_UNSUCCESSFUL	<p>An error other than the above occurred.</p>

Whenever an operation is prematurely aborted, then depending on the type of operation some information will be available on how far did the operation proceed (how many packets were sent or received before the request was cancelled).

Isynchronous Operation Parameters

This section describes operation-specific parameters.

C1394_ISOCH_RCV_FIXED_PKTS

Used with the **ISOCH_OP_RCV_FIXED_PKTS** operation code.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_RCV_FIXED_PKTS.
    IN ULONG      Tag;

    // The memory into which to receive the packets.
    IN void       *PacketBuffer;

    // The size of the packet reception buffer.
    // When calculating the buffer size, remember to account one quadlet
    // per packet (the isochronous packet header).
    IN ULONG      uBufferBytes;

    // The maximum *payload* size in quadlets of the packets to be received.
    IN USHORT     ushMaxPayloadQuads;

    // Operation Flags (byte swap, use SyCode, use TagCode, bigger ignore/partialRcv).
    IN USHORT     Flags;

    // The SyCode with which to synchronize the start of reception.
    IN UCHAR      IsochSyCode;

    // The TagCode with which to synchronize the start of reception.
    IN UCHAR      IsochTagCode;

    // The channel number to receive from.
    IN UCHAR      ChannelNumber;

    // The number of packets received into the buffer.
    // This is considered valid even if a receive operation was aborted
    // due to a timeout, or some other reason.
    OUT ULONG     uPacketsReceived;
}
C1394_ISOCH_RCV_FIXED_PKTS, *PC1394_ISOCH_RCV_FIXED_PKTS;
```

The operation-specific flags that can be specified in the *Flags* field are:

Value	Description
RCV_START_ON_SYCODE	The value in the <i>IsochSyCode</i> field must be matched in the first isochronous packet to be received in the buffer.
RCV_FORCE_SYCODE	The value in the <i>IsochSyCode</i> field must be matched on all isochronous packets that are received in the buffer.
RCV_START_ON_TAGCODE	The value in the <i>IsochTagCode</i> field must be matched in the first isochronous packet to be received.
RCV_FORCE_TAGCODE	The value in the <i>IsochTagCode</i> field must be matched on all isochronous packets that are received in the buffer.
RCV_BYTE_SWAP	Byte-swap the packet data on a quadlet basis, while the packet is being received. This byte swap is performed by the DMA channel itself, not software. For efficiency reasons this byte swap operation also byte swaps the header of the isochronous packet (1 quadlet), so the applications should be prepared to do their header checks as appropriate in this case.

The **RCV_START_ON_SYCODE** and **RCV_FORCE_SYCODE** flags are mutually exclusive. Only one of them can be specified. Similarly, the **RCV_START_ON_TAGCODE** and **RCV_FORCE_TAGCODE** flags are mutually exclusive. Only one of them can be specified.

The combination of these flags impose criteria that define the exact point in the isochronous stream from where the command will start receiving data. The *START_ON* flags impose criteria only on the first isochronous packet, while the *FORCE* flags impose criteria on all the isochronous packets. Using the *FORCE* flag(s) one can effectively achieve selective reception of isochronous packets.

Using the ***RCV_FORCE_XXX*** flags imposes non-trivial processing overhead at the hardware level, because the 1394 chip has to execute a more complex set of instructions for each incoming isochronous packet.

This means that even if they are used with an isochronous stream with relatively small bandwidth, they may tie up the 1394 chip to a level where it cannot operate a second isochronous DMA channel at the same time.

If any isochronous packets larger than `ushMaxPayloadQuads*4` appear on the stream, then they will 'appear' partially received to the application. The packet will be received completely, but any data bytes found after the maximum size will be overwritten by the next isochronous packet that will be received.

Upon completion of the command, no matter in what way was the command completed, the 1394 stack provides valid information in the *uPacketsReceived* field.

C1394_ISOCH_RCV_FIXED_DATAUsed with the **ISOCH_OP_RCV_FIXED_DATA** operation code.

```

typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_RCV_FIXED_DATA.
    IN  ULONG      Tag;

    // The memory into which to receive the packet payload.
    // The required size of this buffer is equal to
    // (ushMaxPayloadQuads*4)*uPacketsToReceive.
    IN  void       *DataBuffer;

    // The memory into which to receive the packet headers.
    // ushHeaderQuads quadlets are being received as the header of each packet.
    // The required size of this buffer is equal to
    // (ushHeaderQuads*4)*uPacketsToReceive.
    IN  void       *HeaderBuffer;

    // The number of packets to receive.
    IN  USHORT     ushPacketsToReceive;

    // The number of header quadlets to move to the header buffer.
    // This is set to 1 if only the isochronous packet header is to be stripped
    // from each isochronous packet.
    // However additional quadlets can be copied if required.
    IN  USHORT     ushHeaderQuads;

    // The maximum *payload* size in quadlets of the packets to be received.
    // This is the number of 'pure' payload quadlets (the payload that remains
    // after all header quadlets are removed).
    // This means that the total max payload of any receive packet is equal to
    // (ushMaxPayloadQuads + ushHeaderQuads - 1)*4
    IN  USHORT     ushMaxPayloadQuads;

    // Operation Flags (byte swap, use SyCode, use TagCode, bigger ignore/partialRcv).
    IN  USHORT     Flags;

    // The SyCode with which to synchronize the start of reception.
    IN  UCHAR      IsochSyCode;

    // The TagCode with which to synchronize the start of reception.
    IN  UCHAR      IsochTagCode;

    // The channel number to receive from.
    IN  UCHAR      ChannelNumber;

