A Comparative Study of the Linux and Windows Device Driver Architectures with a focus on IEEE1394 (high speed serial bus) drivers

A thesis submitted in fulfilment of the
requirements for the degree of

MASTER OF SCIENCE

of

RHODES UNIVERSITY

by

MELEKAM ASRAT TSEGAYE

December 2002

Abstract

New hardware devices are continually being released to the public by hardware manufactures around the world. For these new devices to be usable under a PC operating system, device drivers that extend the functionality of the target operating system have to be constructed. This work examines and compares the device driver architectures currently in use by two of the most widely used operating systems, Microsoft's Windows and Linux. The IEEE1394 (high speed serial bus) device driver stacks on each operating system are examined and compared as an example of a major device driver stack implementation, including driver requirements for the upcoming IEEE1394.1 bridging standard.

Keywords

Linux device drivers, Windows device drivers, IEEE1394, IEEE1394 drivers, IEEE1394.1 bridging

Acknowledgments

I would like to thank my supervisor Prof. Richard Foss for his guidance throughout the period of this work, my co-supervisor Bradley Klinkradt for his input, and all the people who helped in proofreading (Abimbola Ogunsipe, Nakkiran Sunassee, Stanley Ntakumba, Robert Kraft and Seun Oyekola).

The financial assistance from the Andrew Mellon Postgraduate Scholarship towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to Rhodes University or the donor.

Table of contents

Table of Figures

2 Device Driver Architectures

3 Designing Device Drivers

4 Implementing Device Drivers

5 IEEE1394 Driver Implementations

6 IEEE1394 Driver Performance Tests

9 Appendix

1 Introduction

The Microsoft Windows operating system is the most widely used operating system in households and businesses around the world [Statmarket, 2002]. When a new device is produced for a PC by a hardware manufacturer, the hardware manufacturer or an operating system's vendor releases device drivers that will enable that device to operate under a target operating system. One of the device drivers released by a hardware manufacturer is usually designed for Microsoft's Windows. Other operating systems for which a device driver may be packaged include MAC OSX from Apple Corporation. In some cases, drivers may be packaged for a UNIX operating system used in professional studios, for example SGI UNIX.

In recent times a UNIX clone operating system called Linux, whose kernel was created by Linus Torvalds [Linus FAQ, 2002] and is maintained by numerous developers around the world, has been gaining in popularity. Its popularity is helped by the fact that it is available at no cost to its users, compared to Microsoft's Windows which costs a substantial amount of money. Device drivers for the Linux operating system are mostly developed by private individuals around the world, who write driver code in their spare time. This is in contrast to drivers for Windows produced by Microsoft, one of the largest commercial software producers in the world, or a hardware manufacturer with fulltime, paid developers.

This work presents the device driver architectures currently utilised by the Linux and Microsoft Windows operating systems, comparing the various facilities offered by the two operating systems. The differences and similarities of the two driver architectures are highlighted through two generic memory device drivers, whose implementation under each operating system is described. As an example of a major device driver stack implementation, the IEEE1394 (high speed serial bus) protocol driver stacks of the two operating systems are examined and compared. IEEE1394 is a protocol for digital audio/video devices, which is targeted at home entertainment, and professional audio/video systems, offering very high data transfer rates. Currently available IEEE1394

devices have a theoretical data transfer rate of 50MB per second (400Mbps) as defined in the IEEE1394 specification.

An IEEE1394 client driver that allows generic IEEE1394 operations already exists under the Linux operating system, but no operational publicly available generic IEEE1394 client driver exists under the Windows operating system. Operations that generic IEEE1394 client drivers should provide to applications are identified, and two generic IEEE1394 client drivers, one for each operating system, are implemented. These client drivers allow applications to perform IEEE1394 operations from the application level. The operations include asynchronous reads, writes, locks, and isochronous talking and listening.

Test applications are constructed to demonstrate the use of the IEEE1394 client driver APIs, and tests are run to evaluate the performance of the IEEE1394 stacks of each operating system by transmitting large buffers of data using the client drivers. The amount of data transmitted, and the time taken for the data to be transmitted are recorded and used to calculate the data transmission rate for each driver in MB per second. The results are then presented graphically. There is currently no published literature available that examines and compares the device driver architecture, and the IEEE1394 stacks of the two operating systems. This works intends to fill this gap. The next section provides previous studies of the two driver architectures conducted by other researchers. The subsequent sections provide a brief overview of the each of the chapters in this work.

1.1 Related Work

A number of works exist that examine the device driver architectures from the two operating systems in detail as well as give examples of device drivers for specific hardware platforms such as the Universal Serial Bus (USB). The Windows device driver development kit (DDK) and Microsoft's hardware development site [Microsoft Hwdev, 2002] provide extensive information on the Windows driver architecture. They also include documentation that details how drivers can be constructed for the various types of hardware devices that can be used on a PC such as sound, network and storage devices.

