



# FireWire™ Reference Tutorial

(An Informational Guide)

January 22, 2010

## **Abstract**

This document is a FireWire tutorial aimed at engineers that have no previous exposure or understanding of FireWire. Its purpose is to familiarize the reader with FireWire and to build a sound and complete understanding of how the FireWire bus operates from the experiences of expert members of the 1394 Trade Association.

# 1394 Trade Association Specification

1394 Trade Association Specifications, Tutorials and Guides are developed within Working Groups of the 1394 Trade Association, a non-profit industry association devoted to the promotion of and growth of the market for IEEE 1394-compliant products. Participants in Working Groups serve voluntarily and without compensation from the Trade Association. Most participants represent member organizations of the 1394 Trade Association. The work product developed within these working groups represent a consensus of the expertise represented by the participants.

Use of a 1394 Trade Association document is wholly voluntary. The existence of a 1394 Trade Association document is not meant to imply that there are not other ways to produce, test, measure, purchase, market or provide other goods and services related to the scope of the 1394 Trade Association Specification. Furthermore, the viewpoint expressed at the time a document is accepted and issued is subject to change brought about through developments in the state of the art and comments received from users of the specification. Users are cautioned to check to determine that they have the latest revision of any 1394 Trade Association document.

Comments for revision of 1394 Trade Association Documents are welcome from any interested party, regardless of membership affiliation with the 1394 Trade Association. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally, questions may arise about the meaning of specifications in relationship to specific applications. When the need for interpretations is brought to the attention of the 1394 Trade Association, the Association will initiate action to prepare appropriate responses.

Comments on specifications and requests for interpretations should be addressed to:

Editor, 1394 Trade Association  
315 Lincoln, Suite E  
Mukilteo, WA 98275  
USA

1394 Trade Association Documents are adopted by the 1394 Trade Association without regard to patents which may exist on articles, materials or processes or to other proprietary intellectual property which may exist within a specification. Adoption of a document by the 1394 Trade Association does not assume any liability to any patent owner or any obligation whatsoever to those parties who rely on these documents. Readers of this material are advised to make an independent determination regarding the existence of intellectual property rights, which may be infringed by conformance to this document.

Published by

**1394 Trade Association  
315 Lincoln, Suite E  
Mukilteo, WA 98275 USA**

Copyright © 2010 by 1394 Trade Association  
All rights reserved.

## Table of Contents

Author.....	4
Contributors .....	4
Copyright Notices .....	5
IEEE Copyright .....	5
IEEE 1394 Technological Background .....	6
Introduction .....	6
FireWire Applications .....	7
FireWire Products.....	8
Host Adapters .....	9
Cables & Connectors.....	12
Repeaters .....	18
Digital Cameras .....	23
External Hard Disks .....	24
Example 1394 Bus.....	25
Double & Quad Host Adapters.....	27
IEEE1394 Standards.....	30
IEEE1394 Technology .....	31
Module Architecture.....	31
Conceptual Bus Model .....	32
IEEE1394 Backplane VS Cable Environment .....	35
FireWire Speeds & Backwards Compatibility .....	36
1394 Addressing Model .....	40
Size Notation & Endianness.....	42
Bus Reset & 1394 Bus Self-Configuration.....	44
Beta Loops & Redundancy.....	47
1394 Protocol Layering.....	47
1394 Bus Packets.....	48
The 1394 Cycle - Part 1 .....	52
The 1394 Cycle - Part 2: Cycle Start Packets & the Cycle Master .....	53
The 1394 Cycle - Part 3: Cycle Structure & Cycle Drift.....	56
The 1394 Cycle - Part 4: Isochronous & Asynchronous Traffic .....	58
The Essence of Isochronous Traffic .....	60
Isochronous Bandwidth.....	62
	2

Asynchronous Traffic .....	62
Asynchronous Stream Packets .....	63
Packet Size Restrictions .....	65
Link Layer Operation .....	66
Transaction types: Read, Write, Lock .....	67
The CSR Model.....	69
Configuration ROM & GUIDs.....	71
Transaction types.....	72
DMA.....	73
DMA Contexts & Context Programs .....	75
Serial Bus Management.....	77

**Author:**

Dimitrios Staikos                      Codemost Technology Co. Ltd.

**Contributors:**

Les Baxter                              Baxter Enterprises

Burke Henehan                      Henehan Consulting

Don Harwood                      PLX Technologies

Eric Anderson                      Apple

All members of the 1394 Trade Association

Photos courtesy of Point Grey Research Inc. and IOI Technology Corporation

## Copyright Notices

### IEEE Copyright

*Portions of this specification are copied from published IEEE standards, by permission.*

*The source documents are:*

*IEEE Std 1212-2001, Standard for a Control and Status Registers (CSR) Architecture for Microcomputer Buses*

*IEEE Std 1394-2008, Standard for a High-Performance Serial Bus*

*The IEEE copyright policy at <http://standards.ieee.org/IPR/copyrightpolicy.html> states, in part:*

*Royalty Free Permission*

*IEEE-SA policy holds that anyone may excerpt and publish up to, but not more than, ten percent (10%) of the entirety of an IEEE-SA Document (excluding IEEE SIN books) on a royalty-free basis, so long as:*

*1) Proper acknowledgment is provided;*

*2) The 'heart' of the standard is not entirely contained within the portion being excerpted.*

*This includes the use of tables, graphs, figures, abstracts and scope statements from IEEE Documents.*

## IEEE 1394 Technological Background

### Introduction

This document aims to provide an extensive description of IEEE1394 that will familiarize the reader with all the significant technological aspects, terminology and features of the IEEE1394 technology.

“IEEE1394 High Performance Serial Bus” is the official name for this technology, which is also known with the commercial trademark FireWire, a trademark owned by Apple. Additionally the i.LINK trademark was registered by Sony for its own implementation using new small connectors.

This document is written for software and hardware engineers who desire to acquire a technical background on IEEE1394 that covers more than the surface, marketing-grade, material that appears in articles found in tech magazines and web sites. Still the document does not ascribe to explaining everything. An effort has been made to keep its size reasonable (60-70 pages including lots of diagrams) so that the expectation that engineers will actually read it can be reasonably high.

The goals are twofold:

1. Help engineers with their FireWire technology evaluations.

“*Should we use FireWire?*” is a very important question that engineers are called upon to answer. These engineers are usually faced with serious time constraints and don’t have a considerable wealth of resources to help them get a clear idea about FireWire so that they can make an educated choice.

FireWire is not simple but this is a natural price to pay for power and flexibility. It is a goal of this document to help engineers get a solid understanding of what FireWire is all about so that they can make their choice based on a full understanding of the bus’ true capabilities instead of fear and confusion infused by competing technology evangelists.

2. Help engineers who work with FireWire make better/correct use of it.

Even after making the decision to work with FireWire many engineers still don’t have a level of understanding that will allow them to use the full capabilities of the bus and build a successful solution. The capabilities of FireWire are incorrectly used, or in the best of cases underutilized. When troubleshooting is required no one is sure about what to do first, because no one truly understands what is going on even when the system works without trouble. The question to the engineer now is “*What on earth is going on?*”

As a natural consequence of this lack of understanding FireWire is unfairly blamed as too complex and unreliable.

It is a goal of this document to help engineers get a solid understanding of what FireWire is all about so that they can use it properly to build successful solutions that they know how to troubleshoot.

## FireWire Applications

We begin the description of FireWire by listing some of the application areas where FireWire is used.

The majority of industrial FireWire applications involve digital cameras. Such applications could be generically categorized as "machine vision" applications, but this is a very broad term. As FireWire is getting more recognized as a reliable solution for such "machine vision" applications, different types of industrial applications have also emerged.

FireWire hard disks are extensively used in storage solutions and hold a considerable portion of the FireWire market.

FireWire applications at the moment include:

- **Robotic Control**  
Such systems use FireWire cameras as "environmental sensors". The computer "sees" the environment through FireWire cameras, performs image analysis and provides movement instructions to robotic "hands" that need to interact with the environment (pick up objects, place objects into new positions, etc).
- **Automated Optical Inspection**  
Such systems use FireWire cameras to take "photos" of products manufactured in automated assembly lines (e.g. PC motherboards, PC adapters, cell phones, etc) and examine whether the said artifacts appear as if they have been constructed properly (i.e. they have no visible discrepancies).
- **Medical Imaging**  
Such systems might use FireWire cameras to create 3-dimensional models of a patient's face or body. These models are then used in various ways through the medical procedures.
- **Filming**  
Such systems might use arrays of FireWire cameras to create special 3-dimensional visual recordings used in special visual effects.
- **Security Surveillance**  
FireWire cameras are used to monitor places of interest for security reasons.
- **Storage**  
High performance external hard disks for storage and backup.
- **Communication Systems**  
FireWire is used as an internal local network in data centers for high speed server-to-server communications.
- **Audio & Pro-Audio**  
Specialized audio applications like amplifier/speaker control, audio channel routing/mixing, audio stream delivery for theater systems and concerts, etc, are done with FireWire.
- **Set-Top Box**  
By FCC regulation all set-top boxes in the USA are required to have a functional FireWire port to permit recording of digital content.
- **Digital Camcorders**  
Many digital camcorders provide a FireWire port to allow easy connectivity to the user's personal computer.
- **Commercial Aviation**  
Such systems use FireWire as a high-capacity local network that can also provide the "Quality of Service" (QoS) required for on demand video streaming in in-flight entertainment systems.



- **Military**  
FireWire is being used as a reliable, high-capacity local network that carries control information and sensor data all over a military aircraft, helicopter or vehicle.
- **Automotive**  
FireWire is making strong efforts to get established as the "in car" communications network for modern and future cars, where services like on demand video, TV, music will be available per passenger seat.

## FireWire Products

This section lists the most commonly available FireWire products. Any product that has a FireWire connector/interface and can communicate over FireWire is considered a "FireWire Product", although it might also have other interfaces too. For example, today most external hard disks have a FireWire, a USB and possibly an eSATA connector.

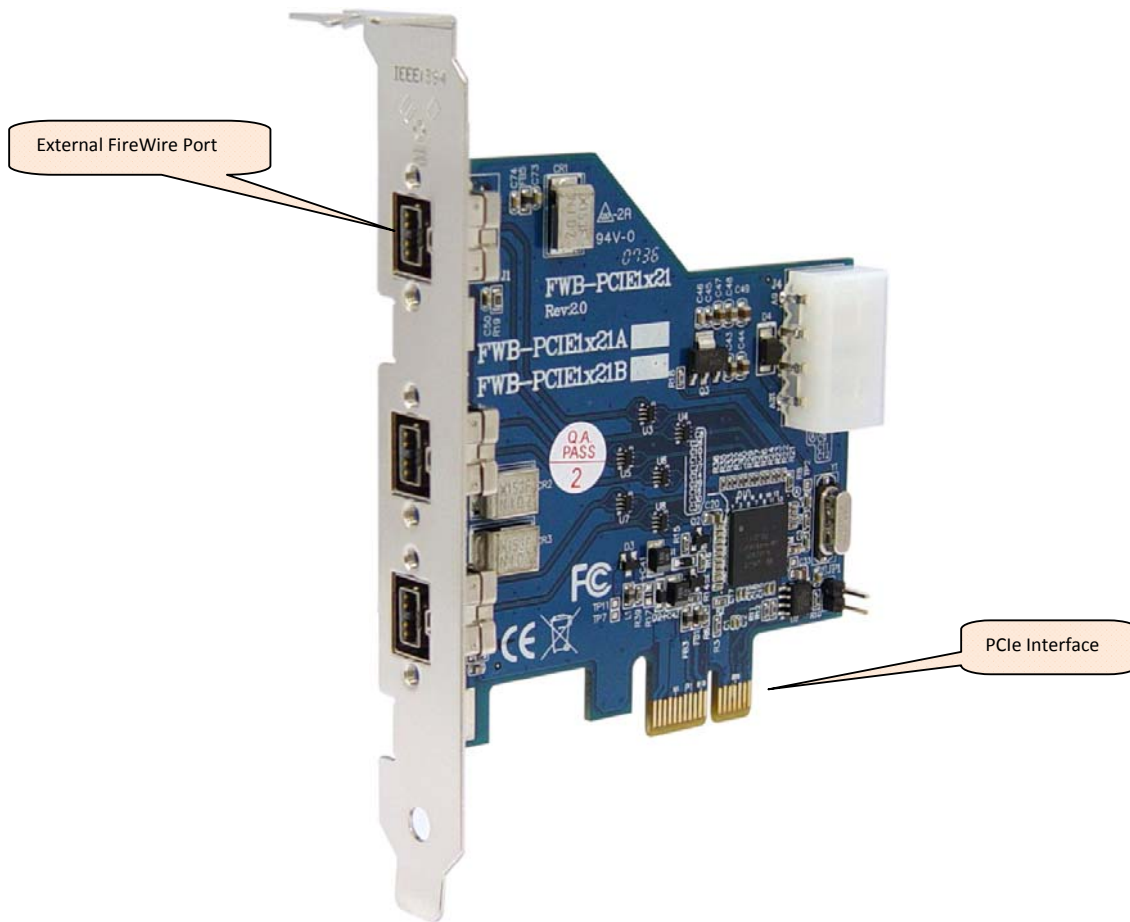
The most common FireWire product categories are shown below:

- **Host Adapters (also known as Host Controllers)**  
These are adapters (cards) that we need to install in a PC so that it obtains a FireWire interface. Many motherboards come with on-board FireWire, so such computers may not need a separate host adapter at all. Still, these too are host controllers that are simply embedded on the motherboard.
- **Cables**
- **Repeaters**  
In essence, a repeater is a cable extender and splitter. If you need to install a FireWire device at a distance greater than the cable in use allows, then you use one or more repeaters so that you can "link" several cables in a row and achieve the desired distance.  
Also, if your FireWire host adapter has 3 ports and you want to connect 5 cameras then you might need one or more multiport repeaters to provide the additional needed ports. For example a 4-port repeater can connect 3 cameras to a single port of a host adapter.
- **Digital Cameras**  
The vast majority of FireWire cameras adhere to the IIDC (Industrial Imaging Digital Camera) standard. This is a standard for the software interface of the camera and the types of video formats that it provides. All non-IIDC implementations are considered "custom" and are used for specialized applications (security, military, etc).
- **External Hard Disks**
- **Digital Camcorders**

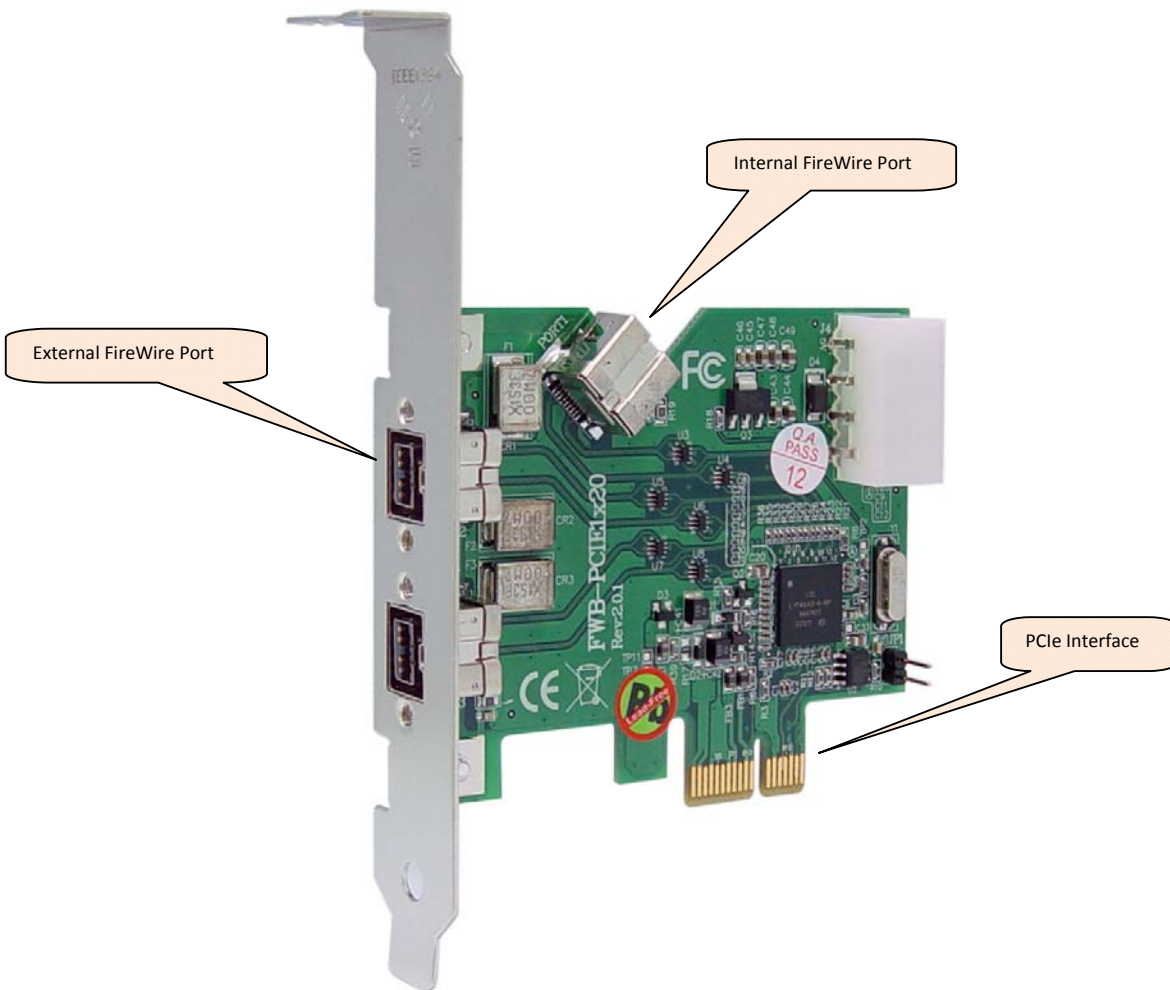
The following sections illustrate some sample FireWire products.

### Host Adapters

This is a FireWire host adapter with 3 external FireWire ports and a PCIe interface to connect to the PC:

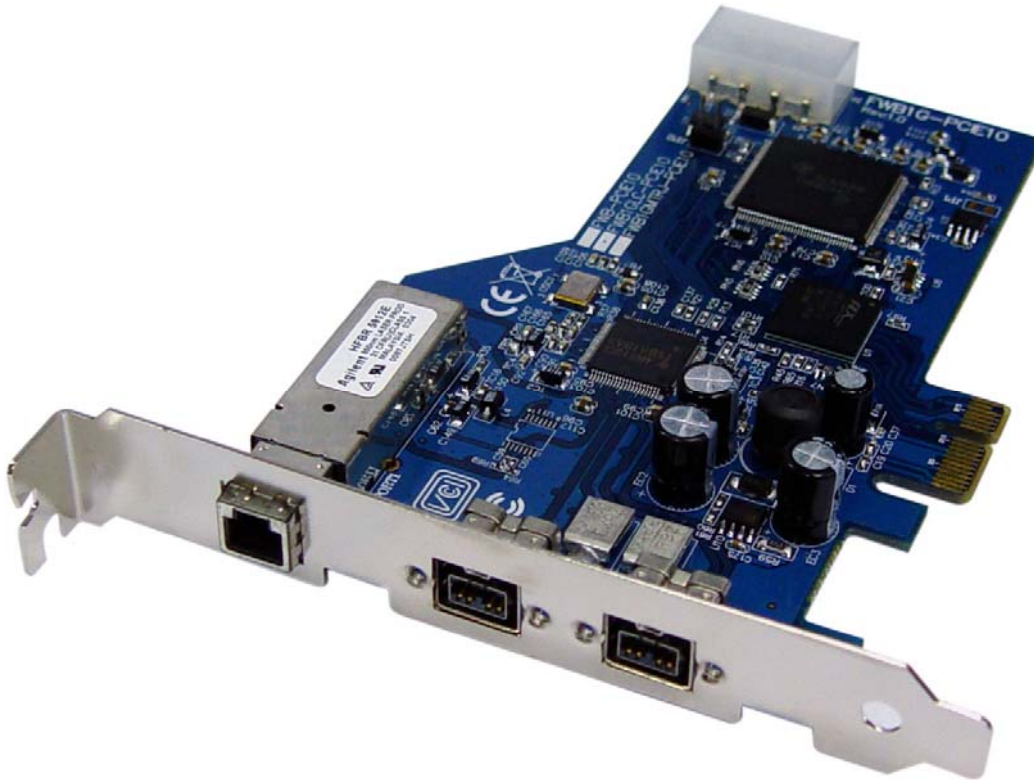


This is a FireWire host adapter with 2 external FireWire ports, 1 internal FireWire port and a PCI-Express interface:



The terms "external" and "internal" refer to whether the port is actually accessible from outside the PC enclosure (visible from the outside).