    // The number of packets received into the buffer.
    // This is considered valid even if a receive operation was aborted
    // due to a timeout, or some other reason.
    OUT ULONG      uPacketsReceived;
}
C1394_ISOCH_RCV_FIXED_DATA, *PC1394_ISOCH_RCV_FIXED_DATA;

```

The flags that can be used and the related restrictions are the same as for **C1394_ISOCH_RCV_FIXED_PKTS**.

The maximum size of the isochronous packets to be received is the sum of:

1. The 'header' quadlets $H \geq 1$ (includes the isoch packet header quadlet and possibly one or more 'protocol' header quadlets). The value of H is stored in the *ushHeaderQuads* field.
2. The 'pure' payload quadlets $M \geq 1$. The value of M is stored in the *ushMaxPayloadQuads* field.

The actual maximum payload of an isochronous packet (its *data_length*) is $(M+H-1)$ quadlets.

The payload of the N^{th} packet received is stored at byte offset $(M*4) * N$ in the data buffer, and its header quadlet is store at byte offset $(H*4) * N$ in the header buffer.

If a packet with a payload larger than $(M+H-1)$ quadlets appears on the isochronous stream, then the packet will either be ignored or will be partially received depending on the operation-specific flags set for this operation.

If a packet smaller than $(M+H-1)$ quadlets but larger than $(H-1)$ quadlets is received, then simply a couple of bytes stay unused in the data buffer.

If a packet smaller than $(H-1)$ quadlets is received, then a couple of bytes will stay unused in the header buffer, and M quadlets will stay unused in the data buffer.

C1394_ISOCH_RCV_FIXED_DATA_NH

Used with the **ISOCH_OP_RCV_FIXED_DATA_NH** operation code.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_RCV_FIXED_DATA_NH.
    IN  ULONG      Tag;

    // The memory into which to receive the packet payload.
    // The required size of this buffer is equal to
    // (ushMaxPayloadQuads*4)*uPacketsToReceive.
    IN  void      *DataBuffer;

    // The number of packets to receive.
    IN  USHORT     ushPacketsToReceive;

    // The maximum *payload* size in quadlets of the packets to be received.
    IN  USHORT     ushMaxPayloadQuads;

    // Operation Flags (byte swap, use SyCode, use TagCode, bigger ignore/partialRcv).
    IN  USHORT     Flags;

    // The SyCode with which to synchronize the start of reception.
    IN  UCHAR      IsochSyCode;

    // The TagCode with which to synchronize the start of reception.
    IN  UCHAR      IsochTagCode;

    // The channel number to receive from.
    IN  UCHAR      ChannelNumber;

    // The number of packets received into the buffer.
    // This is considered valid even if a receive operation was aborted
    // due to a timeout, or some other reason.
    OUT ULONG      uPacketsReceived;
}
C1394_ISOCH_RCV_FIXED_DATA_NH, *PC1394_ISOCH_RCV_FIXED_DATA_NH;
```

The flags that can be used and the related restrictions are the same as for **C1394_ISOCH_RCV_FIXED_DATA**.

The payload of the Nth packet received is stored at byte offset $(ushMaxPayloadQuads * 4) * N$ in the data buffer.

If a packet with a payload larger than *ushMaxPayloadQuads* quadlets appears on the isochronous stream, then the packet will either be ignored or will be partially received depending on the operation-specific flags set for this operation.

If a packet smaller than *ushMaxPayloadQuads* quadlets is received, then simply a couple of bytes stay unused in the data buffer.

C1394_ISOCH_XMIT_PKTS

Used with the **ISOCH_OP_XMIT_PKTS** operation code. This operation code is used when the client wants to transmit one or more isochronous streams, that have a variable block size.

When the request involves more than one isochronous streams, the number of isochronous stream packets that will be transmitted per isochronous cycle is adjustable by the caller.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_XMIT_PKTS.
    IN ULONG Tag;

    // The memory (system address space) that contains the packets.
    IN void *PacketBuffer;

    // The size of the packet buffer.
    IN ULONG uBufferBytes;

    // The number of packets in the buffer.
    IN ULONG uPacketsInBuffer;

    // Operation Flags (byte swap, cycletimer insert, packets per cycle).
    IN USHORT Flags;

    // The transmission speed.
    IN C1394_SPEED_CODE TransmissionSpeed;

    // The number of packets per cycle.
    // This is only used if the XMIT_PACKETS_PER_CYCLE flag is specified
    // in FIREAPI_ISOCH_REQUEST.Flags
    IN USHORT PacketsPerCycle;

    // The quadlet offset where to insert the cycle timer value.
    // This is only used if the XMIT_INSERT_CYCLE_TIME flag is specified
    // in FIREAPI_ISOCH_REQUEST.Flags
    IN USHORT CycleTimerInsertOffset;

    // The number of isochronous packets that were transmitted.
    OUT ULONG uPacketsTransmitted;
}
C1394_ISOCH_XMIT_PKTS, *PC1394_ISOCH_XMIT_PKTS;
```

The operation-specific flags that can be specified in the *Flags* field are:

Value	Description
XMIT_BYTE_SWAP	Byte-swap the payload data on a quadlet basis, while the packet is being transmitted. This byte swap is performed by the DMA channel itself, not software. When this option is specified, the packet header will also be byte-swapped upon transmission, so the class driver expects the packet header in little endian format.
XMIT_PACKETS_PER_CYCLE	Indicates that the 1394 stack should use the value in <i>PacketsPerCycle</i> in order to decide how many packets per cycle should be transmitted. See the comments below for a detailed explanation of the use of this field.
XMIT_LOOP	Indicates that the transmit request should be executed repeatedly, until it is interrupted by some event (C1394IsochCancel, one or more bus resets or a timeout if the request flags specify these options).
XMIT_INSERT_CYCLE_TIME	Inserts the 32-bit value of the cycle timer in the packet before transmitting it, at the quadlet offset specified by <i>CycleTimerInsertOffset</i> . This is only available if supported by the adapter.

The 1394 stack expects to find in *PacketBuffer* a series of complete isochronous packets, which means 1 quadlet for the header followed by the data for each packet. Each packet starts at the next quadlet boundary after the previous one.

This means that if the current packet starts at offset *Base* and contains *data_length* bytes, then the next isochronous packet is expected at offset $(Base + ((data_length+3) \& 0xFFFC))$.

If an isochronous packet is zero bytes long, then the next isochronous packet starts on the next quadlet.

By default the 1394 stack does not byte-swap the data upon transmission. This means that the data are transmitted as they appear in memory, which in turn means that the header of each packet must be properly prepared in big endian format.

If the **XMIT_BYTE_SWAP** flag is specified in the *XmitPkts.Flags* field, then both the data and the header will be transmitted byte-swapped. This means that in this case the isochronous packet header must appear in little endian format.

The *PacketsPerCycle* field is used when the **XMIT_PACKETS_PER_CYCLE** flag is specified in the *XmitPkts.Flags* field. It is used to control the number of packets that will be transmitted per isochronous cycle when more than one isochronous streams are involved in an isochronous transmit request.

If the request only contains packets for a single isochronous stream then this flag is ignored, and should not be specified. It is a 1394 requirement that an isochronous stream number can only appear once per cycle.

If the flag is not specified and multiple streams are found in the packet buffer, then the 1394 stack will transmit one isochronous packet per cycle.

Otherwise the 1394 stack will use the value of *PacketsPerCycle* as the basic directive in order to decide how many packets it will transmit per cycle. The number of packets that will actually be transmitted will always be limited by the 1394 standard requirement that each channel (stream) number can only appear once per isochronous cycle.

For example, if the channel numbers in successive packets are 1-2-1-2-1-1-1-2-1-2-1-2, and *PacketsPerCycle* is 2, then the packets will be transmitted as [1-2]-[1-2]-[1]-[1]-[1-2]-[1-2]-[1-2].

If *PacketsPerCycle* is zero, then the class driver will transmit as many isochronous packets per cycle as possible, always subject to the rule that each stream number is transmitted once per cycle.

Upon completion of the command, no matter in what way was the command completed, the class driver provides valid information in the *uPacketsTransmitted* field.

IMPORTANT NOTE: The data payload portion of an isochronous packet is always constructed as a quadlet multiple. For example, if *data_length* is 41, the data payload that must be actually transmitted will be 44 bytes. The last 3 bytes are called *padding*. They are meaningless and are usually set to zero. The application must take this into consideration when preparing the packets for transmission.

C1394_ISOCH_XMIT_FIXED_PKTS

Used with the **ISOCH_OP_XMIT_FIXED_PKTS** operation code.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_XMIT_FIXED_PKTS.
    IN ULONG    Tag;

    // The memory that contains the packets.
    IN void    *PacketBuffer;

    // The size of the packet buffer.
    IN ULONG    uBufferBytes;

    // The number of packets in the buffer.
    IN ULONG    uPacketsInBuffer;

    // The fixed *payload* size in quadlets of the packets to be transmitted.
    // This applies for ALL isoch packets to be transmitted.
    IN USHORT    ushPayloadQuads;

    // Operation Flags (byte swap, cycletimer insert).
    IN USHORT    Flags;

    // The transmission speed.
    IN C1394_SPEED_CODE    TransmissionSpeed;

    // The quadlet offset where to insert the cycle timer value.
    IN USHORT    CycleTimerInsertOffset;

    // The number of isochronous packets that were transmitted.
    OUT ULONG    uPacketsTransmitted;
}
C1394_ISOCH_XMIT_FIXED_PKTS, *PC1394_ISOCH_XMIT_FIXED_PKTS;
```