Cant [Cant, 1999] and Oney [Oney, 1999] provide a detailed examination of the Windows NT driver architecture including implementation of drivers for specific hardware platforms such as the parallel port and USB. Lozano [Lozano, 2002] explains and expands on device driver development documentation already available from Microsoft. Dhawan [Dhawan, 1995] looks at network drivers from various operating systems including Windows and UNIX.

Although no device development kit comparable to the Windows DDK exists in Linux, the Linux headquarters [Linuxhq, 2002] provides some information on Linux kernel programming. Jay *et al* [Jay *et al*, 2001] and Kulkarni [Kulkarni, 2000] present Linux driver programming by example. A comprehensive driver development guide is provided by Rubini *et al* [Rubini *et al*, 2001]. Implementation of drivers for specific hardware platforms such as USB and SCSI is presented by Fliegl [Fliegl, 2000] and Gilbert [Gilbert, 2002]. Pajari [Pajari, 1991] provides a guide for creating device drivers to be used by a generic UNIX system. Some of his concepts also apply to Linux since Linux is a clone of the UNIX operating system. Sun Microsystems [Sun, 2002] also provide a guide to driver writing for their own implementation of a UNIX operating system, Solaris.

1.2 Architectures

Chapter 2 examines the device driver architectures used by the Windows and Linux operating systems, followed by a comparison of the two architectures. The IEEE1394 protocol is introduced, and the IEEE1394 stacks currently in use by the two operating systems are examined and compared. The new IEEE1394 specification called IEEE1394.1 is also presented. This is a draft document that defines bridging of multiple IEEE1394 buses. The requirements that the IEEE1394 stacks of the two operating systems should meet to be IEEE1394.1 bridge-aware are also presented.

1.3 Driver Design

Chapter 3 presents issues that must be considered when designing device drivers for the two operating systems. The design issues discussed are memory management, functional design, hardware management, and driver layering (modularisation). The designs of the

current IEEE1394 driver stacks of the two platforms are looked at, and operations that new IEEE1394 client drivers should provide are outlined.

1.4 Driver Implementation

Chapter 4 presents components that must be implemented to create Windows and Linux device drivers, as well as a comparison of the Windows and Linux driver APIs. Issues such as driver naming and installation, which are important to device driver developers are examined. An implementation of a simple generic memory device driver, for each operating system, is also presented. The chapter ends with a look at the driver development environments that are used to create device drivers for the two operating systems. One of the tasks that must be performed when developing any piece of software is debugging. Debugging of device drivers, using the debuggers available for each operating system's kernel, is also investigated.

Chapter 5 presents the current IEEE1394 driver implementations of the two operating systems. The implementation of two new IEEE1394 client drivers called *raw1394*, one for each operating system, is discussed. A comparison of these two driver implementations is also presented. Modifications that must be made to the current Linux IEEE1394 host controller and IEEE1394 core drivers, to make the Linux IEEE1394 stack IEEE1394.1 bridge-aware, are also presented. The chapter ends with a demonstration of the IEEE1394 client drivers and test software created during this work.

1.5 IEEE1394 Driver Performance Tests

Chapter 6 presents tests conducted to compare the performance of the Linux IEEE1394 stack to the Windows IEEE1394 stack. The tests conducted involve the transfer of large data buffers broken up into common packet sizes, using different combinations of IEEE1394 client drivers.

2. Device Driver Architectures

A device driver enables the operation of a piece of hardware by exposing a programming interface that allows a device to be controlled externally by applications and parts of an operating system. This section presents the driver architectures currently in use by two of the most commonly used operating systems, Linux and Microsoft Windows. As an example their IEEE1394 driver stacks are examined. IEEE1394 is a high speed serial bus protocol that is gaining wide industry acceptance [1394ta, 2002]. The IEEE1394.1 standard, still in draft phase, defines IEEE1394 bridging [IEEE1394.1, 2002]. Requirements from this standard that bridge aware device drivers need to fulfil are also identified.

2.1 The Windows Driver Architecture

In 1980, Microsoft licensed the UNIX operating system from Bell labs, later releasing it as the XENIX operating system. With the first IBM PC, MS DOS version 1 was released in 1981. MS DOS version 1 had a similar driver architecture to UNIX systems based on XENIX [Deitel, 1990]. The difference between this and UNIX systems, was that the operating system came with built-in drivers for common devices. Device entries did not appear as file system nodes as they do on a UNIX system (see section 2.2). Instead reserved names were assigned to devices. For example, CON was the keyboard or screen, PRN the printer and AUX the serial ports. Applications could open these devices and obtain a handle to associated drivers as they would with file system nodes, and perform I/O to them. The operating system, transparent to applications, translated reserved device names to devices that its drivers managed.