The next example is a PCI-Express host adapter with 2 FireWire ports and one GOF port:



FireWire data can be transmitted over different types of cables and each type of cable has its own port (connector) as in the above example with the GOF port.

Finally, the next host adapter is an ExpressCard adapter, suitable for laptops:







## Cables & Connectors

FireWire has several different types of ports and thus several different types of cables. The "standard" FireWire cables have 4 different connectors and then there are different connectors for each additional type of cables (CAT5e, POF, GOF, etc). The additional cable types are never referred to as "FireWire cables"; thus the term "FireWire cable" means the standard (original) 1394 cable type.

The existence of 4 different connectors for the standard 1394 cables is a result of the evolution of the 1394 standard and the use of two signaling modes, 1394a and 1394b signaling. However it can lead end users into serious confusion and by many people it is considered as one of the weaknesses of FireWire when compared to technologies like USB.

The table below lists the 4 different connectors for FireWire cables:

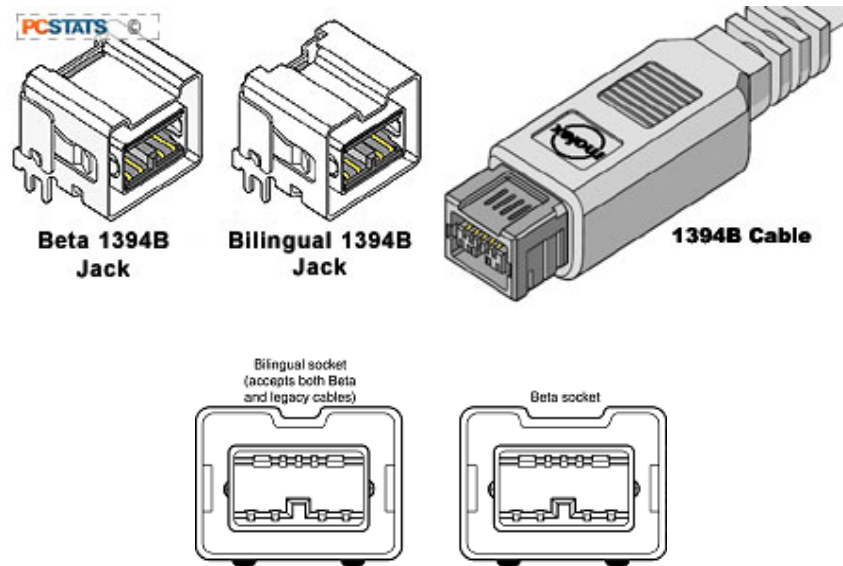
Connector	Characteristics	Photo
4-pin (i.Link)	Can only use 1394a signaling. Speed up to 400Mbps. Cannot carry cable power.	
6-pin	Can only use 1394a signaling. Speed up to 400Mbps. Can carry cable power.	

Bilingual	Can use 1394a or 1394b signaling. Speed up to 800Mbps (specified up to 3200 Mbps). Can carry cable power.	
Beta-only	Can only use 1394b signaling. Speed up to 800Mbps (specified up to 3200 Mbps). Can carry cable power.	

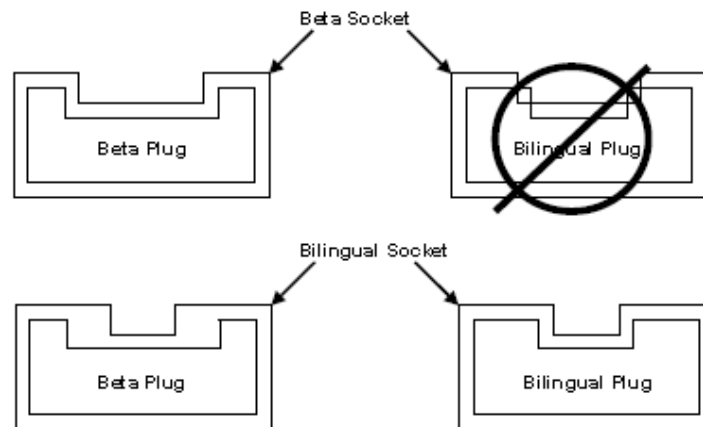
The Bilingual and Beta-only connectors look so much alike that even FireWire experts may have trouble making them apart. Of course, this also means that the plugs for these sockets also look very much alike, leading to even further confusion. So we must clarify the difference here.

By looking at the photos above the difference might not be immediately evident. Beta-only connectors have a wider opening at the top of the socket. Respectively, beta-only plugs (jacks) have a wider opening at the top, as shown in the image below (where a Beta-only cable is shown against the two jacks):





By this insightful design it is possible to insert a Beta-only cable into either jack, but a bilingual cable can only be inserted in the bilingual jack. This is illustrated graphically in the next image which is taken from the IEEE1394-2008 standard:



NOTE: Bilingual plug will not plug into a Beta socket.  
Beta plug will plug into a Bilingual socket.

**Figure 4-46—Interface mating chart**

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 4-46)*

The FireWire cables usually manufactured have the same type of connector at each end: either 6-pin on both ends, or 9-pin on both ends.

However it is generally possible that the cable has different connectors on each end.

The following pictures show FireWire cables with a 4-pin and a 6-pin connector, usually called "4-to-6" cables:





The next pictures illustrate FireWire cables with a 6-pin and a 9-pin bilingual connector (6-to-9 cables), the first one utilizing a locking connector:



 POINT GREY



Of course, in the cables shown above, the 9-pin connector is a bilingual connector. Obviously it would not make sense to make a cable that supports only 1394a signaling on one end and only 1394b signaling on the other end. As a result there is no such thing as a *6pin-to-9pin(BetaOnly)* cable.

There are even 4-to-9 cables, where the 9-pin connector is obviously a bilingual connector:



## Repeaters

As described earlier, 1394 repeaters serve two basic roles:

1. Cable Extenders
2. Cable Splitters

1394 repeaters look similar to USB hubs or Ethernet hubs/switches but they are actually totally different. As their name implies, their single purpose is to **repeat signals between their ports**. Every bit that gets received in one of their ports gets immediately repeated (retransmitted) on their other ports.

The 1394 protocol ensures that incoming data traffic on any device can only come from one port at a time, so there is never a problem or any other complexity with the operation of a 1394 repeater (like buffering packets, etc).

The following fact about 1394 devices must be stressed here:

*All<sup>1</sup> 1394 devices are repeaters.*

This is one of the basic premises of the 1394 design. A host adapter with 3 ports is also a repeater that repeats all signals between its 3 ports, a digital camera with 2 ports is also a repeater that repeats all signals between its two ports, etc.

This way, we can connect a set of 1394 devices in any almost way we like and a 1394 bus is formed, because all the signals are repeated by all devices across the whole cable length.

The devices in the category that we are describing now are "*nothing more than repeaters*" so they are just called "repeaters". This does not mean that they are the only devices that repeat traffic. It means that these devices only repeat traffic and perform no other function.

Most 1394 repeaters have only 3 ports, due to the same technological limitations that were mentioned earlier for host adapters (the available silicon only supports up to 3 ports). These are repeaters with exactly one 1394 node inside them (one 1394 chipset).

However some designs enclose more than one 1394 node to make more repeating ports available. The marketing departments of companies that make such repeaters, tend to market them as "hubs", but it should be clearly understood that they are just composite "signal repeaters".

---

<sup>1</sup> All multi-port FireWire devices. A single port device obviously has no target ports to repeat to.

The following image displays a 3-port repeater with 9-pin bilingual ports:



The picture shows two viewpoints of the device so that we can see all three ports (one port at the top side and one on the left and right sides).

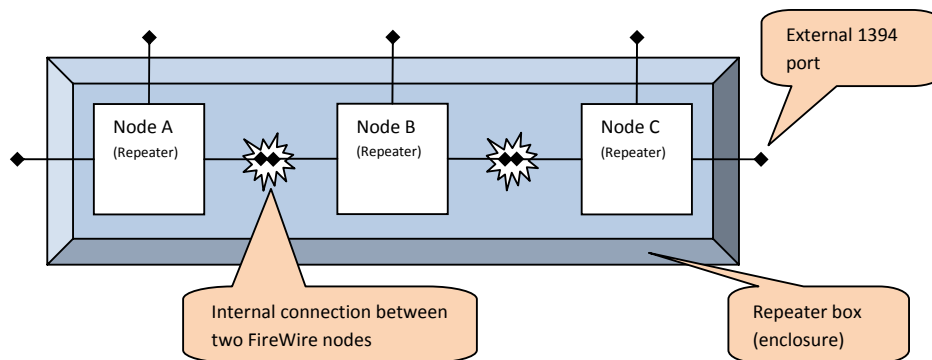
The round hole is a power plug. 1394 repeaters do not only repeat signals. They also repeat cable power (1394 cables can also carry power). A repeater does not require external power to operate. Its power consumption is so small that it can function with the power already available on the the 1394 cable. However, other, more power-hungry devices with no external power may exist on the bus and they expect to find enough power on the cable. The power plug on a repeater acts as an additional "entry point" for bus power so that all devices can function properly.

The following images illustrate two 5-port repeaters (two viewpoints):





As stated earlier, and as can be seen on the diagram on top of the first device, these devices actually contain 3 FireWire nodes, i.e. 3 smaller "elementary" repeaters as shown below:

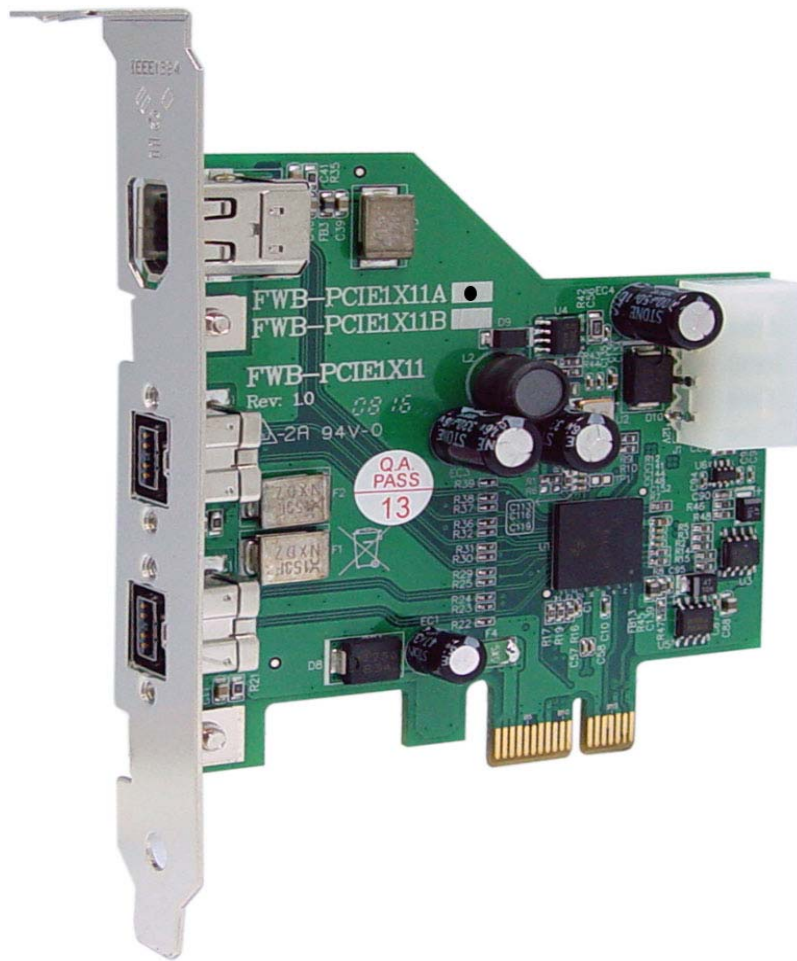


So this device actually contains three 3-port repeaters. These repeaters are connected to each other (so that they are on the same 1394 bus) and the remaining 5 ports are exposed outside the box.

There are also 2-port repeaters, which basically serve as *cable extenders* since they don't provide any new connection point to the bus. The following image shows such a product:



FireWire devices/repeaters can also repeat traffic into ports of different types. The following image shows a 1394 host adapter (which is also a repeater of course) that has two 1394b ports and one 1394a port:



Most "plain repeater" products in the market don't come with a mixture of 1394a and 1394b ports, but may come with mixtures of other types, usually two 1394 ports together with some other cable type.

The picture below illustrates two viewpoints of a repeater by Point Grey with two 1394a ports and a CAT5e port, using GigE signaling to transport the FireWire data over the CAT5e cable:



The next picture below shows two viewpoints of a repeater with two 1394b ports and a port for an optical cable:





## Digital Cameras

As stated earlier, 1394 cameras are present in many FireWire applications. Some people also use the more generic term "sensor", since a camera is nothing more than an optical sensor, usually in the visible spectrum. However, an infra-red camera is most likely to be considered as a "sensor" rather than a camera.

By definition, all 1394 cameras are digital. They capture and encode "images" in digital bits (zeros and ones) and then transmit these digital bits over the 1394 cable.

Most 1394 cameras implement the IIDC protocol, of which the current version is 1.32 (as of 2009, with version 2.0 in the making). This is a "software interface" protocol, which defines a set of registers and commands that an IIDC compliant camera should "understand" (implement) so that software can interrogate the camera and configure it as needed.

The IIDC protocol also defines a set of standard, predefined video formats (image size + pixel encoding + packaging) that cameras may implement, but also allow each manufacturer to define custom video formats.

The reason for the existence of the IIDC standard is of course *interoperability*. When standard video formats are being used, then it should be possible to replace a camera from one manufacturer with a camera from another that supports the same video format, without any need for changes in the software.

Since most of the 1394 cameras in the market adhere to the IIDC protocol, and IIDC applies only to 1394 cameras (i.e. there is no IIDC for GigE cameras), the terms "1394 camera" and "IIDC camera" are in today (2009) interchangeable. However IIDC 2.0 will also support additional interface technologies (GigE, USB, etc).

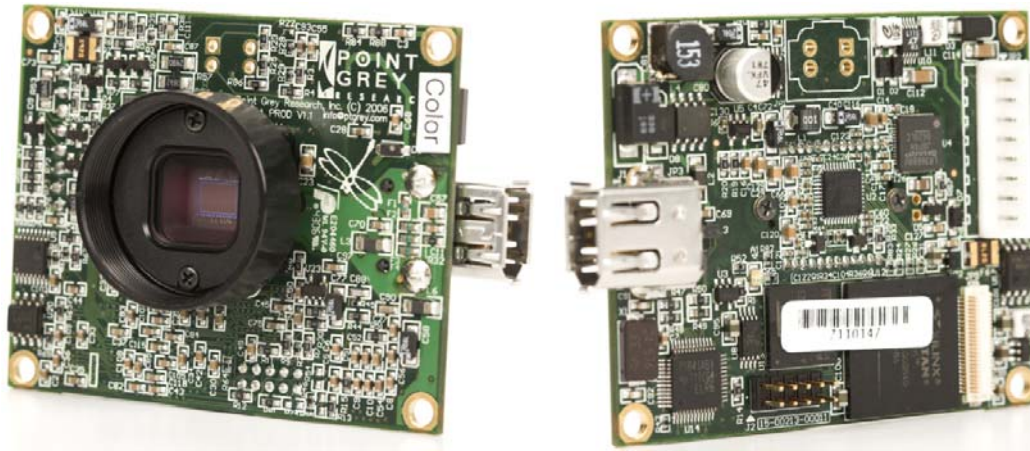
The following image shows an IIDC camera with one 1394b bilingual port:



The socket on the top left corner is for "triggering" purposes, not for power. Most IIDC cameras don't have a power socket; they get the power they need from the 1394 cable (which, as mentioned earlier, carries power).



The picture below displays both sides of a "board" camera, i.e. a camera without any enclosure, so that it can be precisely adapted to the specific needs of the customer:



As shown in the image above, FireWire cameras are often sold without a lens, so that the customer can decide on the exact lens that is appropriate for their needs, and of course they may have just a single FireWire port.

### External Hard Disks

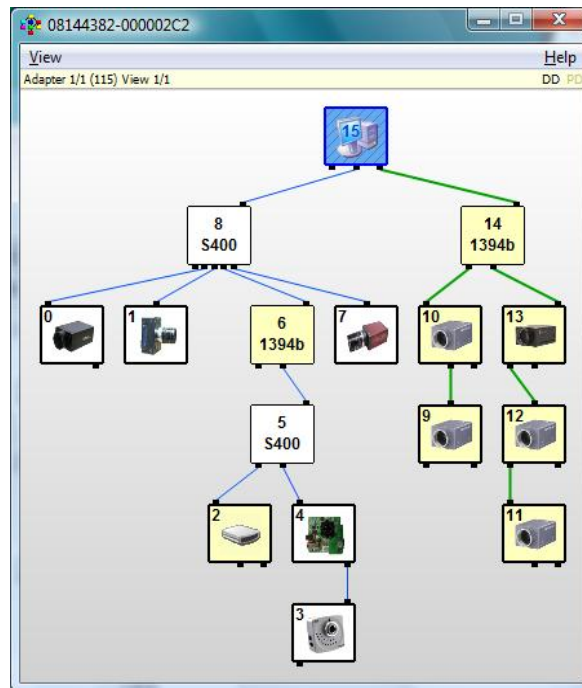
External hard disks today have a multitude of interfaces and FireWire is usually included. The image below shows a hard disk that has two 1394b ports and one 1394a port (all part of a single 1394 "node" so they repeat signals), an eSATA port and a USB port.

External hard disks like the one shown below, usually require more power than the 1394 bus can carry. They cannot operate only on bus power and have a separate power socket (unlike cameras). However many FireWire hard disk models can get powered from the FireWire bus, because the FireWire bus carries much more power than other powered buses like USB.



### Example 1394 Bus

The following image shows an illustration of a 1394 bus as depicted by a software tool:



We will have to run a little ahead of ourselves to explain everything to full detail (not all relevant technical issues have been explained yet), but there is enough we can describe about this typical 1394 bus. For starters, the fact that software can draw this image means that 1394 provides very rich connectivity and topology information. "Smart" devices like cameras and disks also provide a fairly rich description of themselves (camera vendor, model, etc).

So, as seen above, 1394 software has detailed information about how many ports each device has, which port of every device gets connected to which port of another device, etc.

Yellow nodes are 1394b nodes<sup>2</sup> and white nodes are 1394a nodes<sup>3</sup>. The 1394 bus allows the user to connect both 1394a and 1394b devices on the same bus. Blue links are 1394a connections and green links are 1394b connections. It should be apparent that a 1394a device (such as node 8) can only connect to a 1394b device (such as node 6) through a 1394a connection.

Nodes 6 and 14 are 1394b repeaters and nodes 5 and 8 are 1394a repeaters. In reality, node 8 is a host adapter with six 1394a ports, hosted inside a PC that is turned off. Since the device is turned "off", it functions only as a repeater, so the software depicts it that way.

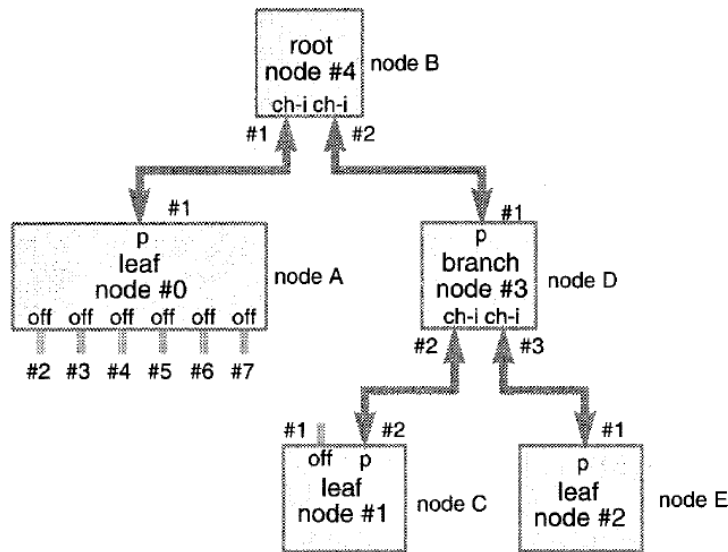
Also note that this is a single node 6-port device. That was only technically possible with chips of 1394a technology. No 1394b chips have been manufactured that support more than 3 ports.

<sup>2</sup> Capable of 1394b signaling and usually also 1394a signaling.

<sup>3</sup> Capable of 1394a signaling only.