The operation-specific flags that can be specified in the *Flags* field are:

Value	Description
XMIT_BYTE_SWAP	Byte-swap the packet data on a quadlet basis, while the packet is being transmitted. This byte swap is performed by the DMA channel itself, not software. When this option is specified, the packet header will also be byte-swapped upon transmission, so the class driver expects the packet header in little endian format.
XMIT_LOOP	Indicates that the transmit request should be executed repeatedly, until it is interrupted by some event (C1394IsochCancel, one or more bus resets or a timeout if the request flags specify these options).
XMIT_INSERT_CYCLE_TIME	Inserts the 32-bit value of the cycle timer in the packet before transmitting it, at the quadlet offset specified by <i>CycleTimerInsertOffset</i> . This is only available if supported by the adapter.

The buffer must obviously be big enough to hold all the packets. This means that:

$$uBufferBytes \geq uPacketsInBuffer * (ushPayloadQuads + 1) * 4$$

The class driver expects to find in *PacketBuffer* a series of complete isochronous packets, which means 1 quadlet for the header followed by a maximum of *ushPayloadQuads* for the data. The isochronous payload of any isochronous packet can be less than *ushPayloadQuads*4* if desired. It can even be zero, but still a header with a zero *data_length* field must appear.

However in the **ISOCH_OP_XMIT_FIXED_PKTS** operation code, the class driver demands that no matter the size of each individual isochronous packet, the Nth packet (N=0..uPacketsInBuffer-1) is located at offset $N * (ushPayloadQuads + 1) * 4$ from the start of the packet buffer. If some isochronous packets have a smaller payload, then the memory bytes from the end of that packet to the beginning of the next are being wasted.

Upon completion of the command, no matter in what way was the command completed, the class driver provides valid information in the *uPacketsTransmitted* field.

By default the drivers does not byte-swap the data upon transmission. This means that the data are transmitted as they appear in memory, which in turn means that the header of each packet must be properly prepared in big endian format.

If the **XMIT_BYTE_SWAP** flag is specified in the *XmitFixedPkts.Flags* field, then both the data and the header will be transmitted byte-swapped. This means that in this case the isochronous packet header must appear in little endian format.

Upon completion of the command, no matter in what way was the command completed, the class driver provides valid information in the *uPacketsTransmitted* field.

IMPORTANT NOTE: The data payload portion of an isochronous packet is always constructed as a quadlet multiple. For example, if *data_length* is 41, the data payload that must be actually transmitted will be 44 bytes. The last 3 bytes are called *padding*. They are meaningless and are usually set to zero. The application must take this into consideration when preparing the packets for transmission.

C1394_ISOCH_XMIT_DATA

Used with the **ISOCH_OP_XMIT_DATA** operation code.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_XMIT_DATA.
    IN ULONG    Tag;

    // The memory that contains the data that
    // will be used as the packets' payload.
    IN void     *DataBuffer;

    // The size of the data buffer.
    // This size should be greater or equal to the sum of the data_length
    // field of the packet headers found in HeaderBuffer, minus the total
    // size of the extra header quadlets (ushHeaderQuads-1)*uPacketsInBufer*4.
    IN ULONG    uBufferBytes;

    // The buffer that contains the headers to be used.
    // The required size of this buffer is equal to
    // (ushHeaderQuads-1)*uPacketsInBufer*4.
    IN void     *HeaderBuffer;

    // The number of packets in the buffer.
    IN ULONG    uPacketsInBuffer;

    // The number of header quadlets contained in the header buffer for each packet.
    // This is set to 1 if the header buffer only contains the isochronous
    // packet header for each isochronous packet.
    IN USHORT   ushHeaderQuads;

    // Operation Flags (byte swap, xmit loop, cycletimer insert, packets per cycle).
    IN USHORT   Flags;

    // The transmission speed.
    IN C1394_SPEED_CODE    TransmissionSpeed;

    // The number of packets per cycle.
    // This is only used if the XMIT_PACKETS_PER_CYCLE flag is specified
    // in C1394_ISOCH_XMIT_DATA.Flags
    IN USHORT   PacketsPerCycle;

    // The quadlet offset where to insert the cycle timer value.
    // This is only used if the XMIT_INSERT_CYCLE_TIME flag is specified
    // in C1394_ISOCH_XMIT_PKTS.Flags
    IN USHORT   CycleTimerInsertOffset;

    // The number of isochronous packets that were transmitted.
    OUT ULONG   uPacketsTransmitted;
}
C1394_ISOCH_XMIT_DATA, *PC1394_ISOCH_XMIT_DATA;
```

The flags that can be used and the related restrictions and comments are the same as for **C1394_ISOCH_XMIT_FIXED_PKTS**. For comments regarding the use of **XMIT_BYTE_SWAP** and **XMIT_PACKETS_PER_CYCLE** see that section.

The header quadlets for each packet contain the isochronous packet header quadlet, and optionally one or more quadlets of 'protocol' data. The header quadlets of the N^{th} isochronous packet are located at offset $(\text{ushHeaderQuads} * 4) * N$ in the header buffer.

The number of bytes that will be read from the data buffer and will be added to the payload of each packet is $\text{data_length} - (H - 1) * 4$, provided of course that $(H - 1) * 4 < \text{data_length}$. The value of *data_length* is found in the isochronous packet header quadlet.

It can be the case that the *data_length* of a packet is smaller than $(\text{ushHeaderQuads} - 1) * 4$. In this case some bytes are unused in the header buffer, and no bytes are being used for this packet from the data buffer.

IMPORTANT NOTE: The data payload portion of an isochronous packet is always constructed as a quadlet multiple. For example, if *data_length* is 41, the data payload that will be actually transmitted will be 44 bytes. The last 3 bytes are called *padding*. They are meaningless and are usually set to zero. With FireAPI, if the *data_length* of a packet is not a multiple of 4, then the 1394 stack will not ensure a zero-fill for the padding of the data payload. The fill will actually be the first bytes of the payload of the next isochronous packet.

This only affects the padding bytes that will be added. The next isochronous packet will start from the correct position in the buffer.

C1394_ISOCH_IDLE_CYCLES

Used with the **ISOCH_OP_IDLE_CYCLES** operation code.