MS DOS version 2 introduced the concept of loadable drivers. Since Microsoft had made the interface to its driver architecture open, this encouraged third party device manufacturers to produce new devices [Davis, 1983]. Drivers for these new devices could then be supplied by hardware manufacturers and be loaded/unloaded at runtime into the kernel, manually.

Later on, Windows 3.1 was released by Microsoft. It had support for many more devices and utilised an architecture based on MS DOS. With its later operating systems, Windows 95, 98 and NT, Microsoft introduced the Windows Driver Mode (WDM). The WDM came about because Microsoft wanted to make device drivers source code compatible with all of its new operating systems [Microsoft WDM, 2002]. Thus, the advantage of making drivers WDM compliant is that once created, a driver need only be recompiled before it is usable on any of Microsoft's later operating systems.

There are two types of Windows drivers, legacy and Plug and Play (PnP) drivers. The focus here is only on PnP drivers, as all drivers should be PnP drivers. PnP drivers are user friendly since very little effort is required from users to install them. Another benefit of making drivers PnP is that they get loaded by the operating system only when needed, thus they do not use up system resources needlessly. Legacy drivers were implemented for Microsoft's earlier operating systems and their architecture is outdated. The Windows Driver Model (WDM) is a standard model specified by Microsoft. WDM drivers are usable on all of Microsoft's recent operating systems (Windows 95 and later).

2.1.1 The WDM driver architecture

There are three classes of WDM drivers: filter, functional and bus drivers [Oney, 2001]. They form the stack illustrated in figure 2.1. In addition, WDM drivers must be PnP aware, support power management and Windows Management Instrumentation. Figure 2.1 shows the messages exchanged between the various driver layers. A standard structure called an I/O Request Packet (IRP) is used for communication. Whenever a request is made from an application to a driver, the I/O manager builds an IRP and passes it down to the driver, which processes it and when done, "completes" the IRP [Cant, 1999]. Not every IRP filters down to a bus driver. Some IRPs get handled by the layers above and are returned to the I/O manager from there. Hardware access to a device is done through a hardware abstraction layer (HAL).

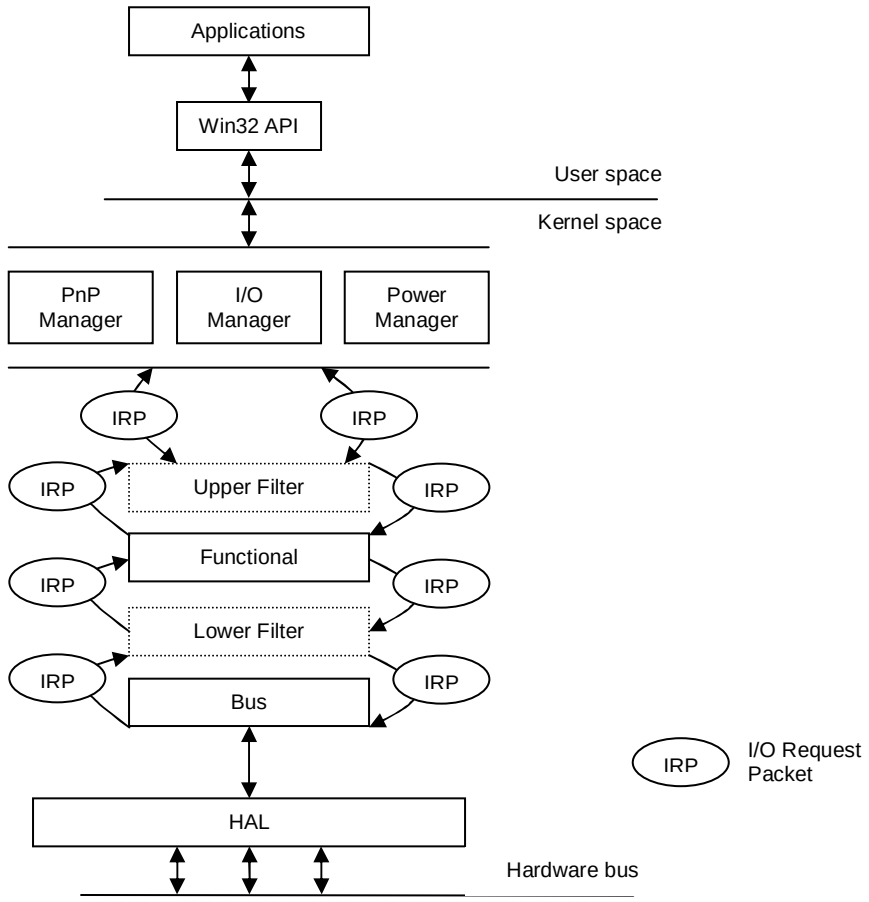


Figure 2.1 The WDM Driver Architecture

2.1.1.1 Bus Drivers

A bus driver enables the operation of a particular bus for which it was written. Microsoft supplies bus drivers for buses such as PCI, USB, IEEE1394 and SCSI. Bus drivers can also be supplied by hardware vendors. Bus drivers serialise accesses to a bus and are able to service requests from multiple clients. They keep track of devices that are attached to the particular bus they are managing.