Some other facts are worth mentioning in this image. All 1394 buses are logically organized as "non-circular graphs", or "trees". Every connected port is connected to either a "child" node or a "parent" node, with no circles (loops) being formed. At the top of this hierarchy lies the "root" node, which has no parent.

These relationships are also demonstrated in the following diagram from the 1394-2008 standard:



**Figure E.22—Topology information after identification of root node**

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure E.22)*

The parent-child relationships are purely "logical", not functional. They only serve for building up the tree topology and are utilized during the bus arbitration process (selecting which node is going to transmit data) to propagate arbitration requests to the node that acts as arbitrator.

The "parent" does not offer any other "special" services to each "child" nor the other way round, neither does the parent perform any "special" functions on behalf of its children. The devices operate independent of one another and they would operate the same way even if the parent-child relationship was reversed.

Any node, even a repeater, can serve as the root node. However, in practice, a "more capable" node is usually assigned the role of the root node; most often, a PC acts as the root node, as in the picture shown earlier.

When there is only one PC on the bus, then FireWire looks and acts like a peripheral bus; a bus that was made so that peripherals can connect to a PC (like the USB bus).

However, 1394 is not strictly a peripheral bus; it can function as one. In the words of the 1394-2008 standard, paragraph 1.1.1: *This standard describes a high-speed, low-cost serial bus suitable for use as a peripheral bus, a backup to parallel backplane buses, or a local area network.*

The fact that the operating systems of Microsoft and Apple treat FireWire as a peripheral bus does not make FireWire a "peripheral bus".

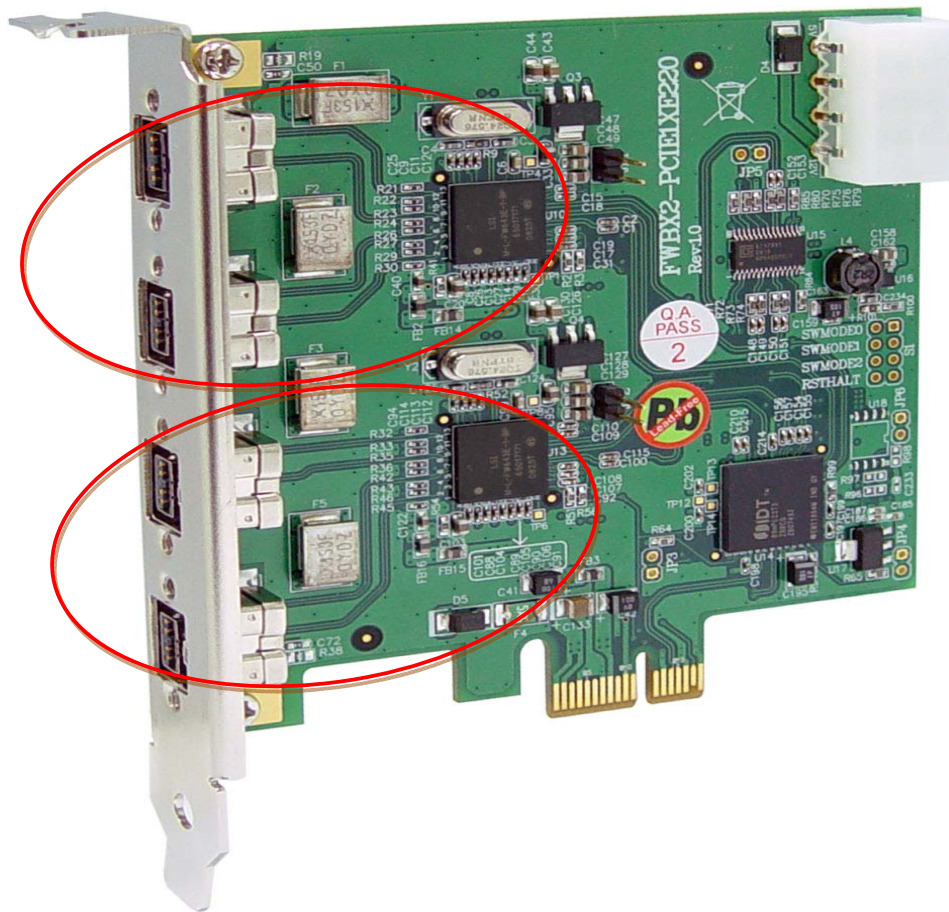
## Double & Quad Host Adapters

In the last couple of years "dual" and "quad" FireWire host adapters have appeared on the market. These devices take advantage of the core PCI specification that allows a physical PCI/PCIe device to have more than one "logical" device in it (i.e. function).

A standard host adapter is a single physical device with just one "function". A dual host adapter is a single physical device with two "functions" and a "quad" adapter has four "functions".

So a "dual" adapter is functionally equivalent to two completely independent host adapters, however it saves space inside the PC, and allows more 1394 "functions" to be installed into a given number of slots.

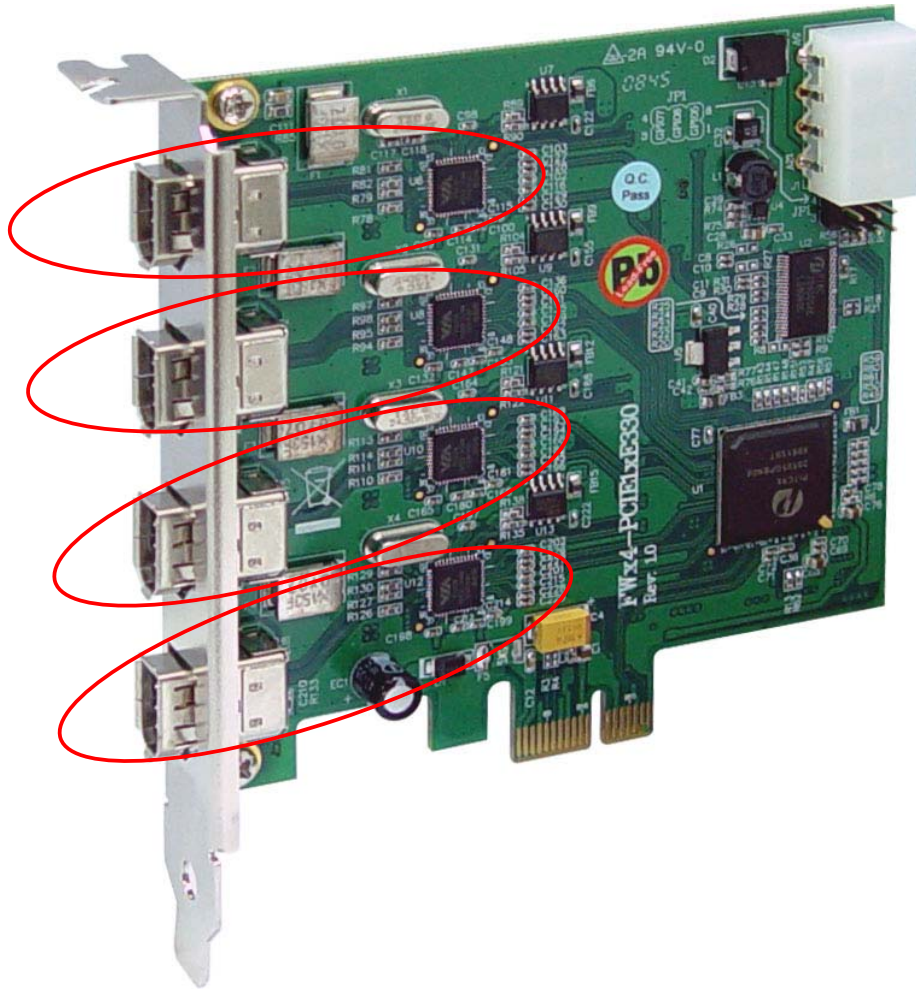
The image below shows a dual adapter and the core of the two functions:



Although this physical device appears to have four 1394 ports, these ports do not repeat traffic between all of them, since each pair belongs to a different adapter and the two adapters are completely independent of one another (not connected to each other; not on the same 1394 bus). The chips used in these adapters support three ports, but there is not enough space to accommodate these ports on the board, so there are only two per adapter.



Similarly, the next image shows a quad adapter:



In this example, there is only one 1394 port exposed for each of the four adapters.

It must be emphasized again that these adapters are logically independent of one another. This means that when computer software asks the operating system "how many 1394 adapters you have?" the answer is "four", not "two dual adapters" or "one quad adapter". *Normal* computer software (applications) has no way to determine if two 1394 adapters come physically in one piece (a dual adapter) or two pieces (two single adapters).

*Special* computer software (device drivers) can get some extra "hints" that an adapter might be dual by examining the PCI-tree topology, but still the answer cannot be definite, because even if this software "sees" a PCI switch connecting two 1394 adapters, it cannot be sure of the physical location of this PCI switch (on a multi-adapter or on the motherboard).

In the same vein, when 1394 device drivers themselves load, the operating system does not distinguish in any way between single, dual and quad adapters. The device driver has no idea there are dual or quad adapters, unless the user (system integrator) has planted special additional information to the system (e.g. in the system registry) that helps the device driver identify these multi-adapters correctly.

The reasons why dual and quad adapters have appeared in the market are several:

- Save space in PCI slots.
- Bandwidth limitations  
Suppose an application that needs to work with 8 cameras, where any 3 cameras combined exceed the bandwidth of FireWire. There can only be 2 cameras per FireWire bus so the application requires 4 buses, which are readily provided by a quad adapter or 2 dual adapters. If 4 separate adapters were used, then a configuration problem would be evident, since in most modern computers motherboards don't have 4 free PCI slots (or 4 free PCIe slots).
- FireWire has a limit of 63 devices per bus. These of course include devices like repeaters. If an application needs to monitor 500 environmental sensors, then these sensors will have to be arranged into multiple FireWire buses.

However, a dual or a quad adapter cannot solve the problem of an application that requires a high resolution video format at a very high frame rate, a combination that results in more than the 800Mbps available by FireWire currently.

## IEEE1394 Standards

The first FireWire standard was released by IEEE in 1995. As the technology evolved, several amendments (extensions) were made to the standard. All these amendments were reconciled into a single document in 2008.

The following list shows the evolution of the 1394 standard and the major contributions on each step:

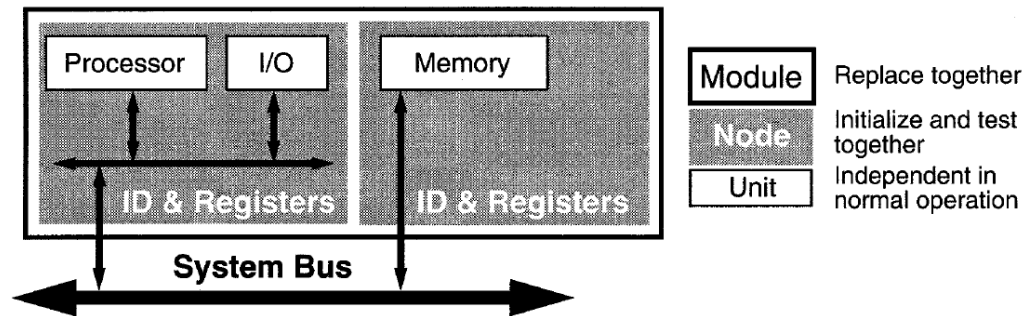
- IEEE1394-1995  
The very first FireWire standard. Introduced bit rates of 100Mbps, 200Mbps and 400Mbps.
- IEEE1394a-2000 (amendment)  
Introduced significant performance improvements (improved efficiency).
- IEEE1394b-2002 (amendment)  
Introduced the 1394b signaling and the 800Mbps transfer rate.
- IEEE1394c-2006 (amendment)  
Electrical specifications for new cable types (CAT5e, CAT6, GOF, POF, GigE)
- IEEE1394-2008  
Incorporates original document and amendments into a single standard. Also introduces electrical specification for 1600Mbps and 3200Mbps data rates.

## IEEE1394 Technology

### Module Architecture

The following reference from the 1394-2008 standard, annex Q1, describes the 1394 "Module Architecture":

*The serial bus architecture is defined in terms of nodes. A node is an addressable entity, which can be independently reset and identified. More than one node may reside on a single module, and more than one unit may reside in a single node, as illustrated in Figure Q.1.*



**Figure Q.1—Module architecture**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.1)

*Each module consists of one or more nodes, which are independently initialized and configured. Note that modules are a physical packaging concept and nodes are a logical addressing concept. A module is a physical device, consisting of one or more nodes that share a physical interface. In normal operation, a module is not visible to software. A node is a logical entity with a unique address. It provides an identification ROM and a standardized set of control registers, and it can be reset independently.*

The text above explains why when we are talking about 1394 bus we so often refer to "nodes". In most cases there is only one "node" in each "module" (a physical device), but there are cases where this is not true (dual & quad adapters, "complex" repeaters).

But these exceptions are rare in comparison to the majority of 1394 devices, so in practice the terms "1394 device" and "1394 node" are interchangeable.



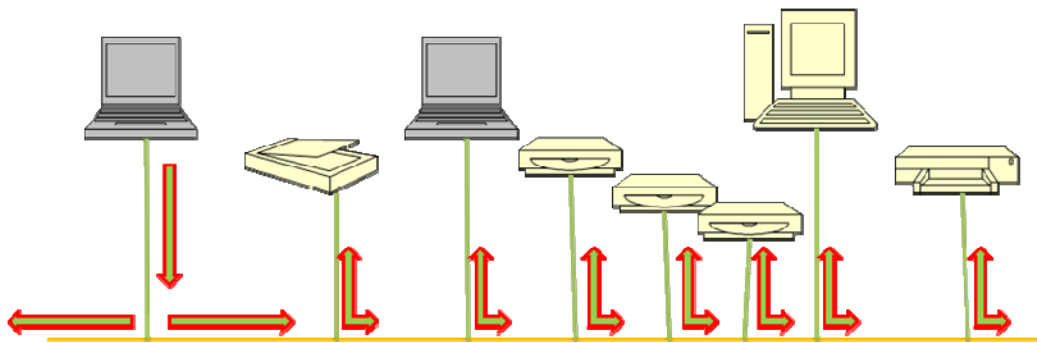
## Conceptual Bus Model

At this stage it is time to make the distinction between a "bus" and a "network". The distinction can be a little blurry, because as we saw earlier, from the reference to 1394-2008 paragraph 1.1.1:

*This standard describes a high-speed, low-cost serial bus suitable for use as a peripheral bus, a backup to parallel backplane buses, or a local area network.*

FireWire is a "bus" that among others is suitable for use as a "local area network". Moreover with the proliferation of so many different "bus" technologies that can also act as a local area network it might be hard to see the distinction.

So at this point we will describe the fundamental conceptual model of a computer "bus" which is shown in the illustration below:



According to this model, there is a "shared communication line" (drawn horizontally across the bottom of the image) and all devices connect directly to this "shared communication line".

There is of course some sort of addressing scheme so that devices can identify themselves with a unique address. However, all traffic is "broadcast" at the wire level. A transmitted packet is "thrown" on the communication line and is "visible" to everyone; it travels through the whole line for everyone to "see".

Among those that "see" the packet is the intended target of the packet. This device recognizes the packet as one that refers to itself and "picks it up" (receives it). The 1394 standard prohibits one device from receiving a packet addressed to another device. An exception to this rule is a bus analyzer that receives all traffic, simply because it "decides" to do so. The traffic is for everyone to "inspect" on the shared communication line, and the data packet does not stop "propagating" across the bus because some node decided to receive it.

The decision to receive a packet is performed automatically by hardware components: The headers of all packets that appear on the communication line are automatically inspected by the hardware of each node. As soon as the hardware recognizes its own address in the packet header, it proceeds to receive the full packet.

Since there is only a single shared communication line available, then there must be a way for deciding who gets to transmit next. This is a process known as "arbitration" and different buses implement it in different ways.

This fundamental bus architecture is exactly how the first local area networks (Thin & Thick Ethernet, 10BASE2/10BASE5) were physically constructed. The shared communication line was injected with BNC connectors so that the devices could physically attach to the shared cable, as shown below:



So were Thin & Thick Ethernet networks or buses after all?

The answer is nicely depicted in a relevant topic in Wikipedia. They were "bus networks":

*A bus network topology is a network architecture in which a set of clients are connected via a shared communications line, called a bus.*

So there is a clear distinction between the shared communications line, which is a "bus" and the entire network that is a "bus network".

This certainly explains some of the possible confusion between what a *bus* is and what a *network* is. However, in all the above cases we are strictly referring to "local area" networks, because only then it is possible to physically share the same communications line.

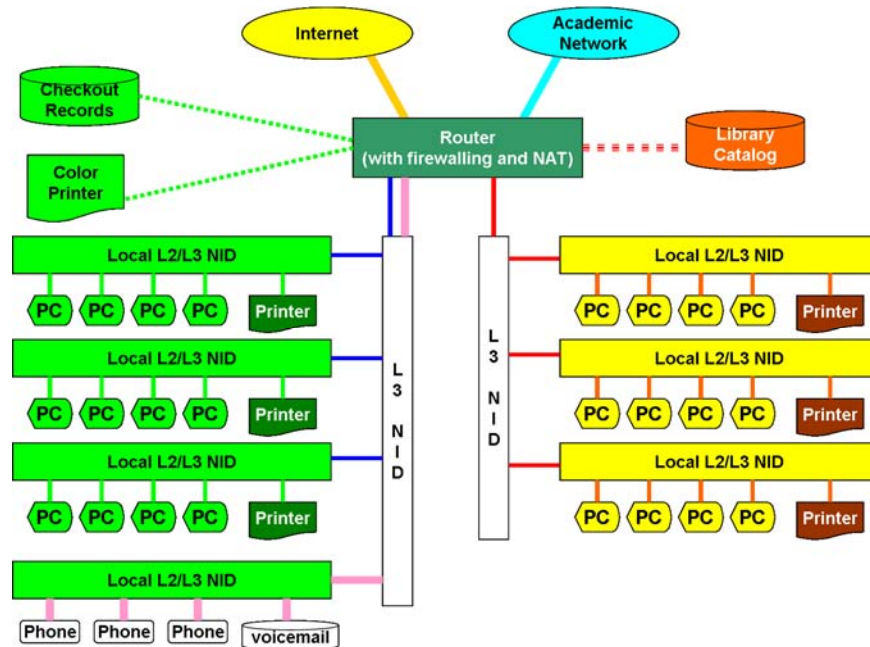
The term "network" is a bit too broad by itself, so more often the correct terms to use are "computer network" or "telecommunications network".

In general, the term "computer network" refers to a "*group of interconnected computers*", where "computers" is taken with its broader meaning; i.e. a printer is also a computer, a router is also a computer, performing highly specialized functions, in contrast to a personal computer that can do "anything" through software.

The term "telecommunications network" refers to a "*network of telecommunications links and nodes arranged so that messages may be passed from one part of the network to another over multiple links and through various nodes*".

These two terms are in fact overlapping to a great degree. Any non trivial local area computer network (e.g. one that spans a whole building) involves many links and many nodes (routers, hubs, etc) that create the illusion of a "local network".

The diagram below depicts one such "local area" computer network:

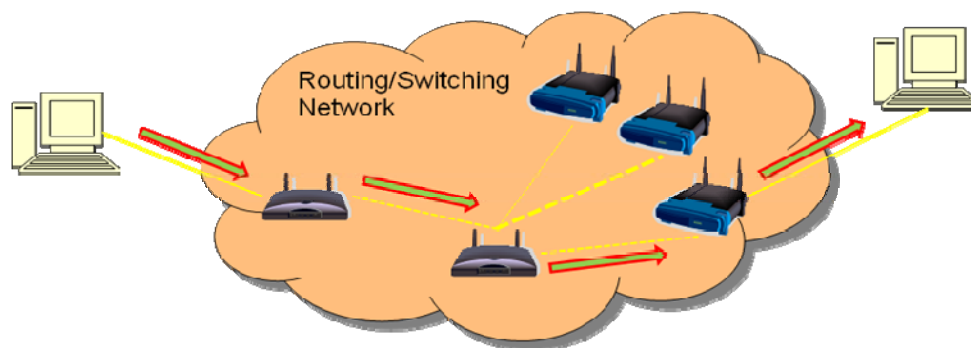


In the above diagram the term NID means "Network Infrastructure Device". L2 and L3 refer to different types of "network links" according to the TCP/IP specification (L2 is "Data Link", L3 is "Network Link").

There are distinctive differences between a bus and a network. In a network:

- The traffic does not get propagated to the entire network.
- The various segments of the network operate locally, completely independent of one another.
- There is no central arbitration about who gets to transmit next. Arbitration (if any) is limited to the local segments.
- Traffic going from one segment to another segment has to be appropriately "routed" to the destination by intermediate devices (hubs, routers).