```
typedef struct
{
    // Identification tag. Must be set to TAG_ISOCH_IDLE_CYCLES.
    IN ULONG    Tag;

    // The number of cycles to leave idle before completing this command.
    IN ULONG    uIdleCycles;

    // Operation specific flags.
    IN ULONG    Flags;
}
C1394_ISOCH_IDLE_CYCLES, *PC1394_ISOCH_IDLE_CYCLES;
```

The operation-specific flags that can be specified in the *Flags* field are:

Value	Description
IDLE_CYCLE_LOOP	This specifies that the <i>uIdleCycles</i> field should be ignored and that the adapter channel should stay idle until this command is cancelled or aborted (for example due to a bus reset). This way the channel can be 'triggered' according to an external event.

Isynchronous Packet Header Structures & Macros

The `C1394_STREAM_PACKET_HEADER` structure and the `MAKE_ISOCH_HEADER` macro have been defined for the purpose of helping construct isochronous packet headers.

C1394_STREAM_PACKET_HEADER

```
typedef union
{
    // An alias for accessing each byte separately.
    UCHAR    Bytes[4];

    // An alias for accessing it as a quadlet.
    // ENDIANESS DEPENDENT.
    ULONG    Quadlet;

    struct
    {
        // The data_length field
        // ENDIANESS DEPENDENT
        USHORT data_length;

        struct
        {
            // The channel number.
            UCHAR    Channel:6;

            // The tag of this packet.
            UCHAR    Tag:2;
        };

        struct
        {
            // The sy code.
            UCHAR    SyCode:4;

            // The transaction code.
            UCHAR    tcode:4;
        };
    };
}
C1394_STREAM_PACKET_HEADER, *PC1394_STREAM_PACKET_HEADER;
```

MAKE_ISOCH_HEADER

The `MAKE_ISOCH_HEADER` macro, constructs a valid big endian header, given a pointer to a `C1394_STREAM_PACKET_HEADER` structure and the values to store in.

The macro is prototyped as shown below:

```
void MAKE_ISOCH_HEADER(
    PC1394_STREAM_PACKET_HEADER    pStreamPacketHeader,
    USHORT                          data_length,
    C1394_CHANNEL                   ChannelNumber,
    C1394_TAG                        TagCode,
    C1394_SY_CODE                   SyCode
);
```

Additionally the macro `MAKE_ULONG_ISOCH_HEADER` has been defined in order to construct a native `ULONG` value structured like an isochronous packet header (16 bits `data_length`, 2 bits `Tag`, 6 bits `channel number`, 4 bites `transaction code`, 4 bits `sy-code`).

```
ULONG MAKE_ULONG_ISOCH_HEADER(
    USHORT    data_length,
    C1394_CHANNEL    ChannelNumber,
    C1394_TAG    TagCode,
    C1394_SY_CODE    SyCode
);
```

The resulting header quadlet is in the CPU's native endianness.

PHY Packet Structures & Macros

The following structures have been defined for use with PHY packets. The types defined include all the extended PHY packet types defined by P1394A.

All packet types are defined in big endian format. A PHY packet constructed with the use of one of these structures can be directly transmitted to the 1394 bus using **C1394TransmitRaw**.

C1394_PHY_PACKET_GENERIC

This type is used for testing what the actual PHY packet is.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;
        };

        // Valid only if PHTType == PHY_TYPE_ZERO
        struct
        {
            UCHAR:6;
            UCHAR T:1;
            UCHAR R:1;
        };

        UCHAR Bytes_3_4[2];
    };
}
C1394_PHY_PACKET_GENERIC, *PC1394_PHY_PACKET_GENERIC;
```

C1394_PHY_PACKET_SELF_ID_0

This structure describes a Self-ID #0 packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2; // Should Be Binary 10 (PHY_TYPE_SELF_ID).
        };

        // The second byte (Base address + 1).
        struct
        {
            UCHAR gap_count:6;
            UCHAR L:1;
            UCHAR Zero:1; // This is not reserved. It is defined and it must be zero.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR pwr:3;
            UCHAR c:1; // Contender bit.
            UCHAR rsv:2; // Reserved. Must be zero.
            UCHAR sp:2; // Speed code.
        };
    };
};
```

```

    // The fourth byte (Base address + 3).
    struct
    {
        UCHAR m:1;        // More packets.
        UCHAR i:1;        // Initiated bus reset.
        UCHAR p2:2;
        UCHAR p1:2;
        UCHAR p0:2;
    };
};
}
C1394_PHY_PACKET_SELF_ID_0, *PC1394_PHY_PACKET_SELF_ID_0;

```

C1394_PHY_PACKET_SELF_ID_N

This structure describes generically Self-ID packets #1,2,3.

```

typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;        // Should Be Binary 10 (PHY_TYPE_SELF_ID).
        };

        // The second byte (Base address + 1).
        struct
        {
            UCHAR pa:2;
            UCHAR rsv:2;        // Reserved. Must be 0.
            UCHAR n:3;        // Must be 0,1 or 2.
            UCHAR One:1;        // This is not reserved. It is defined and it must be 1.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR pe:2;
            UCHAR pd:2;
            UCHAR pc:2;
            UCHAR pb:2;
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR m:1;        // More packets.
            UCHAR r:1;        // Reserved. Must be 0.
            UCHAR ph:2;
            UCHAR pg:2;
            UCHAR pf:2;
        };
    };
};
}
C1394_PHY_PACKET_SELF_ID_N, *PC1394_PHY_PACKET_SELF_ID_N;

```

C1394_PHY_PACKET_SELF_ID_1

This structure describes a Self-ID #1 packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;          // Should Be Binary 10 (PHY_TYPE_SELF_ID).
        };

        // The second byte (Base address + 1).
        struct
        {
            UCHAR p3:2;
            UCHAR rsv:2;          // Reserved. Must be 0.
            UCHAR n:3;          // Must be 0.
            UCHAR One:1;        // This is not reserved. It is defined and it must be 1.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR p7:2;
            UCHAR p6:2;
            UCHAR p5:2;
            UCHAR p4:2;
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR m:1;          // More packets.
            UCHAR r:1;          // Reserved. Must be 0.
            UCHAR p10:2;
            UCHAR p9:2;
            UCHAR p8:2;
        };
    };
};

C1394_PHY_PACKET_SELF_ID_1, *PC1394_PHY_PACKET_SELF_ID_1;
```