2.1.1.2 Functional Drivers

A functional driver implements the functionality of a device e.g. I/O operations for a device. It services common I/O requests such as reads, writes and IOCTL calls. Every device driver requires a functional driver provided by the device manufacturer. Microsoft provides functional drivers for devices that have a standard interface, such as printers.

2.1.1.3 Filter Drivers

Filter drivers are optional drivers located between bus drivers and functional drivers (lower filter drivers) or above functional drivers (upper functional drivers). Their basic function is to apply some sort of transformation on data exchanged between bus and functional drivers, in the case of lower filter drivers, and between functional drivers and the application layer in the case of upper filter drivers as shown in figure 2.1.

2.1.1.4 Plug and Play

PnP aware drivers need to handle events such as insertion or removal of new devices and adjust their operation accordingly. Bus drivers for example keep a list of devices attached to their bus. When a new device is inserted into a bus the PnP manager sends a message to the bus driver. On receipt of a “new device” PnP message from the PnP manager a bus driver updates its list of devices. Similarly when a device is removed from a bus the PnP manager informs the bus driver which removes the device from its list. The WDM defines a number of PnP Messages e.g. start device, remove device, surprise removal. Oney, [Oney, 1999], and Cant, [Cant, 1999], provide a complete listing and discussion of each message.

2.1.1.5 Power Management

Similarly to PnP aware drivers, Power Management (PM) aware drivers handle PM messages that are sent to them by the power manager. The WDM is compliant with the ACPI (Advanced Configuration and Power Interface) specification [ACPI, 2002]. Device drivers can attain four power states [Oney, 1999]: fully on (D0), almost on (D1), almost off (D2) and fully off (D3). When a power state change is about to occur, the power manager sends power IRPs to all device drivers, each of which sends a reply stating whether or not they can operate in that state. The power manager can then decide whether or not to effect the power state change. This behaviour only works for software generated power state changes. User initiated state changes such as unplugging of a PC from a power source cannot be serviced by the power manager as the whole machine would move straight to state D3.

2.1.1.6 Windows Management Instrumentation

Window Management Instrumentation (WMI) implements the Web Based Enterprise Management (WBEM) standard. WMI enables system management and description in an enterprise network [Oney, 1999]. A driver can be configured, produce log data and generate events through the WMI interface for the benefit of user space applications. A driver that complies with WMI must register for notification of WMI events with the kernel at load time. Only then will it receive WMI IRPs. It must also deregister receipt of WMI IRPs at driver unloading time.

2.1.2 Classes of devices

Device classes serve to distinguish the different types of devices that are present on a machine at a particular point in time. Applications that require accesses to specific types of devices will ask the operating system to enumerate all available devices of a requested device class. For example, CD burning software will be interested in all devices that are in class CDROM. The class of a device is specified by a driver at driver load time. Several system defined device types such as sound driver, disk driver, network driver exist. A 16 bit value is used for representing device classes. Values 0-32766 are used for system defined classes. The driver writer may also define a custom device type using one of the values 32767-65535.

2.2 The Linux Driver Architecture

Linux is a clone of the UNIX operating system. It follows that the Linux operating system utilises a similar architecture to UNIX systems. UNIX operating systems view devices as file system nodes. Devices appear as special file nodes in a directory designated by convention to contain device file system node entries [Deitel, 1990]. The aim of representing devices as file system nodes is so that applications can access devices in a device independent manner [Massie, 1986],[Flynn *et al*, 1997]. Applications can still perform device dependent operations with a device I/O control operation. Devices are identified by major and minor numbers. A major number serves as an index to an array of drivers and a minor number is used to group similar physical devices [Deitel, 1990]. Two types of UNIX devices exist, char and block. Char device drivers manage devices that are accessed sequentially with no buffering, and Block device drivers manage

devices where random access is possible, and data is accessed in blocks. Buffering is also utilised in block device drivers. A block device must be mounted as a file system node for it to be accessible [Beck *et al*, 1998].

Drivers in Linux are represented as modules, which are pieces of code that extend the functionality of the Linux kernel [Rubini, 2001]. Modules can be layered as shown in figure 2.2. Communication between modules is achieved using function calls. At load time a module exports all functions it wants to make public to a symbol table that the Linux kernel maintains. These functions are then visible to all modules. Access to devices is done through a hardware abstraction layer (HAL) whose implementation depends on the hardware platform that the kernel is compiled for, e.g. x86 or SPARC.