The conceptual model of a computer/telecommunications network is shown below:



A data packet from one computer to another gets transmitted through the appropriate parts of the network so as to reach the intended destination.

The engineers that design such networks can implement links of different capacities between different parts of the network, so that they can handle the traffic load that is expected in these links.

FireWire is a bus that can act as a local network as well. There is no routing involved. According to the IEEE 1394.1 standard it is possible to interconnect multiple 1394 buses with 1394 bridges and this would create a network that requires routing, but such bridging products are not widely available.

### IEEE1394 Backplane VS Cable Environment

The conceptual bus model that was shown earlier, consisting of a single backbone line with nodes connecting to it, may be conceptual, but it also exists in practice as a physical implementation and is usually called a "Backplane Bus".

Indeed the IEEE1394 standard defines 1394 for two "environments" as shown in the reference below from IEEE1394-2008 annex Q2:

*The physical topology of the serial bus is divided into two "environments," as shown in Figure Q.2. One is called the **backplane environment** and is defined in this standard, although implementations may require additional physical-layer descriptions contained within other backplane bus standards. The other part is the **cable environment** and is completely specified in this standard. Interconnected nodes may reside in either environment without restriction.*

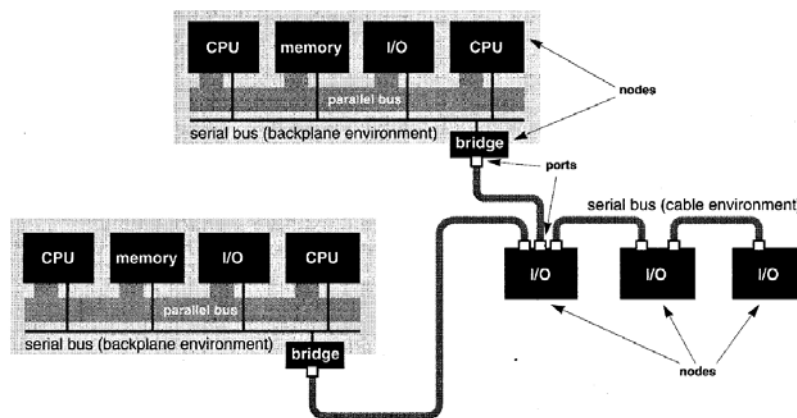


Figure Q.2—Serial bus physical topology

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.2)

#### Q.2.1 Cable environment

*The physical topology for the cable environment is a noncyclic network with finite branches and extent. "Noncyclic" means that closed loops are unsupported.*

#### Q.2.2 Backplane environment

*The physical topology of the backplane environment is a multidrop bus. The media consists of two single-ended conductors running the length of the backplane. Connectors distributed along the bus allow nodes to "plug into" the bus.*

The backplane implementation of 1394 has been used in applications in specialized telecommunications, space and industrial systems, often utilizing the two conductors reserved

for a serial bus in the VME, PCI and Futurebus standards. The rest of this document refers solely to the "cable environment".

As can be seen from the diagram above and previous bus examples, the shared communications line in the cable environment is "built-up" by many cables joined together through 1394 nodes that also act as signal repeaters.

### FireWire Speeds & Backwards Compatibility

Another topic that should be clarified here is the topic of transmission speed for FireWire. The code names for these speeds are shown in the table below:

<b>100 Mbps</b>	S100
<b>200 Mbps</b>	S200
<b>400 Mbps</b>	S400
<b>800 Mbps</b>	S800
<b>1600 Mbps</b>	S1600
<b>3200 Mbps</b>	S3200

For completeness it should be stated that these are "nominal" rates. S100 is not actually 100Mbps, but 98.304 Mbit/s<sup>4</sup>. This is almost 100Mbps so for marketing reasons it is called 100Mbps (98.304 is a very awkward number to pronounce or to remember).

In the 1394 standard, this 98.304 Mbps data rate is called the "Base Rate". All other data rates are **exact multiples** of the Base Rate.

So for example S800 is actually  $8 \times (\text{Base Rate}) = 786.432 \text{ Mbps}$ .

The S1600 and S3200 data rates are fully defined (electrical specification) and at the time of this writing and in the implementation stage.

The following points must be made clear:

- FireWire devices support multiple transmission speeds. They can transmit and receive data even at lower speeds than their *maximum capability*.
- Similarly, FireWire devices can repeat traffic which runs at lower speed than their *maximum capability*.

This is clearly done for reasons of backwards compatibility. For example, it should be possible to have a PC with an S800 host adapter control an S400 IIDC camera. Since the camera cannot "understand" higher speed traffic, all commands sent to it must be transmitted at S400. Similarly, any responses or image data from the camera will be at a S400 data rate, so the S800 host adapter must be able to receive them.

Note that in the points made above, the author mentions "maximum capability" instead of "maximum speed". There is a reason for that and this is yet another source of confusion over FireWire. This confusion is further extended by the S800 data rate that came about with the

---

<sup>4</sup> The dot '.' is used as a decimal separator in this document in accordance with the IEEE1394 standards.

advent of the 1394b-2002 standard. It is always correct to say "*a 1394b device*", but saying "*an S800 device*" can be technically incorrect sometimes as we shall see.

To start with, 1394a and 1394b are signaling modes, not speed tags. It is perfectly legal (and actual devices exist) to have a host adapter with three 1394b ports that can only transmit up to S400. So saying about a device that it is a "1394b device" technically only tells us that the device has 1394b ports and can use 1394b signaling (and 1394a too if the ports are bilingual).

Moreover it should be stressed that technically, 1394 nodes cannot be said to have a "maximum transmission speed", a "maximum data rate". The maximum data rate capability is a property of the FireWire ports, not the FireWire node.

Let's look again at a previous example, a host adapter with one 1394a port and two 1394b ports.



In the most likely case the 1394b ports are capable of the S800 data rate, but the 1394a port cannot achieve more than the S400 data rate, because higher data rates are only possible with 1394b signaling.

So it would be slightly incorrect to describe this device as an "S800 device", because some of its ports have lower capabilities. It would be better to describe it as "S800-capable device".

Note that the above is only possible with 1394b chips, because they are able to support different speeds per port and different port types. In most cases however, all the ports of a 1394b device have the same maximum speed, so in the terminology used with 1394b devices we might say "*that is an S800 1394b device*" without being incorrect. In any case we must remember that S800≠1394b; S800 and 1394b are not names for the same thing.

A question that naturally rises from the above is how exactly a 1394 node with an S400 and an S800 port repeat traffic?

To answer this we must start with the 1394 standard fact that the transmission speed of a packet is an electrical attribute and cannot change along the way. A packet that gets transmitted as an S100 packet will traverse the whole bus at that data rate, even while it crosses higher speed links between devices.



Naturally a transmission at a lower data rate can cross a link of a higher data rate. But what about a packet at a higher data rate that reaches a link of lower data rate?

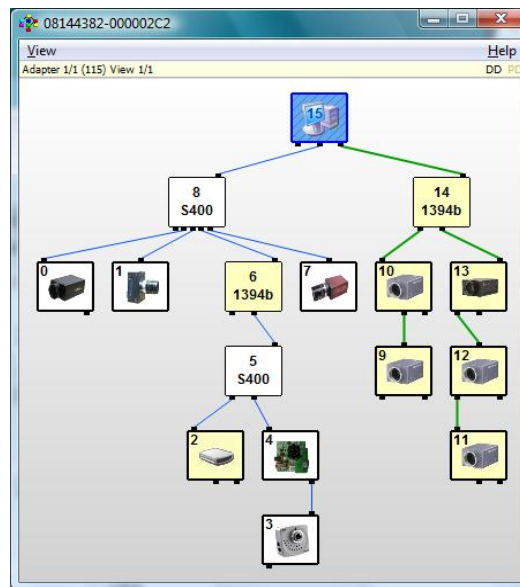
The answer of FireWire is elegantly simple:

*If a data signal cannot be repeated by some port, then it will not be repeated.*

This means that 1394 nodes don't always repeat all signals, but:

*All 1394 nodes always repeat all signals that they are capable of repeating to those ports that are capable of repeating them.*

Let us go back yet again to another of our previous examples, where we had a mixed 1394a and 1394b bus and see some of the implications of the above:



Green links indicate connections that use 1394b signaling and blue indicates connections that use 1394a signaling. Thin links indicate S400 speed and thick links indicate S800 speed. Yellow nodes are 1394b nodes (capable of 1394a and 1394b signaling) and white nodes are 1394a nodes (only capable of 1394a signaling).

Node 15 is drawn in blue to help us identify the local node (there might be multiple PCs on the bus), but it is a 1394b node. Why? Well, it has a 1394b connection to node 14, so it is capable of 1394b signaling, thus it is a 1394b node.

The connection between nodes 15 and 8 is a 1394a connection since node 8 is a 1394a node, most likely with a 6-to-9 cable from a 9-pin bilingual port, but could also be a 6-to-6 cable from a 6-pin port on node 15.

When the camera at node 10 sends a packet at S800, this packet will get repeated on the cable through nodes 9, 14, 13, 12, 11 and 15 but at that point it will not be repeated further. This means that if the camera of node 10 wanted to send a packet to the disk at node 2, and sends that packet at S800, then the packet will not reach its destination.

Technically speaking "Node 2 is unreachable at S800 from node 10".

If the camera at node 10 wants to communicate with node 2, it must send the data packet at the S400 data rate (or slower).

This is one of the major complexities with 1394 software. Using the topology information provided by the 1394 bus (the same information that the program above used to draw the 1394 bus topology) the software may calculate the so called "path-speed table" (or Speed Map). This is a table/matrix that shows the maximum data rate that any node can use to reach any other node. Alternatively, the software may "try & see", which means to make an attempt to communicate with each device at the maximum speed, and then successively the lower rates, until it succeeds at some rate.

Additionally, depending on the transmission speed, there are limitations on the maximum size of data packets. So if a node needs to send a "big" packet, a packet with a size that requires S800, to a node that is only reachable in S400, then sending this packet is impossible.

It is the responsibility of the designers of 1394-based systems to ensure that the system components (1394 nodes) are connected to each other in such a topology that permits operation without problems of this sort.

The reason that the designers of the 1394 protocol decided not to repeat higher rate traffic into lower rate links may not be immediately obvious.

Suppose that a node sends a small S800 packet, a 16-byte packet, to another node. The packet is small enough to fit in an S400 link, so why not "translate" (downgrade) this packet to S400 and "repeat" it over the S400 links too?

The answer is that "transmission rate" equals "transmission time". A 16-byte S800 packet takes  $16 \times 8 \text{b} / 800 \text{Mbps}$  to transmit (or cross an S800 link). If it is transmitted at S400 it takes  $16 \times 8 \text{b} / 400 \text{Mbps}$ , which is easy to see that is exactly the double amount of time.

If this was permitted, then the same packet would take different amounts of time to propagate on different portions of the bus, which would make timing sensitive operations like bus arbitration exceedingly complex.

So the designers of FireWire preferred the simpler electrical solution (same transmission rate everywhere) over the software complexity of calculating path-speed tables. And in any case, arbitrary arrangements of FireWire devices only occur in end-user scenarios and usually end users don't have that many devices. In complex industrial systems, the system designers set up the 1394 bus in such ways as to minimize or eliminate such issues.



## 1394 Addressing Model

From 1394-2008 annex Q3:

### ***Q.3 Addressing***

*The serial bus follows the CSR architecture for 64-bit fixed addressing, where the upper 16 bits of each address represent the node\_ID. This provides address space for up to 64 k nodes.*

*The serial bus divides the node\_ID into two smaller fields: the higher order 10 bits specify a bus\_ID and lower order 6 bits specify a physical\_ID. Each of the fields reserves the value of all "1"s for special purposes, so this addressing scheme provides for 1023 buses, each with 63 independently addressable nodes.*

FireWire follows the command and status register (CSR) architecture of IEEE Standard 1212-2001. According to this standard, the address space of the bus is a 64-bit memory space. Doing actions on this bus consists (conceptually) of accessing "memory locations" in this 64-bit address space. The memory address specified somehow translates (maps) initially to one of the nodes in the bus and then to some CSR register "inside" that node.

So, this 64-bit address identifies both the node and some "logical unit" inside the node (the unit that will handle the requested action on this memory address). This implies that the 64-bit memory address is partitioned in two parts, the first being something akin to the node's "external address" (how it is known to the rest of the bus) and the second being an "internal address" which only has meaning "inside" the node.

This concept is very similar to the TCP/IP protocol: The *IP address* is the "external address" while the TCP/UDP port is the "internal address", which identifies which of all the software modules inside the PC should get hold of an incoming packet.

FireWire uses the high 16 bits of the 64-bit address as the "external address", while the rest 48 bits of the address identify the "internal address" within the node.

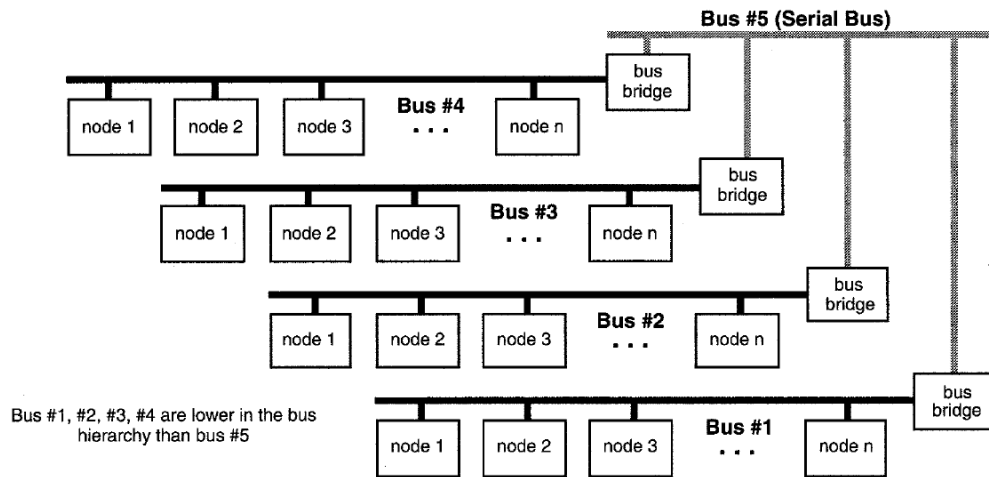
The "external address" of a 1394 node on the 1394 bus is called the **Node ID**. The Node ID is 16 bits in size and is further divided in two parts, the 10-bit **Bus ID** and the 6-bit **Physical ID**.

The 10 bits for the Bus ID provide for 1024 possible values. One of those values (all 1s, 1023, 0x3FF) is reserved as having a special meaning. It identifies the "*Local 1394 Bus*" and is also called the "*Local Bus ID*".

To understand the meaning of a "Bus ID" we must briefly mention 1394 bridges. The IEEE 1394.1 standard provides support for "1394 bridges", that is devices that can help two or more independent 1394 buses communicate with each other. In such a scenario, these buses would have different Bus IDs. Traffic that uses the local Bus ID is intended for the "local" bus and thus ignored by the 1394 bridge. For all traffic that does not have the Local Bus ID, a bridge is supposed to examine the Bus ID value and forward (retransmit) the packet to the destination 1394 bus (or towards the destination bus).

This is packet forwarding, not packet repeating. The bridge is supposed to receive the packet into a buffer, decide which bus to forward it to, then arbitrate on that bus and retransmit the packet. So, the two buses are totally independent of one another.

Theoretically one can build a whole complex network of 1394 buses interconnected by 1394 bridges as shown in the diagram below:



(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 1.1)

Since such big topologies with multiple buses are possible, it is only reasonable to expect that each bus should have its own ID, so that the bus bridges can forward traffic as appropriate. Hence the 10-bit Bus ID within the 16-bit Node ID.

In practice there are no 1394 bridges at this time, as the attempt to create a working standard for their operation has proven exceedingly complex and at the same time the "market" could do just fine without bus bridges. As a result a FireWire bridge standard exists but nobody paid for a bridge device to be made.

As a result, in 99.99...9% of the cases, all the traffic on FireWire buses uses the local Bus ID. There are some interesting tricks that can be performed by changing the Bus ID of some nodes on the bus (but not all nodes) which accounts for the remaining 0.00...01%.

The 6-bit physical ID is a far more interesting topic. 6 bits yield 64 possible values, of which the value 63 (all ones) is reserved to mean "broadcast".

However, the interesting fact about physical IDs is that they do not remain constant over time. As the bus is reconfigured (devices plugged in or unplugged) the physical IDs get updated accordingly.

This "Plug'n'Play" trick is at the same time one of the major strengths and troubles of FireWire. You can plug any device anywhere on the 1394 bus, and the bus will reconfigure automatically (create a new tree topology) and assign physical IDs to devices. No human intervention needed!!!

That is very sweet at the hardware level, however at the software level it is not so easy to deal with devices whose physical address might change any time.

More details on this automatic bus reconfiguration will be given later in this document.

The following diagram from the 1394-2008 standard describes the 1394 memory model addressing in a nutshell, while also showing the further partitioning of the internal 48-bit address space into "initial memory space", "private space" and "register space":

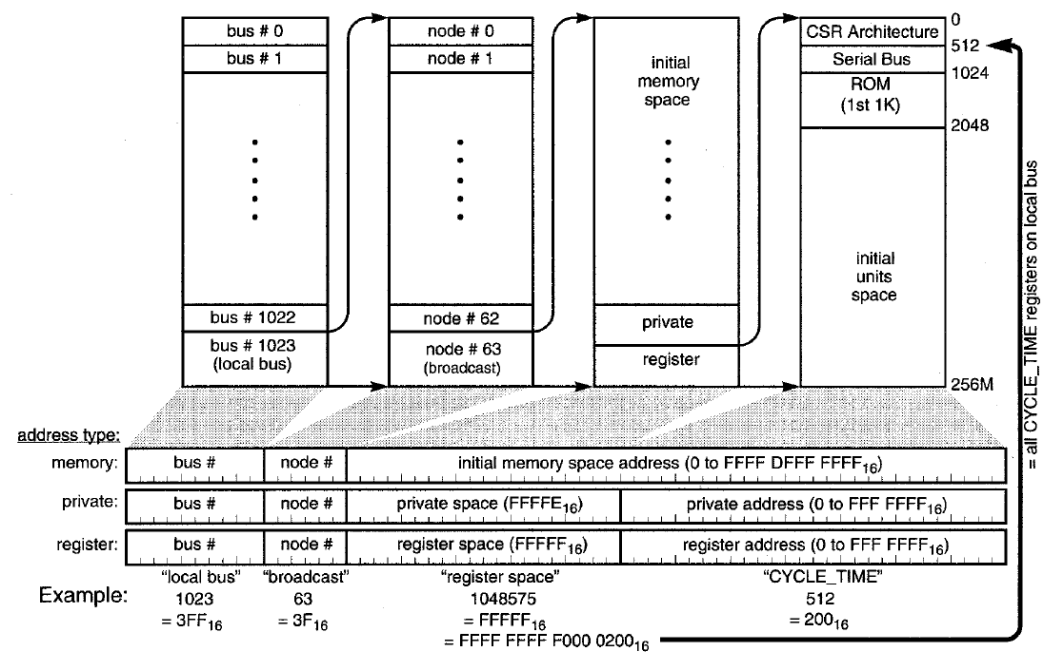


Figure Q.3—Serial bus addressing

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.3)

### Size Notation & Endianness

The following are referenced from 1394-2008 paragraph 1.5.3:

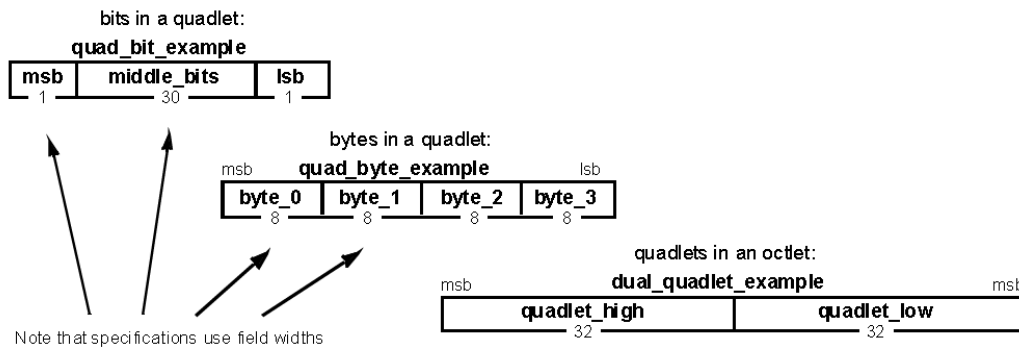
Size (in bits)	16-bit word notation	32-bit word notation	IEEE standard notation (used in this standard)
8	byte	byte	byte
16	word	half-word	doublet
32	long-word	word	quadlet
64	quad-word	double	octlet

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Table 1.1)

The terms in the last column are the ones used by the 1394 standard and the ones that will be used in this document further on.