C1394_PHY_PACKET_SELF_ID_2

This structure describes a Self-ID #2 packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;          // Should Be Binary 10 (PHY_TYPE_SELF_ID).
        };

        // The second byte (Base address + 1).
        struct
        {
            UCHAR p11:2;
            UCHAR rsv:2;          // Reserved. Must be 0.
            UCHAR n:3;           // Must be 1.
            UCHAR One:1;         // This is not reserved. It is defined and it must be 1.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR p15:2;
            UCHAR p14:2;
            UCHAR p13:2;
            UCHAR p12:2;
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR m:1;          // More packets.
            UCHAR r:1;          // Reserved. Must be 0.
            UCHAR p18:2;
            UCHAR p17:2;
            UCHAR p16:2;
        };
    };
};
C1394_PHY_PACKET_SELF_ID_2, *PC1394_PHY_PACKET_SELF_ID_2;
```

C1394_PHY_PACKET_SELF_ID_3

This structure describes a Self-ID #3 packet.

```

typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;          // Should Be Binary 10 (PHY_TYPE_SELF_ID).
        };

        // The second byte (Base address + 1).
        struct
        {
            UCHAR p19:2;
            UCHAR rsv:2;
            UCHAR n:3;          // Must be 2.
            UCHAR One:1;       // This is not reserved. It is defined and it must be 1.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR p23:2;
            UCHAR p22:2;
            UCHAR p21:2;
            UCHAR p20:2;
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR m:1;          // More packets. Must be zero.
            UCHAR r:1;          // Reserved. Must be 0.
            UCHAR p26:2;
            UCHAR p25:2;
            UCHAR p24:2;
        };
    };
};
C1394_PHY_PACKET_SELF_ID_3, *PC1394_PHY_PACKET_SELF_ID_3;

```

C1394_PHY_PACKET_LINK_ON

This structure describes a PHY Link-ON packet.

```

typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR PhysicalID:6;
            UCHAR PHYType:2;          // Should Be Binary 01 (PHY_TYPE_LINK_ON).
        };

        UCHAR Zeroes[3];
    };
};
C1394_PHY_PACKET_LINK_ON, *PC1394_PHY_PACKET_LINK_ON;

```

C1394_PHY_PACKET_CONFIGURATION

This structure describes a PHY Configuration packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;           // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR gap_count:6;
            UCHAR T:1;
            UCHAR R:1;
        };

        // The third and fourth bytes. SHOULD BE ZEROS.
        UCHAR Bytes_3_4[2];
    };
}
C1394_PHY_PACKET_CONFIGURATION, *PC1394_PHY_PACKET_CONFIGURATION;
```

C1394_PHY_PACKET_EXTENDED

This structure generically describes an extended PHY packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;           // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR PacketSpecificBits:2;           // Packet Specific.
            UCHAR type:4;                          // The packet type.
            UCHAR T:1;                             // Should be zero.
            UCHAR R:1;                             // Should be zero.
        };

        UCHAR PacketSpecificBytes[2];
    };
}
C1394_PHY_PACKET_EXTENDED, *PC1394_PHY_PACKET_EXTENDED;
```

C1394_PHY_PACKET_PING

This structure generically describes a PHY Ping packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;          // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR:2;                // Should be zero.
            UCHAR type:4;           // Should be PHY_EXTENDED_PING (0) for PHY Ping Packets.
            UCHAR T:1;              // Should be zero.
            UCHAR R:1;              // Should be zero.
        };

        // The third and fourth bytes. SHOULD BE ZEROS.
        UCHAR Bytes_3_4[2];
    };
}
C1394_PHY_PACKET_PING, *PC1394_PHY_PACKET_PING;
```

C1394_PHY_PACKET_REMOTE_ACCESS

This structure generically describes a PHY Remote Access (PHY Register Read) packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;          // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR page_2_1:2;        // Bits 1 and 2 of the page field.
            UCHAR type:4;           // Should be PHY_EXTENDED_READ_BASE_REGISTER (1) or
                                   // PHY_EXTENDED_READ_PAGED_REGISTER (5).
            UCHAR T:1;              // Should be zero.
            UCHAR R:1;              // Should be zero.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR reg:3;            // The reg field.
            UCHAR port:4;           // The port field.
            UCHAR page_0:1;         // Bit 0 of the page field.
        };

        // The fourth byte (Base Address + 3).
        UCHAR Reserved;
    };
}
C1394_PHY_PACKET_REMOTE_ACCESS, *PC1394_PHY_PACKET_REMOTE_ACCESS;
```

C1394_PHY_PACKET_REMOTE_REPLY

This structure generically describes a PHY Remote Reply (PHY Register Read) packet.

```

typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;          // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR page_2_1:2;        // Bits 1 and 2 of the page field.
            UCHAR type:4;            // Should be PHY_EXTENDED_BASE_REGISTER_CONTENTS (3)
                                     // or PHY_EXTENDED_PAGED_REGISTER_CONTENTS (7).
            UCHAR T:1;               // Should be zero.
            UCHAR R:1;               // Should be zero.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR reg:3;             // The reg field.
            UCHAR port:4;            // The port field.
            UCHAR page_0:1;          // Bit 0 of the page field.
        };

        // The fourth byte (Base Address + 3).
        UCHAR Data;
    };
}
C1394_PHY_PACKET_REMOTE_REPLY, *PC1394_PHY_PACKET_REMOTE_REPLY;

```

C1394_PHY_PACKET_REMOTE_COMMAND

This structure generically describes a PHY Remote Command packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;    // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR:2;           // Should be zero.
            UCHAR type:4;      // Should be PHY_EXTENDED_REMOTE_COMMAND (8)
                               // for remote command packets.
            UCHAR T:1;         // Should be zero.
            UCHAR R:1;         // Should be zero.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR:3;           // Should be zero.
            UCHAR port:4;
            UCHAR:1;           // Should be zero.
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR cmdnd:3;
            UCHAR:5;           // Should be zero.
        };
    };
};
C1394_PHY_PACKET_REMOTE_COMMAND, *PC1394_PHY_PACKET_REMOTE_COMMAND;
```