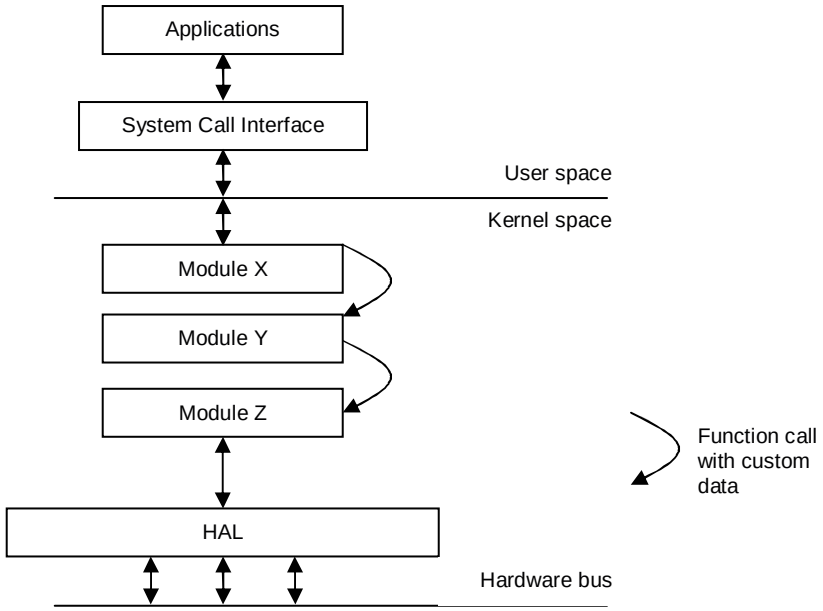


Figure 2.2 The Linux Driver Architecture

2.2.1 Classes of devices

Three classes of devices exist in Linux. Character devices, Block devices and Network devices. Character devices are accessible as byte streams usually through the read and write system calls. Block devices are accessed in multiples of blocks. The kernel exposes the same interface to character and block devices for applications. Network devices are

accessed differently from character and block devices because they involve data exchanges between remote entities. Thus they are accessible though a different interface e.g. the socket interface with calls such as *send*, *recv*, *listen* and *connect*. Linux retains much of the UNIX architecture, the difference being that char device nodes corresponding to block devices have to be created in UNIX systems, whereas in Linux, the Virtual File System (VFS) interface blurs the distinction between char and block devices [Beck *et al*, 1998]. Linux also introduces a third type of device i.e. a Network device.

2.2.2 Plug and Play in Linux

As of kernel 2.4 Linux does not yet have a standardised PnP system [Laywer, 2001]. A facility exists whereby a driver can load another driver at runtime. A driver for a particular bus e.g. USB when notified by a USB controller of a new device's arrival, may load a driver that it knows can service the new device. Alternatively, it may execute a user space application supplying the device's hardware ID as an argument. The application can then load an appropriate driver for the device.

2.2.3 Power Management in Linux

Power Management support for devices compliant with the APM (Advanced Power Management) standard [APM, 2002] is available in the Linux Kernel. It is mostly aimed at laptop computers that use batteries. Support for the ACPI standard, which is the successor to APM is also available. The kernel documentation for version 2.4 states that ACPI support is still under development. The ACPI driver supports the sleep, resume and power off modes [Glenn, 2001].

2.3 The Linux and Windows Driver Architectures Compared

As can be seen in figures 2.1 and 2.2, a number of similarities exist between the two operating systems. On both systems, drivers are modular components that extend the functionality of the kernel. Communication between driver layers in Windows is through the use of I/O request packets (IRPs) supplied as arguments to standard functions, whereas in Linux, function calls with parameters customized to a particular driver are used. Windows has separate kernel components that manage PnP, I/O and Power. These

components send messages to drivers using IRPs. In Linux there is no clear distinction between these components. The kernel may have modules loaded that implement this functionality but their interface to drivers is not clearly specified. Once data is passed by the kernel to a driver, this data may be shared with other drivers in the stack through an interface specific to that driver. In both environments, hardware access through a HAL interface is implemented for the specific platform the kernel is compiled for.

2.4 IEEE 1394

IEEE1394 is a high-speed serial bus protocol that allows data transfers at speeds of 100, 200 and 400 Mbits per second as specified in the IEEE1394-1995 and IEEE1394a standards, and at 800 and 1600Mbits per second as specified by the recently approved IEEE1394b standard. Two modes of data transfer are available. Asynchronous mode which offers guaranteed packet delivery and isochronous mode which offers guaranteed bandwidth. Up eighty percent of the available bus bandwidth may be allocated to isochronous transfers and the rest gets used for asynchronous transfers [Anderson, 1999]. In the absence of isochronous traffic, asynchronous transfers may use as much bandwidth as available. Isochronous mode is especially of interest to digital audio and video applications. IEEE1394 devices are plug and play aware. In other words every time a new IEEE1394 node is inserted into or removed from a bus, device enumeration takes place.