A related very important issue has to do with byte Endianness (little endian versus big endian):

*The serial bus uses big-endian ordering for byte addresses within a quadlet and quadlet addresses within an octlet. For 32-bit quadlet registers, byte 0 is always the most significant byte of the register. For a 64-bit quadlet-register pair, the first quadlet is always the most significant. The field on the left (most significant) is transmitted first; within a field, the most significant bit (MSB), i.e., the leftmost, is also transmitted first. This ordering convention is illustrated in Figure 1-3.*



**Figure 1-3—Bit and byte ordering**

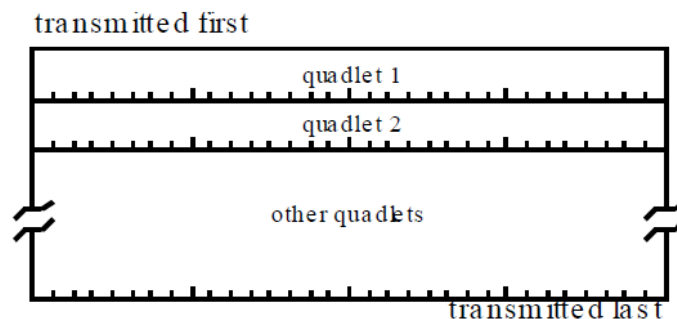
*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 1.3)*

The big endian convention is basically a "byte numbering" convention so that all nodes on the 1394 bus can have the same interpretation of the data packets that are sent or received. Internally a node may use a little endian processor; however the data packets that this node transmits must follow the big endian standard so that all other nodes can understand them (interoperability).

In the same vein, the 1394 standard in paragraph 1.5.5 defines a "data packet" as follows:

*Serial bus packets consist of a sequence of quadlets.*

Of course there are many different specific types of packets, but according to this definition any 1394 data packet is a sequence of quadlets as show below, which means of course that its size is always a multiple of 4 bytes:



**Figure 1-4—Sample packet format**

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 1.4)*

## Bus Reset & 1394 Bus Self-Configuration

The "Bus Reset" is such a fundamental feature of the 1394 bus that a concise definition for it is not included in the standard's glossary of terms.

So the author will provide a non-official definition:

*A bus reset is an event that interrupts the normal operation of the 1394 bus and forces all nodes into a special state that clears the previous topology information and starts a new phase of bus configuration. Once bus configuration has completed, normal bus operations can resume.*

Technically there are two types of bus resets:

1. **Long Bus Reset:** Takes longer to complete the bus reset phase (before reconfiguration starts) and might corrupt data packets.
2. **Short Bus Reset:** This is also known as "arbitrated bus reset" because the initiator arbitrates for the bus (as if to transmit a packet) and then initiates the bus reset. It is much faster and does not corrupt any data packets because by definition no one else was transmitting at that time.

It is easy to figure out that long bus resets happen when a node is added to or removed from the 1394 bus. Obviously a device that has not yet been added to the bus cannot arbitrate and request a short bus reset. The same goes for device removal; the device cannot possibly know that an outside agent (usually a human) is about to unplug it. It is the physical unplugging itself that causes the bus reset.

Short bus resets are usually used by software modules that want to reconfigure the bus in special ways (e.g. establish a new root). Whenever possible software should always initiate a short bus reset and avoid initiating long bus resets.

According to 1394-2008 annex H the bus configuration timeline after a bus reset is as follows:

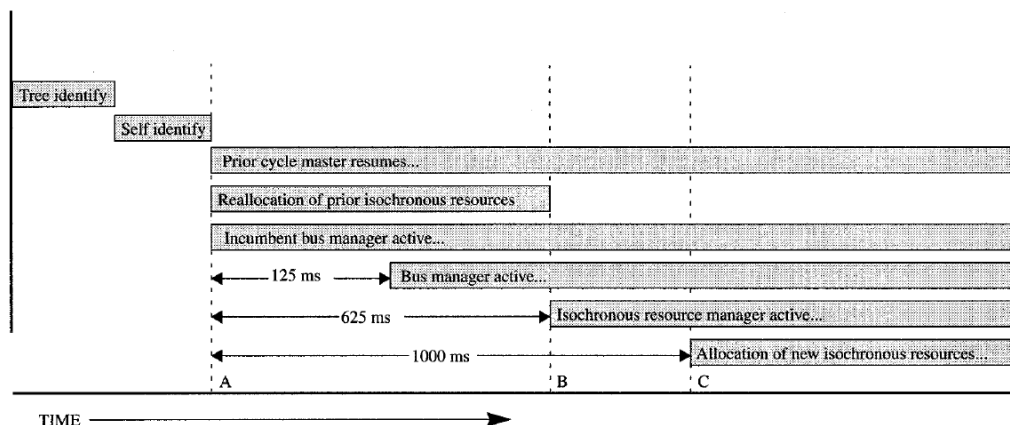


Figure H.1—Bus configuration timeline

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure H.1)

Before the "Tree Identify" phase, there is another brief phase (not shown) called the "Bus Initialization" phase. In that phase every node determines the connection status of its ports.

The "Tree Identify" phase is described by the 1394-2008 standard as follows:

*After a bus initialization process, the tree identify process translates the general network topology into a tree, where one node is designated a root and all of the physical connections have a direction associated with them pointing towards the root node. The direction is set by labeling each connected port as a "parent" (connected to a node closer to the root) or "child" port (connected to a node further from the root).*

That means that the "Tree Identify" phase is where nodes are organized into a hierarchy with the root node at the top (a node only with children).

The "Self Identify" phase is the phase where the physical IDs are assigned to each node in a deterministic manner. This means that once the tree has been identified, there is only one way to assign the physical IDs.

We can roughly think that the on a given bus the physical ID assignments depend on the root. If node A is the root on this bus, then there is only one way to assign the physical IDs. If the root is changed to node B then there will be a different way, but if node A is restored as root (through software) then the original assignment will come into place. As long as the physical topology is not changed (no nodes are added or removed), if node A is the root then the same physical ID assignment will be performed.

Technically the "Self Identify" phase is separate from "Tree Identify" because a new process (algorithm) is executed to complete this deterministic assignment of physical ID.

After this phase, normal bus operations can resume (bus arbitration and packet transmission). Parts of these operations are also depicted in the diagram above, however these tasks involve normal traffic. The "Tree Identify" and "Self Identify" phases are performed through special signal exchanges.

So, to sum things up:

- A bus reset occurs whenever there is a change in the physical topology of the 1394 bus (1394 nodes get connected or disconnected) or software decides to logically reconfigure the topology of the existing bus (new root, new gap\_count, clear bus error, etc).
- The bus reset interrupts the normal operation of the bus and forces every node to enter a "special" reconfiguration mode.
- Immediately after the bus reset follows the "Bus Initialization" phase, where every device determines again which of its ports are connected and which are not.
- Then comes the "Tree Identify" stage where the parent-child relationships are determined, building a hierarchical bus topology. At the end of this phase all connected ports know whether they are connected to a "child node" or a "parent node". The "root node" is also determined and is the only node whose ports are only connected to "child nodes".
- Next is the "Self Identify" phase where the physical IDs are assigned to each node.
- Normal bus operation (data packet exchange) can now commence again.

A little detail that should also be made clear is that technically saying "*connect a device to the bus*" or "*remove a device from the bus*" may be inaccurate. You can also connect another 1394 bus segment to the bus (add a whole bunch of devices already connected to one another), or remove a whole segment from the bus (remove a bunch of devices at the same time). Both of these events will generate only one bus reset, not one bus reset per connected or disconnected device.

Technically a "topology change" means:

*The connection state of any port of the 1394 bus has changed.*

It is such a "port connection state" change that triggers a bus reset.

For example it is possible through software to "disable" a connected 1394 port. This means that the port is still "physically connected" (the cable was not removed) but "logically disconnected" (it broke the electrical connection with the port at the other end of the cable). Such an event is a "port connection state" change, so a bus reset will be initiated by 1394.

The distinction between long and short bus resets is not the only one. Bus resets are also characterized as "hard" and "soft":

- **Hard Bus Reset:** Initiated by a change in port connection state. All hard bus resets are long bus resets.
- **Soft Bus Reset:** Initiated by software (for some reason). Can be short or long.

Hard Bus Resets are easy to understand. But, why are Soft Bus Resets needed?

Soft Bus Resets exist for a number of reasons. The most prominent of those reasons is so that software can designate which 1394 node is going to be the root node in the bus topology.

In the normal operation of the post Bus-Reset phases, the node that will end up being root is selected non-deterministically. To put it in a cruder manner, the root node is selected randomly. This does not mean that a random process is followed. It means that when a bus reset happens (a) all nodes have equal probability of becoming the root node and (b) there is practically no way to guess who will actually become the root.

However, for reasons described in following chapters, software modules may desire that a specific node becomes the root. So 1394 provides a way to change the "equal probability" of all nodes and favor a specific node. This is done by transmitting a special data packet. This data packet informs the target node that when the next bus reset happens it should "bend the rules". It also informs all other nodes that somebody else is going to bend the rules so they better behave "by the rules".

The nodes get "informed" but for the topology to actually change a bus reset has to happen. So the software initiates a Soft Bus Reset and the task of assigning a specific node as root gets completed.



## Beta Loops & Redundancy

It has been mentioned earlier that the 1394 bus topology is a hierarchical tree with no circles (loops). With 1394a-only buses (buses that contain only 1394a nodes) this is true at the physical level. You cannot follow the cables around and end up where you started.

If a loop was introduced in a 1394a-only bus then the algorithm of the "Tree Identify" phase would fall into an infinite loop, eventually timeout and generate a new bus reset. This would repeat, causing an infinite sequence of bus resets until someone physically removed the loop.

The 1394b standard added a very powerful capability to FireWire. Now the presence of physical loops is permitted! What happens is that during the "Bus Initialization" phase, the 1394b nodes on the bus detect the presence of the loops and logically "break" them. To logically break a loop essentially means to "logically remove" one of the links that are part of the loop; a link whose removal will "break" the loop.

The details of how the loops are detected and how the breaking link is selected are outside the scope of this document. However the "breaking" of the link is not. Earlier we mentioned that it is possible for software to disable a port and thus make it be logically disconnected, although the actual cable is still connected.

Something similar is taking place in this case too. The 1394 bus automatically disables the ports on the selected link so as to break the loop. No software intervention is required. It all happens automatically in the special processing phases after the bus reset. This kind of port disabling is called a "Loop-Disable".

What is even more amazing is the fact that multiple loops can be present at the same time. 1394 carefully selects and automatically breaks all appropriate links, so that the bus (a) has no more loops and (b) all devices remain accessible.

All these are interesting, but the question is "why should we allow loops at all"? Is it so that if a human error is committed in the configuration, we won't have the "infinite bus reset sequence" problem?

It turns out that the reason for this feature is "redundancy".

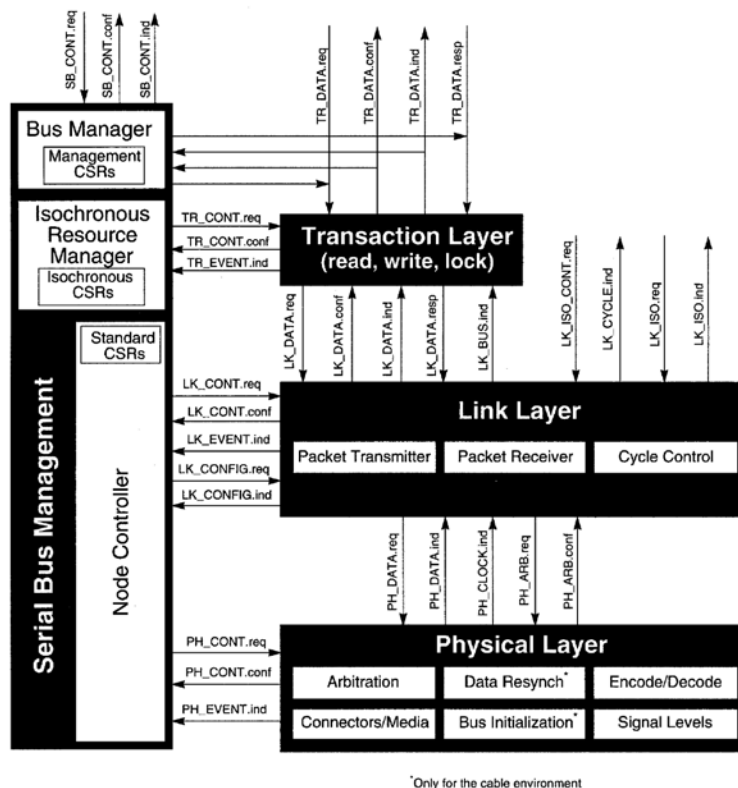
The reason that system designers want to have loops in a 1394 bus is their desire to have more than one physical "cable paths" between some devices, so if there is a hardware problem on one cable path then the devices will still be accessible through the alternate cable path(s).

1394 does some extra magic here. Whenever a bus reset occurs it recalculates all the loops and if there has been a connectivity change that will "break" the bus in two pieces, then it automatically reactivates the appropriate loop-disabled ports so as to bring the bus back into one piece.

This way 1394b offers redundancy, which is a significant advantage compared to other competitive technologies.

## 1394 Protocol Layering

The 1394 protocol is a protocol for data transmission, so naturally it has been defined in a layered fashion (a stack of layers). The diagram below comes from the 1394-2008 standard and shows all the defined layers of the "Serial Bus Protocol" (SBP) stack:



(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.4)

Actually the "Serial Bus Management" layer sits on the top of the hierarchy, but because it uses services from all layers below it, it is drawn on the side.

Of the layers shown above, the Physical and the Link Layer are usually implemented in hardware and referred to as the "PHY chip" (or simply the PHY) and the "Link chip" (or simply the LINK).

In most implementations until a few years ago, the PHY and the LINK were implemented as separate chips on the boards, but in the recent years several chip makers have come up with integrated PHY+LINK chips that include both functionalities in one piece of silicon. However, this is an implementation detail. The PHY and the LINK perform totally different functions, so they are being discussed as different units. For example, it is the PHY that controls the ports of a 1394 node, determines port connection states, makes arbitration requests, repeats all traffic, etc.

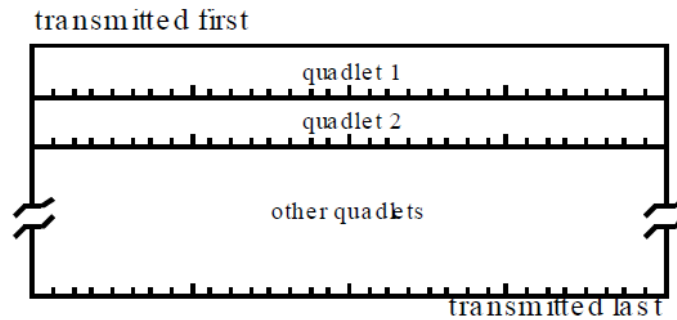
The "Transaction Layer" and the "Serial Bus Management" (SBM) layer are mostly implemented in software (or firmware) inside the 1394 nodes that support them. Not all nodes require all layers. For example a repeater only has a PHY but no LINK. Devices like cameras also have a LINK and Transaction layers, but usually no SBM layer. The SBM layer is usually implemented in computers (PCs).

## 1394 Bus Packets

We have already stated the basic definition of a 1394 data packet in an earlier paragraph:

*Serial bus packets consist of a sequence of quadlets.*

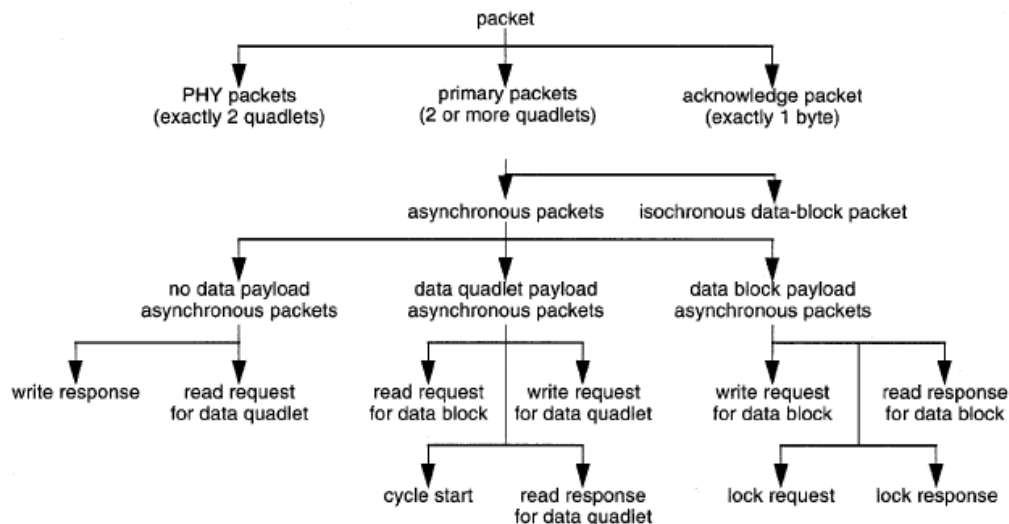
Any 1394 data packet is a sequence of quadlets as show below, which means of course that its size is always a multiple of 4 bytes:



**Figure 1-4—Sample packet format**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 1-4)

However, there are many different types of data packets, depending on their characteristics and purpose. The full hierarchy is shown below in a diagram from the 1394-2008 standard:



**Figure 6-1—Serial bus packets**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-1)

At the first level of the hierarchy there are three types of packets:

#### 1. PHY Packets

These packets are exactly 2 quadlets (8 bytes) in size and are not acknowledged by recipients. As their name implies, they were originally intended for sending information, sending commands and getting responses to and from the PHY chips of the nodes on the 1394 bus. In these packets the two quadlets are not independent of one another. The second quadlet is the binary inverse of the first (acting like a checksum).

However, PHY packets were later "overloaded", which means that they were put to a new use. The VersaPHY protocol defines a new subset of PHY packets by using

some reserved fields. In these packets both quadlets contain information (so they are not the inverse of one another).

2. Acknowledge Packet

According to 1394-2008 paragraph 3.1.5 an acknowledge packet is "*An 8-bit packet that may be transmitted in response to the receipt of a primary packet*".

3. Primary Packet

According to 1394-2008 paragraph 6.2.1 "*A primary packet is distinguished from an acknowledge packet or a PHY packet by its length and/or encoding. The length of a primary packet shall be at least two quadlets (a zero-data isochronous packet). A primary packet shall consist of a packet header, and it may also include a data block*".

Granted to say that the definition of the primary packet given above is a little obscure, so it would serve quite as well to think of primary packets as "*any packet that is not a PHY packet or an acknowledge*".

Also note that in the definition of acknowledge packets, they are "optional". This means that not every type of primary packet gets an acknowledge when it gets received.

Once we are past the first level categorization of packet types we can move to the second level, which distinguishes primary packets between "asynchronous packets" and "isochronous packets".

The 1394-2008 standard states the following in paragraph 6.2.1:

*Two basic varieties of primary packet are defined for the serial bus: the asynchronous packet and the isochronous packet, which are distinguishable by the transaction code value.*

This simple statement of course does not reveal all the complexity, but tells us one important thing:

*From a "data" point of view asynchronous and isochronous packets are the same thing: a 1394 data packet with a header and possibly a body. Isochronous packets just happen to use some specific transaction code value (10) in their headers, while asynchronous packets use other values.*

This means that there is nothing intrinsically remarkable about isochronous data packets. It is the special way in which they get handled that makes "isochronous traffic" such a special feature of FireWire. That will be explained in detail in later paragraphs.

Before we move further we must step back one moment and clear out the terminology. In computer software literature the term "**asynchronous**" appears very often, together with its opposite term which is "**synchronous**". However FireWire uses the terms "**asynchronous**" as opposed to "**isochronous**" to distinguish the two basic types of its primary packets.

So what is the difference between "*asynchronous*", "*synchronous*" and "*isochronous*"? Although the difference may be apparent to a native Greek speaker (all these words originate from the Greek words "ασύγχρονο", "σύγχρονο", "ισόχρονο"), or to someone who has been immersed for years in the details of FireWire, the author has noticed some considerable confusion over them in the past in people that are not so familiar with FireWire.