C1394_PHY_PACKET_REMOTE_CONFIRMATION

This structure generically describes a PHY Remote Confirmation packet.

```

typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;           // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR:2;                 // Should be zero.
            UCHAR type:4;            // Should be PHY_EXTENDED_REMOTE_CONFIRMATION (0xA)
                                     // for remote confirmation packets.
            UCHAR T:1;               // Should be zero.
            UCHAR R:1;               // Should be zero.
        };

        // The third byte (Base address + 2).
        struct
        {
            UCHAR:3;                 // Should be zero.
            UCHAR port:4;
            UCHAR:1;                 // Should be zero.
        };

        // The fourth byte (Base address + 3).
        struct
        {
            UCHAR cmdnd:3;
            UCHAR ok:1;
            UCHAR disabled:1;
            UCHAR bias:1;
            UCHAR connected:1;
            UCHAR fault:1;
        };
    };
};
C1394_PHY_PACKET_REMOTE_CONFIRMATION, *PC1394_PHY_PACKET_REMOTE_CONFIRMATION;

```

C1394_PHY_PACKET_RESUME

This structure generically describes a PHY Resume packet.

```
typedef union
{
    // Access each quadlet (beware the endianness stuff).
    QUADLET PhyQuadlet[2];

    struct
    {
        // The first byte (Base address).
        struct
        {
            UCHAR RootID:6;
            UCHAR PHYType:2;    // Should Be Binary 00 (PHY_TYPE_ZERO).
        };

        // The second byte (Base Address + 1).
        struct
        {
            UCHAR:2;           // Should be zero.
            UCHAR type:4;      // Should be PHY_EXTENDED_RESUME (0xF) for remote command pkts.
            UCHAR T:1;         // Should be zero.
            UCHAR R:1;         // Should be zero.
        };

        // Bytes 3 and 4.
        UCHAR zeroes[2];
    };
}
C1394_PHY_PACKET_RESUME, *PC1394_PHY_PACKET_RESUME;
```

C1394_PHY_PACKET

This structure describes any PHY packet.

```
typedef union
{
    // Access to each byte separately.
    UCHAR Bytes[8];

    union
    {
        // Generic alias to help find out what the PHY packet actually is.
        C1394_PHY_PACKET_GENERIC;

        // Self-ID packets.
        C1394_PHY_PACKET_SELF_ID_0    SelfID0;
        C1394_PHY_PACKET_SELF_ID_1    SelfID1;
        C1394_PHY_PACKET_SELF_ID_2    SelfID2;
        C1394_PHY_PACKET_SELF_ID_3    SelfID3;
        C1394_PHY_PACKET_SELF_ID_N    SelfIDN;

        // Link ON Packet
        C1394_PHY_PACKET_LINK_ON      LinkOn;

        // Configuration Packet.
        C1394_PHY_PACKET_CONFIGURATION    Configuration;

        // Extended PHY Packets
        union
        {
            C1394_PHY_PACKET_EXTENDED;
            C1394_PHY_PACKET_PING                Ping;
            C1394_PHY_PACKET_REMOTE_ACCESS        RemoteRead;
            C1394_PHY_PACKET_REMOTE_REPLY        RemoteReply;
            C1394_PHY_PACKET_REMOTE_COMMAND      RemoteCommand;
            C1394_PHY_PACKET_REMOTE_CONFIRMATION RemoteConfirmation;
            C1394_PHY_PACKET_RESUME              Resume;
        }
        Extended;
    };
}
C1394_PHY_PACKET, *PC1394_PHY_PACKET;
```


The following macros can help in the manipulation of PHY packet structures:

PHYPacketIsValid(PC1394_PHY_PACKET pPhyPacket)

Returns TRUE if the second quadlet of the PHY packet pointed to by *pPhyPacket* is not the binary complement of the first quadlet, otherwise it returns FALSE.

PHYPacketIsValid(PC1394_PHY_PACKET pPhyPacket)

The logical reverse of **PHYPacketIsValid**.

PHYIsConfigurationPacket(PC1394_PHY_PACKET pPhyPacket)

Returns TRUE if the PHY packet pointed to by *pPhyPacket* is a valid PHY configuration packet.

PHYIsExtendedPacket(PC1394_PHY_PACKET pPhyPacket)

Returns TRUE if the PHY packet pointed to by *pPhyPacket* is an extended PHY packet.

PHYPacketExtendedType(PC1394_PHY_PACKET pPhyPacket)

Returns the *type* field of an extended PHY packet. The possible values for this field are shown in the table below (defined in P1394A).

Extended PHY Packet Types	
PHY_EXTENDED_PING	0
PHY_EXTENDED_READ_BASE_REGISTER	1
PHY_EXTENDED_READ_PAGED_REGISTER	5
PHY_EXTENDED_BASE_REGISTER_CONTENTS	3
PHY_EXTENDED_PAGED_REGISTER_CONTENTS	7
PHY_EXTENDED_REMOTE_COMMAND	8
PHY_EXTENDED_REMOTE_CONFIRMATION	A ₁₆
PHY_EXTENDED_RESUME	F ₁₆

PHYPacket_Get_page(pPhyPacket, RemoteType)

PHYPacket_Set_page(pPhyPacket, RemoteType, page)

These macros help handle the manipulation of the *page* field of the extended PHY remote access and remote reply packets. This 3-bit field spans a byte boundary so the platform-independent big endian definitions of the C1394_PHY_PACKET_REMOTE_ACCESS and C1394_PHY_PACKET_REMOTE_REPLY types had to break this field in two. These macros help in the handling of this broken-apart field.

RemoteType should be either *RemoteRead* or *RemoteReply*.

Status Codes Reference *(alphabetical listing)*

STATUS_1394_ABORTED

The operation was cancelled/aborted by the miniport because it could not be performed (for example there is a critical hardware error that halted the adapter, or the miniport is unloading).

STATUS_1394_ADAPTER_ERROR

A non-critical error occurred on the adapter that caused the request to fail. The error affected only this operation and the adapter is otherwise functioning normally.

STATUS_1394_ALREADY_OPEN

A status code returned by **C1394OpenAdapter**, in the case when the application has already opened the specified adapter. The same handle value is returned but its client-side reference count is increased.

STATUS_1394_BR_LIMIT

The operation was cancelled because more bus resets occurred than the operation could tolerate.

STATUS_1394_BUS_RESET

The operation was cancelled because a 1394 bus reset took place. For example a queued asynchronous transaction must be failed because the target NodeID might not be the same after the bus reset.

STATUS_1394_CONFLICT

The operation could not be completed because there was a conflict.

STATUS_1394_CRC_ERROR

An isochronous packet was received with a CRC error.

STATUS_1394_CRITICAL_ADAPTER_ERROR

A critical error occurred on the adapter that caused the request to fail. The adapter is not fully functional and the class driver should take corrective actions (closing an isochronous channel, resetting the adapter etc).

STATUS_1394_DEVICE_BUSY

The operation requested failed because the local adapter is busy.

STATUS_1394_DEVICE_NOT_FOUND

The transmit request was aborted because the target device was not found on the bus.

STATUS_1394_DRIVER_INTERNAL_ERROR

The driver detected a bug in its execution logic (through assertions or sanity checks), and could not complete the operation.

STATUS_1394_DMA_ERROR + xxx

The isochronous operation request failed with an OHCI event code that is not mapped to a more specific error code (like STATUS_1394_FIFO_OVERRUN or STATUS_1394_LONG_PACKET).