2.4.1 Current Bus Usage

Currently available consumer products are used in a single IEEE1394 bus setup as shown in Figure 2.4.1a. Examples of these are storage devices such as high capacity hard drives, digital video cameras and set top boxes that allow conversion of analogue TV signals into MPEG streams to be broadcast on an IEEE1394 home network.

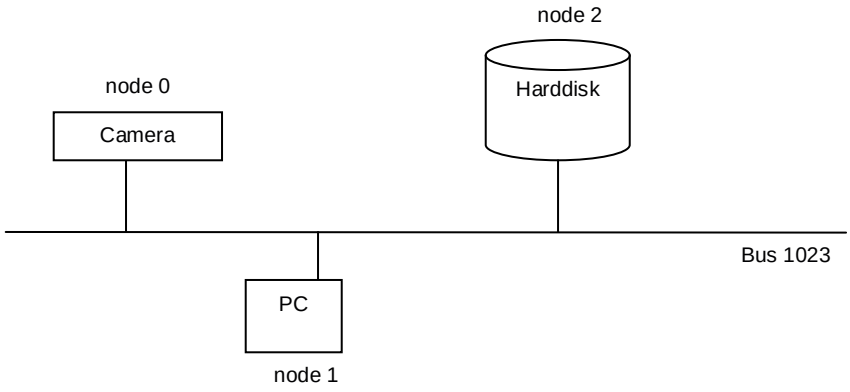


Figure 2.4.1a Example IEEE1394 single bus setup

Figure 2.4.1b illustrates the node addressing scheme specified by the IEEE1394 standard. The maximum number of nodes permitted on a single bus is 64 (2^6). Since node number 63 is used as a broadcast node address only 63 nodes can be connected together in a practical setup. The 48 bit node addressing scheme ensures that a node has a private address space of 2^48 bytes (256 terabytes).

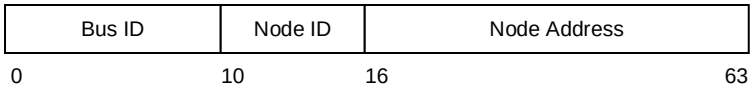


Figure 2.4.1b IEEE1394 node addressing

2.4.2 Increasing Node Count on the Network

The node count can be increased by using multiple buses joined together by a bridge. The job of the bridge is to grab packets and route those destined for foreign buses. Bus IDs as, specified by the IEEE1394 standard, range from 0 to 1023 (see figure 2.4.1b). Bus number 1023 is reserved for the local bus so practically 1023 buses can be joined together. Figure 2.4.2 illustrates an IEEE1394 multi-bus network. A draft IEEE1394 specification (IEEE1394.1) defines the requirements for IEEE1394 bridges and bridge aware devices. Currently there are no IEEE1394.1 capable bridges available on the market.

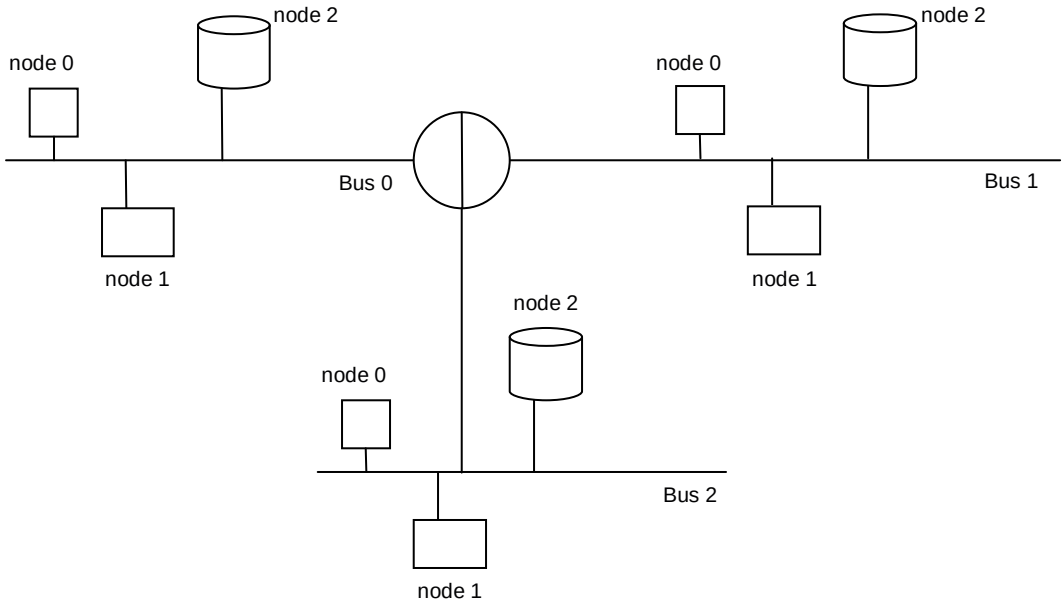


Figure 2.4.2 An example IEEE1394 multi-bus network

2.4.3 Current Windows and Linux IEEE1394 drivers

Both Windows and Linux currently have IEEE1394 driver implementations. The Windows driver has a fully documented API supported by Microsoft [Microsoft1394]. The Linux driver is still under development and is not fully documented [linux1394].