Let us start with the definitions of "*Asynchronous*" versus "*Synchronous*" in computer literature, which is basically a characterization for I/O operations:

- Asynchronous I/O (or non-blocking I/O) is a form of input/output processing that permits other processing to continue before the I/O operation has finished.
- Synchronous I/O (or blocking I/O) is a form of input/output processing where the caller waits for the operation to finish before proceeding with further processing.

The Greek word "*Asynchronous*" is used to describe "*two or more things that are independently happening (evolving) at the same time, without paying any attention to one another*", which is the reason why they are considered independent.

I order some pizza and continue doing my work, while the pizza is being prepared, cooked and finally delivered to me. My work and the pizza preparation are "asynchronous" processes.

In that sense the term "Asynchronous I/O" is a valid use of the language word "Asynchronous". The program kicks off some I/O operation, does other things in the meanwhile, and at some moment picks up the completed I/O operation.

However the Greek word "*Synchronous*" is used to describe "*things that happen in the same time in a fully time dependent manner*".

When a ballet is giving a performance the dancers are performing in a synchronous manner. Some of them do exactly the same moves at exactly the same times, while others that run around and do "solo" moves are fully aware of the time dependencies and know for example that they have to finish some part of their move in 5 beats of the music playing in the background. All the dancers are "synchronized" (they better be), they act in a "synchronous" fashion.

So to call "Blocking I/O" as "Synchronous I/O" is a major misnomer (to give the wrong name to something). To wait until something happens does not make you "synchronous" with that thing. To wait until something happens does not mean that you sit idle doing nothing in the meanwhile. In the asynchronous I/O case the program also "waits" for the I/O to finish (it cannot draw the image onscreen until the image data has been read from the file). The difference is that it does something else in the meanwhile instead of sitting idle.

Very few things in 1394 technology are "synchronous" and they all have to do with the low level operation of the chips and the cable signals that necessarily have to be synchronized. To put it more emphatically:

*There is no such thing as "synchronous" data traffic in FireWire.*

As we saw above, to describe some process as *synchronous* or *asynchronous* there must be more than one elements involved in a time relationship. Something may be synchronous or asynchronous with regards to something else.

Thus the 1394 term of "*Asynchronous Packet*" does not mean that the packet by itself is "asynchronous" but rather that the packet is of the "*Asynchronous Traffic Type*". Similarly an "Isochronous Packet" indicates a packet of the "*Isochronous Traffic Type*".

So, in the 1394 world, "Asynchronous" is the term selected for "*other than Isochronous*". The characteristics of Asynchronous and Isochronous traffic are described in the next paragraphs.

### The 1394 Cycle - Part 1

Let us now see the definition of the term "*Isochronous*" provided by the 1394-2008 standard in paragraph 3.1.65:

*Uniform in time (i.e. having equal duration) and recurring at regular intervals.*

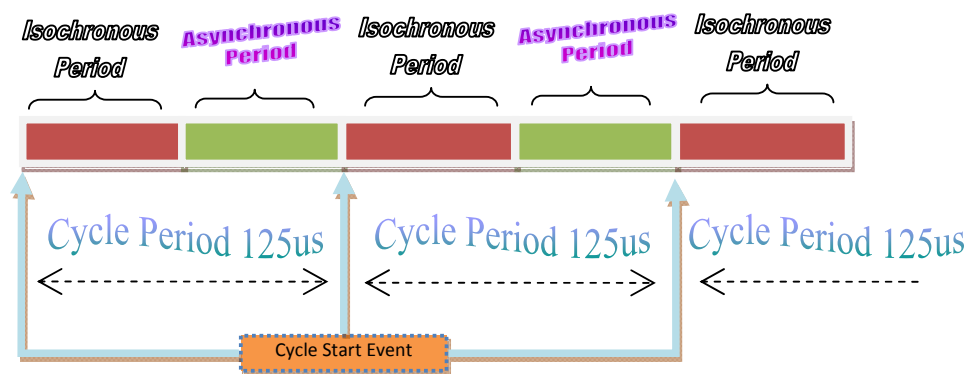
This definition is extremely accurate with regards to the meaning of the respective Greek word. In 1394 too, isochronous traffic is traffic that occurs at regular time intervals and (usually) has equal duration (within each interval).

Now we are better prepared for the more technical descriptions of the 1394 standard:

- **3.1.15 asynchronous packet:** A primary packet transmitted in accordance with asynchronous arbitration rules (outside of the isochronous period).
- **3.1.66 isochronous interval:** A period that begins after a cycle start packet is sent and ends with a subaction indication. During an isochronous interval, only isochronous subactions may occur. An isochronous interval begins, on average, every 125 microseconds.
- **3.1.68 isochronous subaction:** Within the isochronous interval, either a concatenated packet or a packet and the gap that preceded it.

What the above paragraphs try to explain is that there is something called "isochronous interval" or "isochronous period". Anything transmitted outside of this time period is an asynchronous packet. Anything transmitted during this period is an isochronous packet.

We can picture this as the 1394 bus alternating between the isochronous and asynchronous period once every 125 microseconds, as shown below:



The isochronous period ends with some mysterious thing called a "*Subaction Indication*" but we don't need to analyze this further. When the isochronous period ends, the asynchronous period starts. Then, approximately 125 microseconds after the beginning of the previous isochronous period a "Cycle Start Event" will happen (in the form of a Cycle Start Packet) and a new isochronous period will begin.

To be precise, the 1394 protocol rules in essence say that isochronous packets can **only** be transmitted inside the isochronous period. When the isochronous period ends, then

asynchronous packets may be transmitted, until the 125 microseconds elapse and the next "cycle" begins (the next isochronous period starts).

For example, you cannot take an asynchronous packet (transaction code value different than 10) and force it to get transmitted during the isochronous period. The 1394 chip will refuse to transmit it in that period and instead wait for the next asynchronous period to start.

The diagram also illustrates one of the fundamental notions of FireWire: the so called "1394 Cycle", which is a duration of 125 microseconds on average. Note that the "1394 cycle" is not the same thing as the "isochronous interval" (or "isochronous period"). Instead:

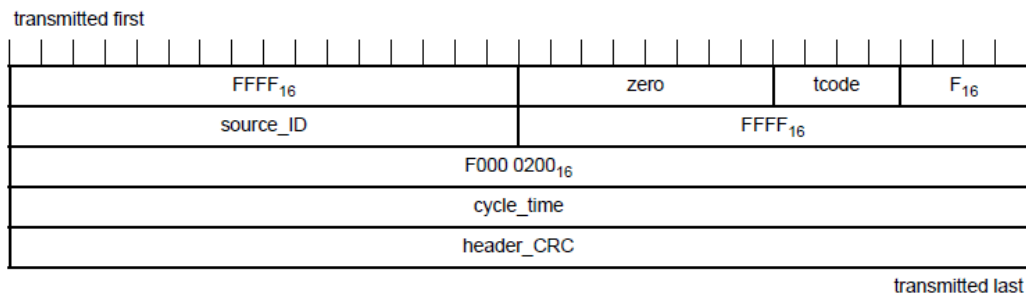
$$(Isochronous\ Period) + (Asynchronous\ Period) = "1394\ Cycle" \approx 125\ microseconds$$

A 1394 cycle starts with a "Cycle Start Event" which appears on the 1394 bus as a data packet that is called the "Cycle Start Packet". The 1394 cycle ends when the next cycle start packet gets transmitted and a new 1394 cycle starts.

### The 1394 Cycle - Part 2: Cycle Start Packets & the Cycle Master

A closer examination of diagram 6.1 (shown several pages earlier) shows that the Cycle Start packet is a primary packet and specifically one of the asynchronous type. The fact that the "Cycle Start" is a normal data packet is very important in understanding the way 1394 isochronous and asynchronous traffic works. The Cycle Start packet is a normal data packet and it gets transmitted like all other data packets. It is not, for example, any special electrical signal that acts as an "electronic clock".

The following diagram from the 1394-2008 standard shows the layout of a Cycle Start packet which is composed of five quadlets:



**Figure 6-10—Cycle start packet format**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-10)

The **tcode** field shall have a value of 8. The **source\_ID** field shall be the Node ID of the node that transmits the Cycle Start packet. The **cycle\_time** field shall contain the contents of the cycle master's CYCLE\_TIME register.

Since the Cycle Start packet is a data packet, then one of the 1394 nodes on the bus must have transmitted it. In 1394 terminology, the node that transmits the Cycle Start packets is called the "Cycle Master". There can only be one Cycle Master at any point in time. Only one node is sending the Cycle Start packets.



The Cycle Master node is determined immediately after a bus reset and it remains in that "role" until the next bus reset. At that point a new Cycle Master might be selected, which will be the one transmitting the Cycle Start packets until the next bus reset, etc.

The following fact is worth stressing out:

*Transmission of the Cycle Start packets is optional.*

A software module can request from the Cycle Master to cease the transmission of the Cycle Start packets. What happens then?

If we trace back to the definitions given earlier, the Isochronous Interval starts with a Cycle Start packet and inside a cycle there is the Isochronous Period and the Asynchronous Period. If there is no Cycle Start packet then there is no Isochronous Period, there are no cycles, and the 1394 bus operates in the asynchronous period all the time.

Not all systems/applications require data traffic of the isochronous type, so it is quite possible to "turn off" the Cycle Start packets and only use asynchronous traffic.

Turning off Cycle Start packets in such cases also yields a performance benefit in the order of 5% which can look quite surprising if we look at the details! The Cycle Start packets are 20 bytes each, transmitted once every 125 microseconds (8000 per second) which means that the percentage of bus bandwidth taken by them is:

$$((20 \times 8000 \times 8) \text{ bits/sec}) / 800\text{Mbps} = 0.15\%$$

It is beyond the scope of this document to explain how the removal of this 0.15% of traffic can yield a 5% performance gain, but suffice it to say that some applications need that extra gain, so it is not unreasonable or uncommon to turn off Cycle Start packets.

We stated earlier that after the bus reset the Cycle Master *has to be determined* (decided upon). Why is this necessary? Are there different "skill levels" in candidate Cycle Masters?

The answer is yes, there are two skill levels: A node either can or cannot act as a Cycle Master. Not all 1394 nodes are capable of generating the Cycle Start packets so such nodes cannot possibly act as Cycle Masters. Generating the Cycle Start packets requires some extra circuitry and possibly firmware support, so some devices choose to not provide this capability and let some other 1394 node take on the Cycle Master task.

*"Cycle Master Capable": A 1394 node that is capable of generating Cycle Start packets.*

Additionally, only the root node in the topology can perform the duty of the Cycle Master. If the root node in a 1394 bus is Cycle Master Capable, then the bus configuration is called "*Isochronous Capable*". The exact same bus with another root node that is not Cycle Master Capable will not be *Isochronous Capable*.

In contrast a single 1394 node is called "*Isochronous Capable*" if it can transmit isochronous traffic. This means that to have isochronous traffic on a 1394 bus, we need:

1. Isochronous Capable nodes on the bus.
2. The bus configured with a Cycle Master Capable node as the root node.
3. Transmission of Cycle Start packets must be enabled on the Cycle Master.

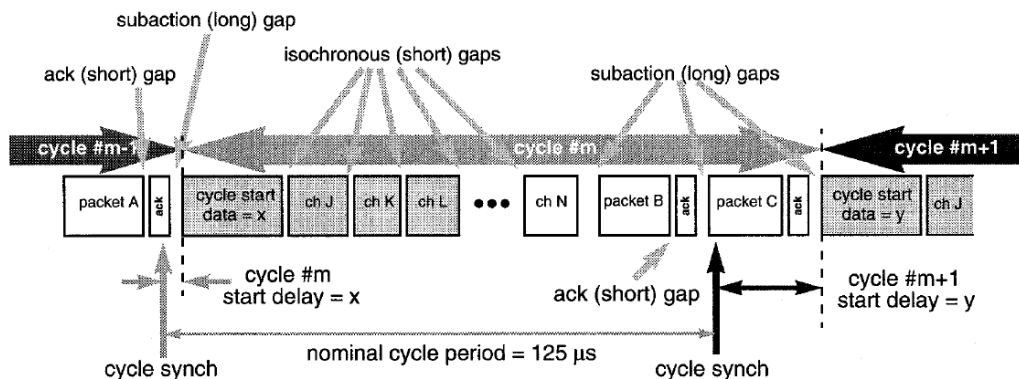
As described in earlier paragraphs, if the software wants to work with isochronous traffic and it detects that the current root is not Cycle Master Capable, then it will locate one Cycle Master Capable node, designate it as the next root and initiate a short bus reset.

In practice, most applications require isochronous traffic, so 1394 software usually manipulates the topology of the 1394 bus (using short bus resets) so that a "Cycle Master Capable" node ends up as the root node in the topology.

All 1394 host adapters are Cycle Master Capable, which means that the "PC device" is Cycle Master Capable, so in most 1394 systems a PC is the root device (namely its 1394 host adapter). However, this is just a convenience and a practical coincidence. There is no rule that says a PC must be the root of the 1394 bus in order to have isochronous traffic.

### The 1394 Cycle - Part 3: Cycle Structure & Cycle Drift

After having introduced all these terms and rules of the 1394 standard, it is time to examine one interesting diagram from the 1394-2008 standard:



**Figure Q.16—Cycle structure**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.16)

This diagram possibly contains way too much information for a single drawing, but we will try to explain it piece by piece.

The first thing that we should pay attention to is the "cycle synch" arrows. Internally every Cycle Master node has a very accurate clock that generates a "cycle synch" signal 8000 times every second (once every 125 microseconds). This indicates to the 1394 chips on the node that they must transmit a Cycle Start packet.

But we already said that the Cycle Start packet is a normal packet. What happens if some other primary packet is getting transmitted at that time? In that case the Cycle Start packet gets delayed. This is the reason why all previous definitions about the cycle duration states things like "approximately" or "on average" together with the 125 microsecond duration.

The way that the 1394 protocol works at lower level makes the Cycle Start packet a kind of "high priority packet". This means that as soon as the transmission of the current packet ends then no other packet will get transmitted, but the "awaiting" Cycle Start packet will take priority instead.

In the diagram above we can see that in both cycle synch events, there were small delays in the transmission of the Cycle Start packet. If this fact was left untreated, then this "logical" clock of the 1394 bus would slowly drift. For example after 24 hours of operation one would expect the number of cycles to be (24h/125usec) but we would find that there actually were much fewer cycles in practice.

This accumulating "cycle drift" would be a serious problem for many applications, so the 1394 standard does some extra magic here. The 1394 chips in the Cycle Master keep track of these mini-drifts and "correct them" very quickly and very naturally as we will see.

This is done by using the Cycle Synch events as the compass. The Cycle Synch events come from a real hardware clock and they never drift. Whenever the Cycle Synch event occurs, it practically tells the 1394 chip "Transmit a Cycle Start packet as soon as possible".

Let us start from time point zero, and suppose the first cycle takes 145 microseconds because the second Cycle Start packet was delayed for 20 microseconds by traffic. The next cycle synch will still occur at time point 250 microseconds. Suppose again that there was traffic and the Cycle Start got delayed 5 microseconds, so it got transmitted at time point 255 microseconds.  $(255-145)=110$ , so the second cycle lasted only 110 microseconds instead of 125 and a 15 microsecond correction was made (of the original 20). The next cycle synch is at 375 microseconds.  $(375-255) = 120$ , which means that if there is no traffic the remaining 5 microseconds will be adjusted for.

If you think about the above, they mean that no matter how heavy the traffic, no matter how long the system runs, each and every Cycle Start packet will be at worse within a 40-50 microseconds time distance from where it should be. There is no "*delay accumulation*" and thus no "*clock drift*". Some cycles might be a little longer than 125 microseconds, some a little shorter, but on average the cycles last 125 microseconds and each and every second has 8000 cycles on average.

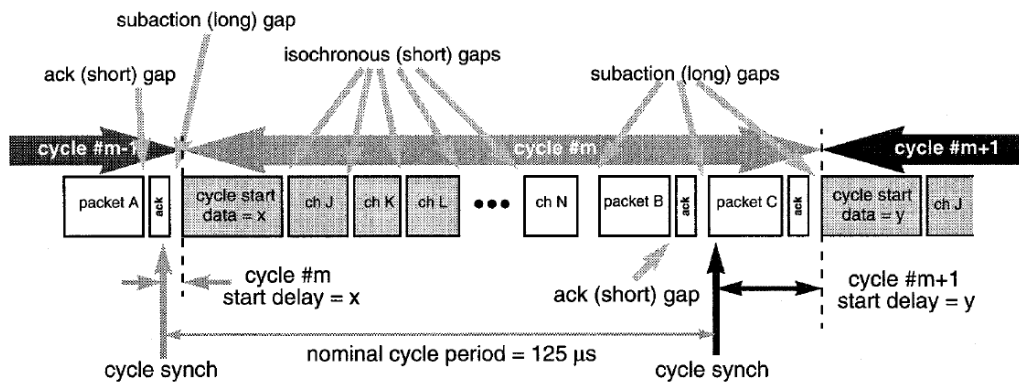
To see why this is true, consider the following "data":

- If I monitor the 1394 bus for exactly 24 hours after the first Cycle Start packet, how many cycle packets will I see? The answer is either  $(24*3600*8000)$  or  $(24*3600*8000)-1$ . Maybe the very last cycle was a little off, so my observation registered one packet less.  
Dividing either number with  $24*3600$  the results are very close to 8000 (within  $1/(24*3600)$ ).
- If I monitor the 1394 bus for exactly 5000 hours, how many cycles will I see? The answer is either  $(5000*3600*8000)$  or  $(5000*3600*8000)-1$ . Dividing either number with 5000 the results are even closer to 8000.

The "cycle synch" events are the real "hardware clock". The Cycle Start packets are a "software clock", a manifestation of the hardware clock on the 1394 bus.

## The 1394 Cycle - Part 4: Isochronous & Asynchronous Traffic

Now let us review the remaining parts of the diagram so that we can finally describe Isochronous and Asynchronous traffic into more detail. The diagram is repeated below for ease of reference.



**Figure Q.16—Cycle structure**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure Q.16)

We explained the significance of the cycle synch events in the previous paragraph. We can see that "cycle #m" started with a small delay (x) and "cycle #m+1" with another delay. However we explained how these delays get corrected, so they never accumulate.

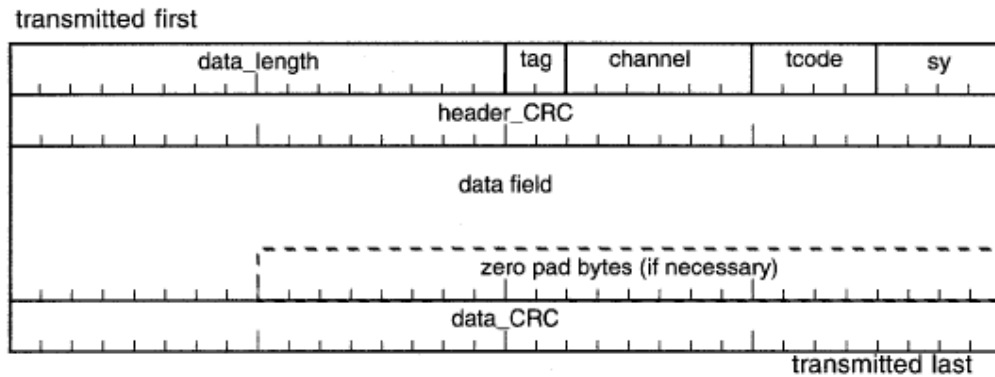
The "gaps" shown around the diagram are durations of "bus idle time". According to the 1394 rules, between isochronous packets either no gap appears (concatenated packets) or a small isochronous gap appears. When the last node that had an isochronous packet to transmit is finished, then everyone is "waiting" for the next isochronous packet but there is none. The bus stays idle a little longer (this is called a *subaction gap*) and the 1394 chips understand that the isochronous period is over, so they proceed with the asynchronous traffic.

All the square boxes in the diagram represent data packets. We can see the "Cycle Start" packet at the beginning of each cycle. The boxes labeled "ch J", ..., "ch N" are isochronous packets, where "ch" stands for "channel". "Packet B" and "Packet C" are asynchronous packets that happen to also have an acknowledgement.