STATUS_1394_DMA_LIMIT

The isochronous operation request could not be queued because its size exceeds the maximum DMA transfer size permitted by the operating system.

STATUS_1394_DUPLICATE_CHANNEL

The isochronous channel number is already in use.

STATUS_1394_FIFO_OVERRUN

An isochronous operation failed because of a FIFO underrun.

STATUS_1394_FIFO_UNDERRUN

An isochronous operation failed because of a FIFO overrun.

STATUS_1394_GAP_COUNT_ERROR

The class driver detected that not all the self-ID packets that were transmitted after the last bus reset contained the same value in their *gap_count* field.

STATUS_1394_INCORRECT_RESPONSE

An outgoing response packet is not valid.

STATUS_1394_INSUFFICIENT_RESOURCES

An operation failed due to some lack of resources other than memory. If a memory allocation failed then preferably STATUS_1394_NO_MEMORY should be used.

STATUS_1394_INVALID_BUFFER_SIZE

The size specified for the supplied buffer is not the expected for the operation requested. This will most probably mean that the buffer is smaller than needed.

STATUS_1394_INVALID_CHANNEL_TYPE

The operation requested is not applicable to the channel type.

STATUS_1394_INVALID_CHANNEL_STATE

The operation requested is not possible at the channel's current operating state.

STATUS_1394_INVALID_DEVICE_STATE

The adapter is in a state from where the requested operation cannot be performed.

STATUS_1394_INVALID_HANDLE

The miniport adapter handle passed to the miniport does not identify one of its adapters.

STATUS_1394_INVALID_ISOCHRONOUS_BUFFERS

A validation check on a number of isochronous buffers failed.

STATUS_1394_INVALID_OFFSET

The 1394 address space offset specified was above the highest 1394 offset (0xFFFFFFFF), or the address range specified spanned the highest 1394 offset.

STATUS_1394_INVALID_PARAMETER

One of the parameters to function was invalid, and there is no other more specific error for the situation (for example STATUS_1394_INVALID_HANDLE or STATUS_1394_INVALID_BUFFER_SIZE).

STATUS_1394_INVALID_REQUEST

The request passed to the miniport was invalid, and there was no other more specific error status for this operation.

STATUS_1394_INVALID_RESPONSE

An response packet was received that was invalid for the corresponding transaction request, or the class driver detected that an outgoing response is not complying with the rules set by the standard with regards to header information (destination NodeID, transaction code, data length).

STATUS_1394_LOCK_FAILED

A call to **C1394CompareSwap** failed.

STATUS_1394_LONG_PACKET

An isochronous receive operation failed because one or more packets were received whose payload was greater than the maximum payload specified in the request.

STATUS_1394_NO_MEMORY

An operation failed due to memory allocation failure.

STATUS_1394_NOT_FOUND

The requested item was not found.

STATUS_1394_NOT_IMPLEMENTED

The request passed to the miniport was valid, the adapter hardware supports the functionality but the miniport does not implement this feature.

STATUS_1394_NOT_SUPPORTED

The request passed to the driver was valid, but the driver software or the adapter itself does not support the required functionality.

STATUS_1394_PENDING

The request is queued in the miniport and will be processed asynchronously.

STATUS_1394_SELFID_ERROR

The class driver detected that there was some structural/logical problem in the self-ID packets that were transmitted after the last bus reset.

STATUS_1394_SIZE_LIMITATION

The request cannot be transmitted because the payload size cannot be transmitted on the path to the destination node.

STATUS_1394_SPEED_LIMITATION

The request cannot be transmitted because the payload size is greater than the maximum allowed for the adapter's speed.

STATUS_1394_SUCCESS

The requested operation completed successfully.

STATUS_1394_TIMEOUT

The requested operation was not completed in the expected amount of time.

STATUS_1394_TOPOLOGY_ERROR

The class driver detected that although there was no structural/logical problem in the self-ID packets that were transmitted after the last bus reset, the bus topology described by them is not valid.

STATUS_1394_TRANSACTION_FAILED

A transaction request has been sent, its corresponding response was received, and the response code indicated that the transaction was not completed successfully.

STATUS_1394_UNSOLICITED_RESPONSE

An application tried to transmit a response that did not have a matching request. This can happen if the response is indeed unsolicited, or if the application was so late that the request was timeout and an additional split transaction timeout occurred and the entry was cleared.

STATUS_1394_UNSUCCESSFUL

The requested operation failed and there is no other more specific error status for the situation.

Change History *(reverse chronological order)*

December 2008 – 5.60

- VersaPHY support added to the FireAPI programming interface.

August 2008 – 5.52

- No additions to the FireAPI programming interface. Additions were made to the Fire-i interface with the addition of the FireiX COM library; please see the relevant documentation.

June 2008 – 5.51

- No additions to the FireAPI programming interface. Additions were made to the Fire-i interface; please see the relevant documentation.

March 2008 – 5.50

- Added DMA Multiplexing support (MultiDMA) with three modes of operation.
- Added `MultiDmaOperation` enumeration for the available MultiDMA modes.
- Added `OID_MULTIDMA_MODE` identifier to retrieve/set the current MultiDMA mode through **C1394QueryInformation/C1394SetInformation**.
- Added `OID_ISO_RECEIVE_DMA_CONTEXTS` and `OID_ISO_TRANSMIT_DMA_CONTEXTS` to **C1394QueryInformation** so that applications and designers can query the actual number of hardware DMA contexts available for isochronous receive and transmit.
- Added new error code for isochronous receive operations: `STATUS_1394_LONG_PACKET`.
- Added the *C1394Retry* functions:
 - **C1394MayRetryTransaction**
 - **C1394RetryReadNodeInQuads**
 - **C1394RetryReadNodeExInQuads**
 - **C1394RetryReadDeviceInQuads**
 - **C1394RetryWriteDeviceInQuads**
- Removed support for User Mode *Bus Reset Exceptions* (**OID_BUS_RESET_EXCEPTIONS**).
- Removed UB1394DH.LIB from FireAPI. Merged into UB1394.LIB. UB1394DH.DLL remains in ubCore as a *DLL function forwarder* that forwards all calls to UB1394.DLL.
- Minor updates to Device Handle functions' documentation.
- Removed FIREI.LIB from FireAPI. Merged into UB1394.LIB. FIREI.DLL remains in ubCore as a *DLL function forwarder* that forwards all calls to UB1394.DLL.
- Added configurable “Minimum Bus Reset Interval” controllable through the `OID_MIN_BUS_RESET_INTERVAL` identifier. Default value is 2 seconds as dictated by the 1394 specifications.
- Added configurable delay between successive successful configuration ROM reads during Plug'n'Play bus enumeration. Controllable through the `OID_ENUM_READ_DELAY` identifier.
- Added configurable delay between retries in configuration ROM reads during Plug'n'Play bus enumeration. Controllable through the `OID_ENUM_RETRY_DELAY` identifier.
- Added VC6/VS2005 BATCH BUILD generator for FireAPI sample code.