2.4.3.1 The Microsoft Windows IEEE1394 Driver Stack

The current Windows IEEE1394 driver stack is shown in figure 2.4.3.1. At the lower end of the stack there is a port driver, known as a host controller. In the WDM, this driver would be a bus driver. Port drivers are specific to the type of chip the IEEE1394 interface card contains e.g. OHCI, PCILynx or Adaptec. Above the port driver, there is a functional driver, called 1394Bus, that performs the basic functions of an IEEE1394 driver, such as sending and receiving packets. Specialised client drivers sit on top of this layer. These can be device specific to each IEEE1394 device e.g. a digital camera or a speaker. An example is the 61883 client driver. 61883 is communication and control interface standard specified by IEEE, used for IEEE1394 audio and video transmission. The 61883 client driver implements 61883-1 (the general 61883 standard) and 61883-6 (the standard for audio transmission over IEEE1394).

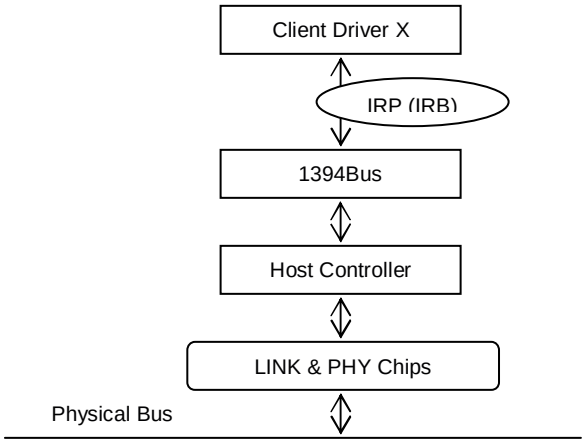


Figure 2.4.3.1 The Windows IEEE1394 Driver Stack

2.4.3.2 The Linux IEEE1394 Driver Stack

The current IEEE1394 Linux driver stack is shown in figure 2.4.3.2. The host controller allows access to the physical IEEE1394 card's PHY and LINK chips and is specific to the chip type present on the card. Ieee1394 is a core driver that provides a uniform view of the IEEE1394 bus through a Highlevel interface.

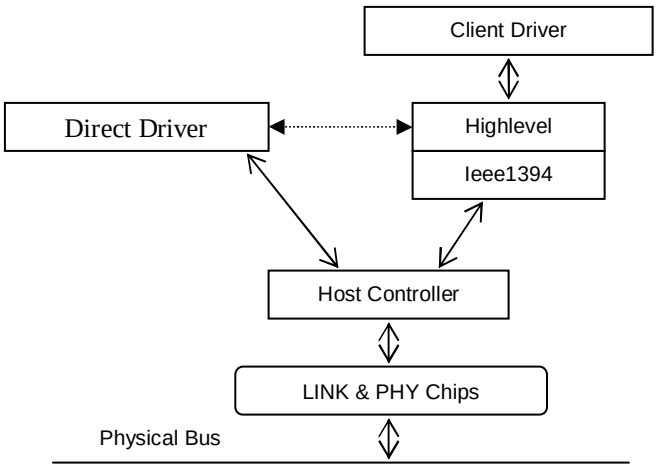


Figure 2.4.3.2 The Linux IEEE1394 Driver Stack

Above the core any number of specialised client drivers can exist. An example client driver is raw1394 which allows generic IEEE1394 operations. Drivers, named “Direct Drivers” in figure 2.4.3.2, can also bypass the Ieee1394 layer and interact with the host controller themselves. At load time every IEEE1394 driver needs to register with the Highlevel interface to get references to available host controllers thus a connection is shown between the direct driver and the Highlevel interface in the figure.

2.4.3.3 Differences between the two Stacks

The Windows and Linux IEEE1394 stacks appear to be very similar as can be seen from figures 2.4.3.1 and 2.4.3.2. A difference exists in the way data is exchanged by the various layers. In Windows, a standard structure called an I/O request block (IRB) attached to an IRP, is used for data exchange between the layers. In Linux any customized structure can be exchanged by the different layers. In Windows, there is limited direct access to the port driver as no API exists for it and most requests have to go through the 1394bus layer, which has a well defined API. In Linux due to the open source nature of the operating system, full access to the host controller’s services is possible since its interfaces are publicly known. The Windows stack is well documented whereas there is little or no documentation for the Linux stack.