This brings us to one significant difference between isochronous and asynchronous packets:

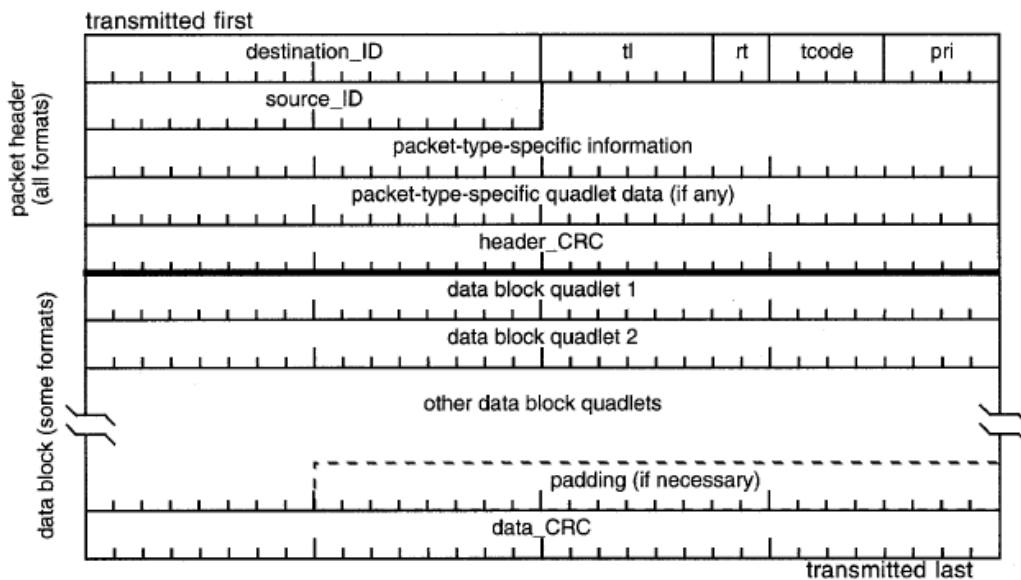
- Asynchronous packets have a target physical ID in their packet header. This is either a specific 1394 node, or all the nodes (broadcast packet when Physical ID is 63). An asynchronous packet is either "addressed" to a specific 1394 node or to everyone.
- Isochronous packets do not have a target physical ID in their packet header. Instead they have an "Isochronous Channel Number" which can take values from 0 to 63. The channel number acts as a "logical label" to this packet in the same way that "CNN" acts as a label for a particular TV broadcasting frequency.

Let us review the packet formats of Isochronous and Asynchronous packets as described in the 1394 standard:



**Figure 6-17—Isochronous data block packet format**

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-17)*



**Figure 6-3—Asynchronous packet format**

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-3)*

The Isochronous packet format diagram is straight forward to understand because there is exactly one type of isochronous packet. There are many types of asynchronous packets and figure 6.3 shows the generic format, with some hints about which parts are optional.

We must make sure now that no confusion creeps in:

- All packets, asynchronous and isochronous get propagated across the whole 1394 bus (provided they can be repeated everywhere).
- The physical ID in the header of an asynchronous packet does not make this packet follow one particular "route" or another. All 1394 nodes "listen" to the traffic that "passes" on the 1394 bus and when they "sense" a packet with their own physical ID (or the broadcast) then they "pick it up", i.e. they decide to receive it.

- A 1394 node can only receive the asynchronous packets with its own physical ID (or the broadcast ID). A node cannot "eavesdrop" on the traffic of another node. Only highly specialized and very expensive devices called "1394 bus analyzers" can receive all asynchronous packets.
- Asynchronous packets may be acknowledged by their receiver.
- In contrast, isochronous packets do not get specifically addressed to any node.
- Any isochronous capable 1394 node can receive any isochronous packet (from any channel).
- Multiple nodes can receive isochronous packets from the same channel number.
- Isochronous packets never get acknowledged by their receivers.
- An isochronous channel number can only appear once in every cycle.
- A 1394 node may transmit multiple isochronous packets in every cycle, but they have to use different channel numbers.

### The Essence of Isochronous Traffic

After having listed all these technical definitions, rules and their implications we are getting closer to understanding the big picture. So it is time to move to a higher level and work our way down until we meet the amazing world of 1394 again.

When I am copying a data file from one computer to another, no one "cares" exactly how long this operation will take. As long as it is not extremely slow, no one "cares" much. The file will end up on the second computer anyway, its data will be valid; a couple of milliseconds more or less don't make any difference. However we do "care" that all the data will be copied and that it will be valid.

In contrast, when playing back some music (an audio stream) everybody cares that the "next" pieces of the song are read from the file in time and that they are also sent to the audio device in a timely fashion. If some of the audio data is "bad" we will hear it. The slightest delay or error results in an audible discrepancy. The visual discrepancies of video streams are not so easy to detect for the human eye. If a frame is missed, we never notice. If 3-4 frames in a row are missed then we might notice. If a portion of some frame was not properly decoded (or the image data was bad) we will probably not notice at all.

As we can see, these are different "types of data" with different requirements. Information like an audio or video stream are generically called "streaming data".

Apart from timely delivery (knowing that the needed data will have been received when it is time to "render" it) there is another equally important issue with "streaming data". This is the "Rate of Delivery".

Suppose that you want to playback a 800MB MPG movie file whose duration is 1 hour. This file is on your hard disk. You make a test copy of the file and it takes 10 seconds to complete. Your computer is pretty fast; it can read and write all that data in just 10 seconds. You start to playback the movie and the MPG is getting decoded and displayed with a modest 30% CPU time. The computer is obviously fast enough, so why should it have a problem with "streaming data" like your movie?



The problem of course is that other devices might get involved, devices that have limited resources, specifically limited memory buffers. Your computer could possibly decode the whole audio stream of that movie in 60 seconds, but that would generate audio data that are 200MB in size. You cannot send that 200MB to your audio device, because its memory buffers can only hold 2 seconds worth of sound. And there starts the trouble.

Writing a software program that delivers streaming data to a hardware device at the appropriate rate that the device can handle, without making timing mistakes ever, never being early (the device is not ready yet), never being late (the device sits idle) is not such an easy task as it might seem. Moreover, things are even more complex for "simple" devices that generate "streaming data". They have to control their transmission rate too, or else they might drive the receiver out of resources.

So streaming data presents us with two major difficulties:

1. Data must never arrive late.
2. Data should not arrive too fast.

The 1394 bus protocol provides isochronous support in order to solve both of these problems at the same time.

Suppose that we have an uncompressed video stream with the following characteristics:

- Image Size: 800x600
- Pixel Encoding (Color Depth) = 16-bits per pixel
- Frame Rate: 20 Frames per Second (fps)

Every image frame has  $800 \times 600 = 480,000$  pixels, at 2 bytes each it yields 960,000 bytes per frame. Every frame must get transmitted in  $1/20^{\text{th}}$  of a second, or every  $8000/20 = 400$  cycles.

Since we can transmit only one isochronous packet per cycle (on a given channel number), we must transmit  $960,000/400 = 2,400$  bytes in every cycle.

The camera then grabs one frame, "cuts" it in 400 packets of 2,400 bytes each and asks the 1394 chip to transmit the packets on isochronous channel 8. That's all it takes to solve both problems!

The camera has nothing more to worry about; it trusts the 1394 chip to do its work properly and know that after 400 cycles that image has been transmitted, with a very stable rate (one packet every cycle).

The receiver of the video stream is also happy with this scheme. For starters, it doesn't have to allocate additional buffers in case data arrives faster than expected. The receiver knows the exact data rate, so it can make the appropriate buffer arrangements. Additionally, the receiver doesn't have to do any timing to display at the 20 fps rate. If some frames arrived early, then the receiver would have to "delay" their display so that the video is smooth. But now the receiver knows that frames are always on time and never early. The arrival of a new frame also indicates that it is time to display that new frame.

So that is what happens with 1394 isochronous traffic. A device that wants to generate a stream of data (whatever kind of data), selects a channel number, chops that data into packets

of the appropriate size and then leaves the dirty work to the 1394 chip, which will dutifully transmit one of these packets on every cycle. The 1394 protocol guarantees to the chip that it will certainly transmit one isochronous packet per cycle.

Of course, it is not required by 1394 nodes that send isochronous packets to use each and every cycle. For example the 960000 bytes of the video frame in our previous example could be packaged in packets of 3000 bytes, which would require 320 cycles to transmit a whole frame. Then, the camera would instruct the 1394 chip to stay idle until an additional 80 cycles elapse and then start with the next frame.

### **Isochronous Bandwidth**

Obviously, the bandwidth of 1394 is not infinite, so some more protocol rules have to exist to keep things in order. Please note, that these are "soft" rules; the 1394 standard describes them but there is no hardware mechanism that enforces them. They are rules that all 1394 nodes "agree to follow willingly":

- Every 1394 node that wants to transmit isochronous traffic must "reserve" (allocate) a unique channel number (a channel that is not currently in use). This guarantees that any channel number only appears once in every cycle.
- The isochronous period should not exceed 80% of the available bandwidth, which means it should not exceed 80% of the 1394 cycle, so it should not exceed 100 microseconds in every cycle. These 100 microseconds are translated by the 1394 standard into "*4915 bandwidth allocation units*".
- Every 1394 node that wants to transmit isochronous traffic must "reserve" (allocate) isochronous bandwidth by allocating "bandwidth allocation units". This is like reserving a "time slot" of the desired duration within the isochronous period.

If all 1394 nodes follow these rules, then at least the 20% of the bus bandwidth will be available for asynchronous traffic and the bus will operate without problems.

All that we described above often get summarized by saying that isochronous traffic is:

- **Guaranteed Timing**  
Packets get certainly transmitted every 125 microseconds.
- **Non Guaranteed Delivery**  
There are no acknowledgements so the sender cannot know if the receiver(s) actually got the data. Incoming packets may get discarded because of FIFO errors.
- **Unfair**  
Only those devices that have allocated bandwidth units are allowed to transmit.

### **Asynchronous Traffic**

From a high level, asynchronous traffic has the opposite characteristics of isochronous:

- **Guaranteed Delivery**  
In almost all cases an acknowledgement is sent back to the sender.

- Non Guaranteed Timing  
Best-effort transmission is used for asynchronous traffic. The sender does not know how "soon" the packet will be transmitted.
- Fair  
All devices will eventually get to transmit.

An additional characteristic that asynchronous transmission has is "*Fair Arbitration*". Each and every node will get equal "chance" to transmit its asynchronous traffic. In contrast, isochronous transmission is not fair: The 1394 nodes that were *quick enough* to allocate the available bandwidth allocation units first, are the only ones that are permitted to transmit isochronous traffic.

As we saw earlier, a device like a camera usually prepares a set of isochronous packets and then asks the chip to transmit them. The term used for this action is that "*the isochronous packets are queued for transmission*".

A similar thing happens with asynchronous traffic. The sender wants to send a file to the receiver (using some higher-level protocol like SBP2), so the sender possibly prepares many asynchronous packets and queues them for asynchronous transmission.

What happens then is as follows:

1. The 1394 chip arbitrates for permission to transmit a single asynchronous packet (which of course can only happen during the asynchronous period).
2. The 1394 chip wins arbitration (gets permission) and transmits the packet.
3. If there are more packets to send, then go to step 1.

Of course, there might be other nodes in the bus that have asynchronous traffic to transmit. So the other nodes run the exact same procedure and the fair arbitration algorithm makes sure that they all have equal rights to arbitration.

Let us suppose there are 10 nodes that have asynchronous traffic. If we suppose that all these nodes have packets of the same size to send, and the size is such that only 2 packets can fit in the asynchronous period, then each of the 10 nodes will be transmitting 1 packet every 5 cycles!

In practice we typically never know the traffic load of other nodes, so sometimes a node can get 1 asynchronous packet every 5 cycles (like the example above), sometimes 10 asynchronous packets within the same cycle (nobody else had anything to transmit), etc.

The net result is that asynchronous traffic does not have guaranteed timing; we don't know how long an asynchronous packet will wait in the queue before it gets transmitted.

### Asynchronous Stream Packets

It would be worth noting in this document the 1394 standard provides with a special exception to the rules described in the previous sections.

It is really a simple concept, but can easily lead to confusion:

*Asynchronous Stream Packets are packets of the isochronous packet format (transaction code = 10) that get transmitted in the asynchronous period.*

The reasons for this are beyond the scope of this document, but the concept of asynchronous stream packets is extremely simple to grasp: The 1394 node prepares an isochronous packet (a packet with the isochronous packet format) but then queues it in the asynchronous transmission queue instead of the isochronous transmission queue, an action that is permissible by the LINK chip. Thus, this isochronous packet gets transmitted in the asynchronous period, with all the restrictions of an asynchronous packet (fair arbitration, unknown timing).

The opposite exception, queuing an asynchronous packet in the isochronous queue, as we saw earlier, is not permitted.

Asynchronous Stream packets are a very attractive feature, especially as a device-to-host notification mechanism, because the device need not know the current physical ID of the controlling host.

However, from the side of the receiver these packets look to high level software exactly the same as normal isochronous packets. This means the receiver must have setup isochronous receive DMA as appropriate; the asynchronous stream packets will not be received by the same DMA engine that handles the rest of the asynchronous traffic.

### Packet Size Restrictions

The amount of data inside a data packet, called the "payload", cannot be arbitrarily large. The maximum size permitted depends on the packet type (asynchronous vs. isochronous) and the packet transmission speed.

The next two tables show the limits that the 1394 standard imposes on the payload:

**Table 6-4—Maximum payload size for asynchronous data packets**

Data rate	Maximum payload (bytes)	Comment
S25	128	TTL backplane
S50	256	BTL and ECL backplane
S100	512	Alpha or Beta mode
S200	1024	
S400	2048	
S800	4096	Beta Mode only (see Table 16-18)
S1600	8192	
S3200	16 384	

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-4)*

**Table 6-7—Maximum payload size for isochronous stream packets**

Data rate	Maximum payload (bytes)	Comment
S25	256	TTL backplane
S50	512	BTL and ECL backplane
S100	1024	Cable base rate
S200	2048	—
S400	4096	—
S800	8192	—
S1600	16 384	—
S3200	32 678	—

*(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-7)*

We can notice that isochronous packets are in favor. The maximum payload of isochronous packets for any transmission speed is the double of the asynchronous packet limit for the same transmission speed.

## Link Layer Operation

The following *state machine* diagram taken from the 1394 standard shows the operation of the 1394 Link Layer, in a way similar to a flowchart:

### 6.3.3 Details of link layer operation

The operation of the link layer packet transmitter and receiver is described by the state machine in Figure 6-21.

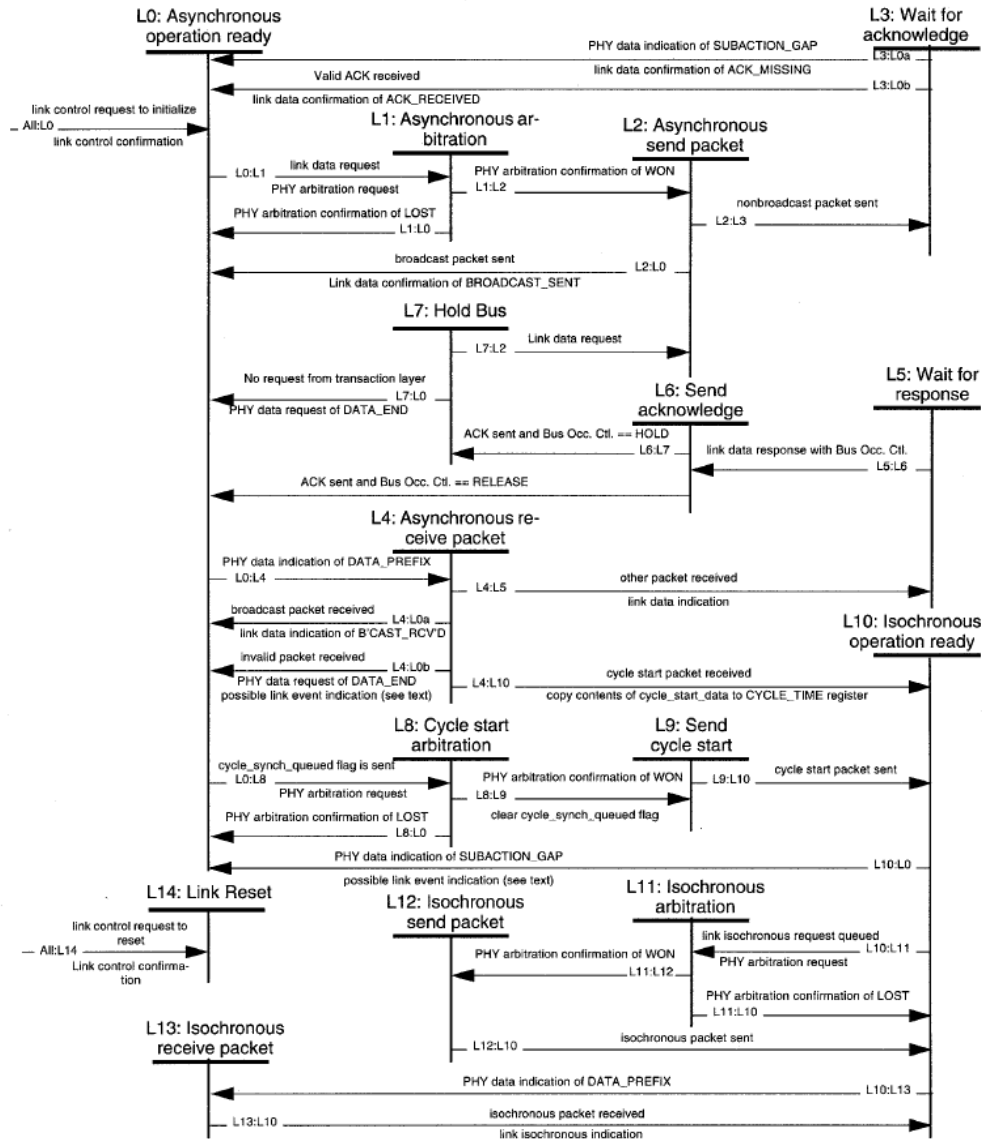


Figure 6-21—Link layer packet transmit/receive state machine

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 6-21)

The diagram is quite complex, but we can quickly traverse some “example” paths and see how all the descriptions that we gave earlier fit nicely inside this state machine.

- The state machine starts at state L0 (Asynchronous Operation Ready).
- When an asynchronous packet is received the transition L0:L4 is taken.

- If this is a normal asynchronous packet, then the transition L4:L5 is taken, then L5:L6 and finally L6:L0 and we are back where we started.
- If at L0 a Cycle Start packet is received, then the path L0:L4:L10 is taken and the state machine is now "Isochronous Operation Ready". This starts the isochronous period.
- If there is an isochronous packet to be sent, then the path L10:L11:L12 is taken and then back to L10.
- At some point a subaction indication occurs and the transition L10:L0 is taken. The isochronous period is over and we move once again back to the asynchronous period.

### Transaction types: Read, Write, Lock

Earlier in this document we discussed the 1394 Addressing Model and we saw that the 1394 protocol abstracts the bus as a fixed 64-bit address space. The operations defined on this model (the actions that can take place between nodes) are called "*transactions*". The term should not be confused with its equivalent in database theory. 1394 "*transactions*" have nothing in common with database "*transactions*". In 1394, the term "*transaction*" has the same meaning as in "*Transaction Layer*".

Needless to say, transactions are performed with asynchronous packets.

The 1394-2008 standard lists the available transaction types as follows:

*The transaction layer provides three operations for the transfer of data between nodes:*

- a) Write transaction—Data are transferred to an address in a different node.*
- b) Read transaction—Data are retrieved from an address in a different node.*
- c) Lock transaction—Data are sent to a different node, used to perform an indivisible function, and the results returned.*

These different transaction types are mapped to different *transaction codes* of the asynchronous packet header.

In general, each transaction is split into a "transaction request" packet and a corresponding "transaction response" packet. Transaction response packets have their own transaction codes in the packet header.

The general procedure is as follows:

1. Node A sends transaction request to Node B.
2. Node B acknowledges the transaction request.
3. Node B does some processing according to the request.
4. Node B sends a transaction response to node A.

In step 2, the acknowledge might be "negative". Node B is unable to receive this request packet (any packet most likely) at this time, so the story ends there.

If the packet is received, then Node B will examine the "validity" of the request and will either do the required processing and reply with a "success" response code, or it will reject the operation and reply with a "failure" response code.



After the transaction request is transmitted and acknowledged, and until the transaction response has been received, Node A considers the transaction as "pending".