July 2007 – 5.21

- Added the `OID_DMA_LIMIT` identifier to **C1394QueryInformation** to discover maximum DMA transfer size permitted by the operating system.
- Added new isochronous receive method (`ISOCH_OP_RCV_FIXED_DATA_NH`) for flawless isochronous fixed data receive on 64-bit systems.

June 2006

- Added function **C1394ReadPHYRegister** to allow easy access to remote PHY registers.
- Compliance with 1394 Base Test Suite (defined by 1394 Trade Association).
- Improved stability of bus reset storm handling.

February 30th 2005

- Made the ubCore 1394 drivers full plug and play compatible. Now any 1394 devices connected to the bus are enumerated by the ubCore drivers and presented to the system in the registry and the device manager. A plug and play driver can be created for each device separately if there is need for it or the user mode API can be used in order to directly access the device.
- Added a WDM compliant IIDC camera driver based on ubCore.
- Added a WDM compliant DV camera driver based on ubCore. This driver is also provided as a kernel mode source sample.

November 17th 2003

- Added function **C1394PingNode** which gives the ability to ping a 1394 bus node and get a ping response time.
- Added functionality to the ubCore drivers for asynchronous streaming operations. The functionality was internally implemented and is available to the user through the existing isochronous receive mechanism for reception and **C1394TransmitPackets** for transmission.

May 5th – 20th 2003

- Added function **C1394GetAddAdapterEventHandle** which returns an event that is set when a 1394 adapter is added to the system (either enabled or physically added).
- Added the event **EventMiniportAdapterRemoved** which indicates that a host adapter has been removed (either disabled or physically removed).
- Added the flag **ALLOCATE_MAX_REQUESTS** in the field **fAdapterChannelOptions** of the **FIREAPI_CHANNEL_PARAMETERS** structure used in function **C1394OpenAdapterChannel**.

September 19th 2000

- Increased the number of pending isochronous request to 25. Previous value was 10.

July 11th – 20th 1999

- **VERY IMPORTANT:** Added the **C1394AcknowledgeBusReset** function which implies new behaviour for outgoing asynchronous transactions.
- **VERY IMPORTANT:** Added **OID_BUS_RESET_EXCEPTIONS**.
- **VERY IMPORTANT:** Updated the device handle emulation code.
- Added section on transaction requests spanning multiple address ranges
- Added section on *Common Errors in Transaction Processing*
- Defined **BANDWIDTH_UNITS** macro that calculates the isochronous bandwidth allocation units as a function of payload bytes and transmission speed.
- Added **C1394GetRootNodeID**.
- **VERY IMPORTANT:** Added **OID_AR_PACKET_TRANSFER** and **OID_DEF_AR_PACKET_TRANSFER**, and a section that explains the mechanism involved.

July 1st – 10th 1999

- Added **ACCESS_BROADCAST_LOOPBACK** in the documentation of **C1394MapAddressRange**.
- Removed **RCV_REJECT_LARGER** flag from the options of **ISOCH_OP_RCV_FIXED_PKTS**.
- Added **ISOCH_OP_RCV_FIXED_DATA**, **ISOCH_OP_XMIT_FIXED_DATA**, **ISOCH_OP_XMIT_FIXED_DATA** and **ISOCH_OP_IDLE_CYCLES**.

June 18th – 23rd 1999

- **VERY IMPORTANT:** Major rewrite of introduction, up to function reference. Added multiple samples demonstrating usage of FireAPI functions, and comments on the way to use these functions.
- Added **C1394CompareSwap**.
- Added **C1394DebugPrint** and **KdPrint**.
- Added the **STATUS_1394_LOCK_FAILED**, **STATUS_1394_FIFO_UNDERRUN** and **STATUS_1394_FIFO_UNDERRUN** return codes.

June 9th 1999

- Added **OID_PCI_LATENCY** in **C1394QueryInformation** and **C1394SetInformation**.

June 2nd 1999

- Added comment in **C1394MapAddressRange** with regards to the type of access required on the address range's backing memory.

May 24th 1999

- Added the **FIFO_ISO_XMIT_ZEROED** flag in the flags returned by **OID_ADAPTER_FIFO**.
- Added **OID_ADAPTER_FIFO** to **C1394QueryInformation** and **C1394SetInformation**.
- Added the *Changing FIFO Settings* section.
- Added the **XMIT_LOOP** flag to the isochronous transmit operations.
- Added section for **C1394_LINK_REGISTER_ACCESS** structure.
- Added **OID_RECEIVE_BUFFER_SIZE** to **C1394QueryInformation**.

May 4th 1999

- Added comment in **C1394GetAdapters**, for the behaviour that occurs on multi-adapter systems.
- Added comments for 64-bit integer arithmetic pitfalls.
- Added comment for the **IsochCancelExact** option to **C1394IsochCancel**.
- Removed kernel-API-specific comments from the descriptions of **C1394IsochCancel** and **C1394IsochQueue**.

April 27th 1999

- Clarified the descriptions of the **XMIT_BYTE_SWAP** and **RCV_BYTE_SWAP** options for fixed packet isochronous transmit and receive.

April 25th 1999

- Merged the user mode isochronous specification into the main API specification. Functions added are: **C1394OpenAdapterChannel**, **C1394CloseAdapterChannel**, **C1394IsochQueue**, **C1394IsochCancel**, **C1394GetNextCompleteRequest**, **C1394SetInformation**. Also added the description of the **FIREAPI_ISOCH_REQUEST** structure and the description of the isochronous request-specific parameters.

Feb 15th 1999

- Added note in **C1394TransmitPackets** about capability to transmit a broadcast at a rate higher than the *broadcast speed*. Update on the same issue in the remarks sections of **C1394WriteNode** and **C1394WriteNodeAsynch**.
- Added *Topology Analysis Error Codes* section.
- Added *SelfID Analysis Error Codes* section.
- Corrected the prototype of **C1394TransmitPackets**.
- Completely removed section on isochronous operations. Will be updated in its entirety.

Feb 10th 1999

- Added **OID_SELFID_ANALYSIS_ERROR** and **OID_TOPOLOGY_ANALYSIS_ERROR** in **C1394QueryInformation**.
- Added the *IncorrectSelfID* field in **C1394_NODE_INFO** structure.

Feb 1st 1999

- Added **OID_CYCLE_START_AVAILABLE** to **C1394QueryInformation**.
- Added the description of **OID_LINK_REGISTER** to **C1394QueryInformation**.

Jan 26th 1999

- Reformatted and updated the possible return values of **C1394OpenAdapter**.

Jan 12th 1999

- Added the **STATUS_1394_CONFLICT** to the return values of **C1394MapAddressRange**.