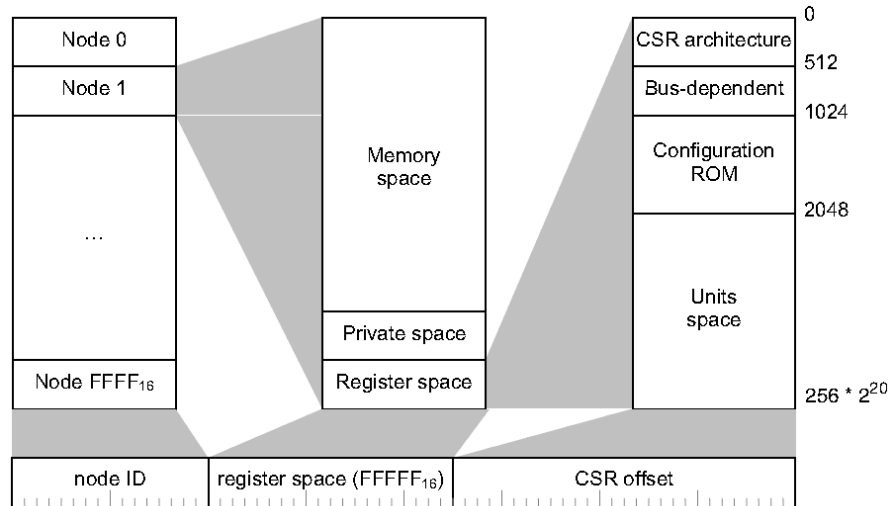
Another option for Node B is to not answer back at all! In this case Node A will "timeout" and will be left wondering if there is something wrong with Node B. And more often than not, not sending back the transaction response packet is an indication of some internal error in Node B.

It should be stressed that all these "transactions" are at the Transaction Layer level. There is no hardware mechanism that forces a response to be sent and every transaction to be "matched" appropriately. In fact Node A is free to directly send a transaction response packet to Node B; Node B will receive the packet normally, will examine it and see that it has no pending request for it and then trash it.

## The CSR Model

As we stated earlier, FireWire follows the Command and Status Register (CSR) architecture of IEEE Standard 1212-2001. We also stated that the 1394 protocol uses fixed 64-bit addresses, where the high 16 bits act as the "external" address and the low 48 bits act as the "internal" address.

The diagram that depicts this in IEEE1212 is shown below:



**Figure 4—64-bit fixed addressing hierarchy**

*(Copyright IEEE. Used with permission. Identical to IEEE 1212-2001 Figure 4)*

We already have seen a very similar diagram from the 1394-2008 standard. We can see that the 48 "internal" address bits are further subdivided into "Register Space" (20 bits) and "CSR Offset" (28 bits). This internal subdivision is provided mainly for "organizational" reasons, creating some internal "law and order" and is mainly a "soft" division. The OHCI chips may provide some slightly different functionality for addresses low in the Register Space, but for all practical purposes of this document we can assume that a CSR's address utilizes the full 48 bits.

So what exactly is a Command and Status Register (CSR)? It seems that this term is so universally fundamental and trivial that neither the IEEE1212 standard (which has the term CSR in its title) nor the IEEE1394-2008 standard care to define it clearly. Obviously everyone knows what a "Register" is (right?), so a CSR is a "register" that can "implement" commands and inform us about "status". I don't know if that leaves out much, but it is a start to our discussion.

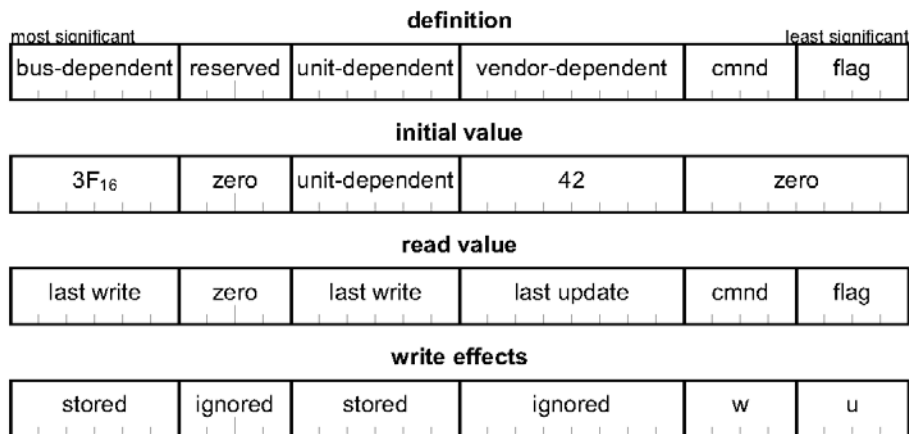
In the 1394 model a CSR is basically a "software<sup>5</sup> mini-module" tied up to a particular 48-bit offset, similar to an interrupt handler tied to an interrupt number. When an asynchronous request packet arrives for that specific 48-bit offset, the device driver first checks if there is any "software mini-module" associated with this offset. If not, the packet is trashed. If yes, then the packet is handled to the "software mini-module" for processing.

<sup>5</sup> or firmware

What the "software mini-module" does is completely irrelevant. It does what the corresponding CSR specification says it will do. Maybe it reads some information for the actual device (e.g. a camera) and returns that "status" information. Or it does an action on the actual device (e.g. start or stop a camera), which classifies it as "command". Of course, it can do both: When a read transaction request is received then it returns "status"; when a write transaction request is received then it performs an action.

We use the term "software mini-module" instead of "software module" to indicate that in most systems a "software module" (a bigger thing) hooks up several different 48-bit offsets, so when a packet comes for one of these offsets it is some part of the software module that takes care of it.

Usually the size of a CSR is a quadlet (4 bytes). The diagram below shows the standards' convention for describing part of the operation of a CSR:



**Figure 7—CSR specification example**

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 7)

The first row shows the overall definition of the internal parts (bit fields) within the CSR, and the other rows how the CSR behaves upon initialization, read or write transactions.

The following is an example CSR from the IIDC standard:

#### 4.1 Camera initialize register

Offset	Name	Field	Bit	Description
000h	INITIALIZE	Initialize	[0]	If you assert this bit, Camera will re-set to initial state. This bit is self cleared, shall wait becoming "0"
		-	[1..31]	Reserved

0-7	8-15	16-23	24-31
Reserved			

Initial values	System dependent
Read values	'0' Done '1' Busy (under initialization)
Write effect	'0' no effect '1' set initial state

This is a very simple CSR. Only one bit is defined and the rest are reserved.

## Configuration ROM & GUIDs

In the earlier diagrams about the 1394 address space we can see a section of the memory space marked as the "Configuration ROM". This is a special purpose memory block which can contain extended information about a node's capabilities. Only "transaction-capable" nodes are required to implement the Configuration ROM (host adapters, cameras, disks).

The Configuration ROM is structured in a hierarchical way as shown below:

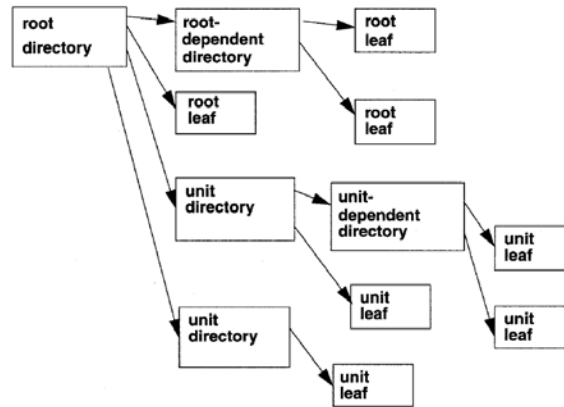


Figure 8-18—Configuration ROM hierarchy

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 8-18)

The memory layout looks like this:

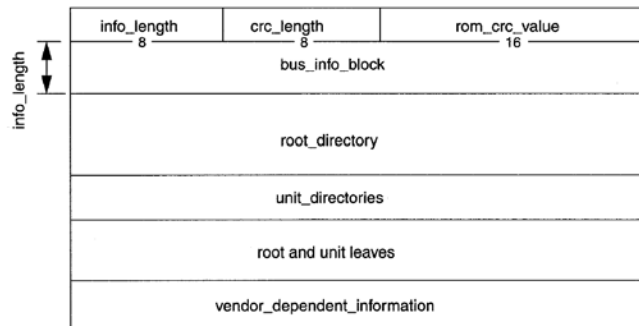


Figure 8-20—General ROM format

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 8-20)

The Bus\_Info\_Block is very important and is shown below:

31 <sub>16</sub> ("1")	33 <sub>16</sub> ("3")	39 <sub>16</sub> ("9")	34 <sub>16</sub> ("4")
capabilities	cyc_clk_acc	max_rec	rsv
		max_ROM	generation
			b
			link_spd
node_vendor_ID			chip_ID_hi
chip_ID_lo			

(Copyright IEEE. Used with permission. Identical to IEEE 1394-2008 Figure 8-21)

The last two quadlets of the Bus\_Info\_Block contain the so called "GUID" which stands for "Globally Unique Identifier". The GUID is comprised by a 24-bit "Vendor ID" and a 40-bit "Chip ID". In fact manufacturers are required to store unique values in these fields in such a way that no two devices have the same 64-bit GUID.

GUIDs are usually being used by software to identify devices across bus resets, when the Physical ID may change.

## Transaction types

The 1394-2008 standard also categorizes transactions according to their "execution" characteristics as shown in the list below:

- **7.3.2.1 Unified transaction**  
*A unified transaction is defined as a transaction that begins with an acknowledged request subaction but is not followed by a response subaction. Only write transactions may normally complete as unified transactions. Read transactions and lock transactions do not normally complete as unified transactions.*
- **7.3.2.2 Split transaction**  
*A split transaction is defined as a transaction that begins with an acknowledged request subaction and is followed by an acknowledged response subaction. Read transactions, write transactions, and lock transactions may be split transactions. Other subactions may occur between the request subaction and the response subaction.*
- **7.3.2.3 Concatenated transaction**  
*A concatenated transaction is defined as a transaction that begins with an acknowledged request subaction and is followed immediately by the corresponding acknowledged response subaction, with no subaction gap between. Read transactions, write transactions, and lock transactions may be concatenated transactions. No other subactions shall occur between the request subaction and the response subaction of a concatenated transaction.*
- **7.3.2.4 Broadcast transaction**  
*A broadcast transaction is defined as a transaction that contains only an unacknowledged request subaction. Only write transactions may be broadcast transactions. Read transactions and lock transactions shall not be broadcast transactions.*
- **7.3.2.5 Pending transaction**  
*A pending transaction is defined as a transaction that is not yet completed.*

Basically most transactions are "Split". The sender sends a transaction request that gets acknowledged, other things happen and then the target returns a transaction response.

In some cases the target is able to send the response packet immediately after the transaction request packet, without leaving the bus available for other nodes to send their traffic. These are the "Concatenated" transactions.

The "Unified" case is a special case for write transactions. The sender sends the write transaction request and the target returns a special acknowledge code that means *"This transaction is completed and no further response will be sent for it"*.

The broadcast and pending transactions we have already covered earlier in this document.

## DMA

When FireWire first came into the market, different vendors provided different implementations of the LINK layer functionality. The difference was not only internal to the chips but applied also to their "external" interface. You needed different software code to talk to each LINK chip and different electronic designs on the adapters.

This is of course a highly undesirable situation in the software industry (and the hardware too), so as soon as FireWire gained enough momentum, the external interface of the LINK chips was standardized to what is today known as "*1394 OHCI*". OHCI stands for "*Open Host Controller Interface*" and there are similar standards for other technologies like USB. Technically we should say "*1394 OHCI*" to make sure no confusion occurs, but since we are in a 1394-specific context we will call the 1394 LINK chips as OHCI chips and use common expressions like "*The OHCI transmits a packet*" to mean "*The 1394 Link layer implemented by the OHCI chip transmits a packet*".

Providing an interface to 1394 for the Host Controller is not an easy task, especially for high performance technologies like FireWire.

Ask this simple question: "When a packet gets received, where does it get stored"?

There are two possible answers:

1. The packet gets stored in on-board memory and then the adapter raises an interrupt. The host software then processes the interrupt and copies the packet from there to main memory.
2. The packet gets stored directly in main memory using DMA (Direct Memory Access). After getting completed the adapter raises an interrupt to inform the host software that a new packet is available.

The same options are available for outgoing packets (packets to be transmitted):

1. The packet gets copied by host software onto the on-board memory. Then the software tells the adapter to transmit the packet.
2. The adapter can read the packet directly from main memory using DMA and then transmit it.

The option of having on-board memory has the following shortcomings:

- How much memory do you put on the board so that you don't have out-of-memory conditions?
- Putting memory on the boards also requires extra circuitry, which (together with the memory itself) make the boards more expensive and less reliable.
- Usually copying a packet from on-board memory to main memory, or in the other direction is much slower compared to other options such as DMA.
- Such data copies are usually done either by the CPU itself (leading to higher CPU utilization) or the so called "System DMA controllers" which might not be available at all times and are slower too.

However, the option of having a device reading or writing directly to main memory is also complicated, because the chip must be able to act as "*Bus Master*" in the Host Controller's bus

(usually a PCI bus) and perform "Bus Mastering DMA". This means that the device is capable of "talking" in the PCI protocol and initiate and control itself transfers to and from main memory in accordance to the PCI protocol.

Bus Mastering DMA largely improves performance, but requires much more complex chips. This is the reason that older Ethernet network adapters used the on-board memory approach. Modern and usually quite expensive GigE<sup>6</sup> adapters may have DMA capability.

FireWire, being a high performance bus, had no option but to standardize OHCI as a chip that is capable of Bus Mastering DMA. The following conceptual block diagram for the OHCI 1.1 specification illustrates this:

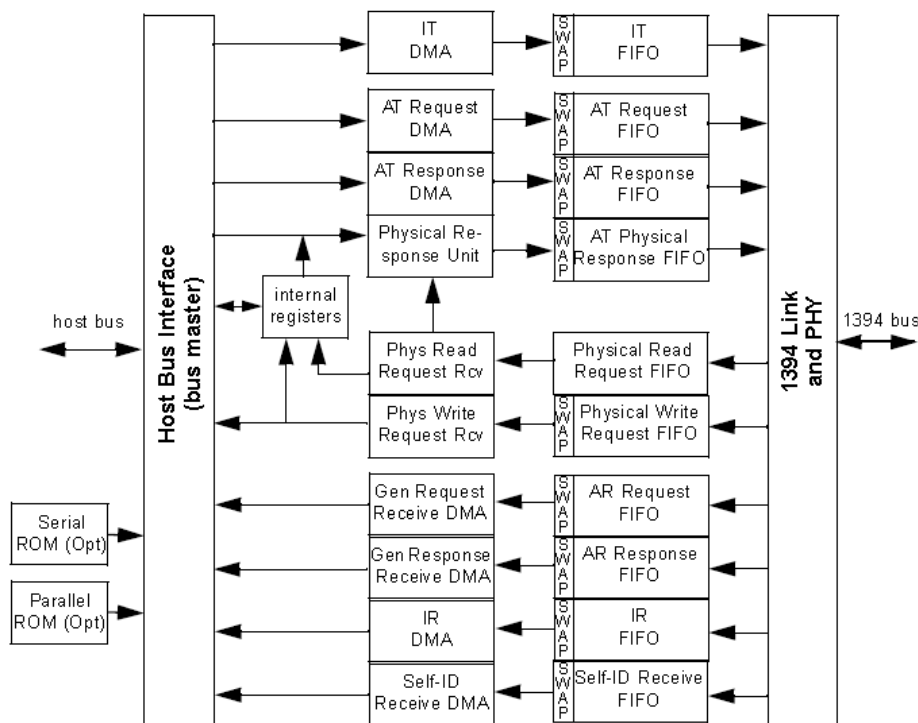


Figure 1-1 — 1394 Open HCI conceptual block diagram

Some acronyms deserve clarification:

<b>AR</b>	Asynchronous Receive
<b>AT</b>	Asynchronous Transmit
<b>IR</b>	Isochronous Receive
<b>IT</b>	Isochronous Transmit

We can see from this diagram that the OHCI chip connects (bridges) the 1394 bus to the host bus. The OHCI chip operation is logically organized into several "DMA Engines" each of which is dedicated to a particular type of 1394 packets.

<sup>6</sup> Gigabit Ethernet



In this conceptual diagram we can see various FIFOs, special purpose memories whose size is usually between 2K and 8K, where packet data are temporarily buffered while awaiting the grant to be transferred over the host bus to the host memory. These memories do not get a full packet and then start the DMA to the host memory. Packet data gets transferred to host memory "block-by-block" while the packet is still being received. The same happens in the reverse direction too. Outgoing packets get moved "block-by-block" from main memory to the appropriate FIFO as the packet is being transmitted on the 1394 bus.

## DMA Contexts & Context Programs

This section will describe in the shortest possible way the operation of the OHCI chip. The OHCI 1.1 specification states the following in paragraph 1.3.2, titled "DMA":

*The 1394 Open HCI supports seven types of DMA. Each type of DMA has reserved register space and can support at least one distinct logical data stream referred to as a DMA context.*

**Table 1-1 — DMA controller types and contexts**

DMA controller type	number of contexts
Asynchronous Transmit	1 Request, 1 Response
Asynchronous Receive	1 Request, 1 Response
Isochronous Transmit	4 minimum, 32 maximum
Isochronous Receive	4 minimum, 32 maximum
Self-ID Receive	1
Physical Receive & Physical Response	0 (not programmable like those above)

*Each asynchronous and isochronous context is comprised of a buffer descriptor list called a DMA context program, stored in main memory. Buffers are specified within the DMA context program by DMA descriptors.*

So a "DMA Context" is distinct logical data stream. Actually it is more helpful to think of a DMA context as a subordinate CPU inside the OHCI chip. It is actually an *execution engine*, a small CPU that executes a program. That program is called the *DMA Context Program*.

What do such programs look like? Usually something like this (example for AR):

1. When a packet up to 4K in size arrives, you can receive the first 1024 bytes at main memory physical address 0xFFA04B00, 3072 bytes at 0xFEED2000. When packet reception is complete, raise AR interrupt.
2. When a packet up to 4K in size arrives, you can receive the first 1600 bytes at main memory physical address 0xFFA619C0, and 2496 bytes at 0xFBE23000. When packet reception is complete, raise AR interrupt.
3. When a packet up to 4K in size arrives ...
- ...
99. Goto step 1

The reasons that packets might have to be split in multiple locations in main memory has to do with virtual memory systems, where a memory buffer that is contiguous in virtual memory space is actually split across several physical memory pages.

Who prepares the instructions of the DMA context programs for all the DMA contexts? In the case of PCs, it is the 1394 device driver that does this. For other types of devices, like cameras, it is the firmware of the device that prepares the context programs.

What is of special interest is the operation of the isochronous receive contexts. As shown in the table above, there are at least four of them. In this case, a software program is expecting a video frame from a camera and the video frame is transmitted isochronously. The software has configured the camera to some specific video format and isochronous channel number. This means that the software knows the exact number of bytes that are needed to receive a single frame, the isochronous packet size and of course the number of packets per frame.

So, the software allocates a buffer of the appropriate size and asks the device driver to prepare a DMA context program for one of the isochronous receive DMA contexts that will receive the expected number of isochronous packets directly into that buffer. So when the frame gets received, it gets received directly into the application's memory buffer, without any extra memory copies or any other involvement of the host's CPUs. This is a significant advantage of FireWire over other competitive technologies.

It should be clear that in this example the software must also know the order of pixel transmission of the image by the camera, so that it can provide the memory locations in the context program as appropriate.

If, for example, the camera is transmitting the image top-to-bottom, then the first instruction of the DMA context will point to the start of the buffer, the second instruction a little after that (a whole packet payload), etc, and the last instruction one packet payload before the end of the buffer.

If instead the camera is transmitting the image bottom-up, then the first instruction of the DMA context will point one packet payload before the end of the buffer, the second instruction two packet payloads before the end of the buffer, etc, and the last instruction at the beginning of the buffer.

IIDC cameras transmit their images top to bottom, but this is just a convention of the IIDC standard.

## Serial Bus Management

The term Serial Bus Management refers to a "layer" in the 1394 protocol model that is responsible for some higher level operations that maintain the bus in a "healthy" operational status.

Many nodes on the 1394 bus may be capable of performing the duties required by the Serial Bus Management layer, but at any point in time only one node is acting as the "Serial Bus Manager" (SBM).

Immediately after the bus reset processing is completed and normal traffic resumes, all SBM-capable nodes perform a well defined "contention" which has one winner. That winner is the "elected" SBM.

Some of these tasks are listed below:

- Make sure that the bus is "Isochronous Capable" (i.e. a Cycle Master Capable node is the root node).
- Optimize bus performance characteristics (e.g. the gap\_count parameter).
- Implement the BUS\_MANAGER\_ID, BANDWIDTH\_AVAILABLE, CHANNELS\_AVAILABLE, and BROADCAST\_CHANNEL registers.

In most FireWire systems the SBM services are provided by a PC.