An important difference between the two stacks is that the Windows stack is commercially usable and supported by Microsoft [Microsoft1394, 2002] whereas the Linux stack is still tagged as experimental. Support for the Linux stack is available from individuals working on the project [Linux1394, 2002].

2.4.3.4 IEEE1394.1 support

Microsoft’s knowledge base article Q319197 [Microsoft KB Q319197, 2002] states that their driver stack does not support IEEE1394 bridging, and investigations of the Linux stack also show that it too does not support IEEE1394 bridging. Adding IEEE1394.1 functionality in the current IEEE1394 drivers will allow existing IEEE1394 devices to operate in bridged environments. This will enable the setting up of large IEEE1394 networks with the node count not exceeding 1023x63(64449).

2.4.4 IEEE1394.1 Bridge Awareness

IEEE1394 bridges are responsible for joining two IEEE1394 buses. They manage a FIFO queue for isochronous and asynchronous packets. They also maintain a routing table used for selective forwarding of packets [IEEE1394.1, 2002]. In a bridged environment, identification of local nodes and nodes on foreign buses is performed using global node IDs. These global node IDs are managed by bridges. Bridges configure themselves for asynchronous transfers, but multi-bus isochronous streams have to be managed. Stream management is discussed in section 2.4.4.6.

Enabling IEEE1394.1 bridging support in the Windows driver would require getting source code for the driver stack from Microsoft. Both the hardware interface driver, host controller and 1394bus driver in figure 2.4.3.1, would have to be modified. Since Microsoft's operating system components are proprietary, this is not possible. Microsoft does have a team working on IEEE1394 drivers and it is left to them to add IEEE1394.1 bridging capability to their driver [Microsoft KB Q319197, 2002]. Otherwise independent reimplementation of both the host controller driver and 1394bus drivers is required.

One of the major advantages that Linux has over Windows is that it is developed open source. Both the Kernel and IEEE1394 driver sources are available. Modifications can be made to the Linux driver to make it IEEE1394.1 bridge aware. Linux's open source nature can be problematic for private companies that want to use its stack in their products since they would be using driver code which is released under the GNU Public License (GPL) [GPL, 2002]. The basic requirement of the GPL is: release of full sources with any implementation that uses code that was previously released under the GPL. There is no restriction on sale and distribution of products as long as clients are provided source code from which they themselves can produce binaries.

Section 9 of the draft IEEE1394.1 standard as of April 2002 sets a number of requirements to be fulfilled by bridge aware drivers. The following sections present those requirements.

2.4.4.1 Indication of bridge awareness

All bridge aware devices must set bit number 28 of the bus information block in their configuration ROM (a place where information such as node capabilities is kept), memory address CSR_REGISTER_BASE + 0x408, to 1. This makes it possible for bridge aware nodes to be identified by bridges as such. (IEEE1394.1 Section 9.2.1)

2.4.4.2 Remote timeout

Timeouts for local and remote transactions are different. Remote transactions may be delayed depending on network topology. Remote timeout values are obtained by using a bridge management message sent to a target bridge. IEEE1394.1 section 6.6.1 defines the TIMEOUT bridge management message which is used for this purpose.

2.4.4.3 Bus reset and quarantine

All bridge aware nodes require the implementation of the QUARANTINE register (IEEE1394.1 section 9.2.3). After a bus reset they are required to remain silent until the orphan bit (bit 0) in this register is cleared. This is done by a bridge when normal bus usage is ready to resume. A quarantine period is necessary after a bus reset so that bridges can manage changes in network topology while all non-bridge nodes keep silent, reducing the load on the network. A node should set bit zero of its quarantine register to one after every bus reset when it detects the presence of a bridge on the bus. This can be done when self id packets are received and the presence of a bridge is detected. IEEE1394.1 section 5.1.1 of the IEEE1394.1 specification defines new bits in self ID packets that are set to indicate whether a particular node is a bridge or not.

2.4.4.4 Differentiation of remote and local packets

A bridge aware device should only be able to receive packets addressed to it. For example, packets with a destination address of node number 1 and bus ID 10 should not be picked up by a node with node number 1 and bus ID 1023 (the local bus). Node addressing is discussed in section 2.4.1.

2.4.4.5 Bridge management messages

Bridge aware devices are required to send and receive bridge management messages to and from bridges. One such message is the JOIN message which is used to create a stream connection between a transmitter node and a receiver node. Bridge management messages are sent using block write requests that contain a data payload conformant with the bridge management message format (IEEE1394.1 section 6.6), addressed to a node's or bridge's MESSAGE_REQUEST and MESSAGE_RESPONSE registers. These registers are a set of registers that must be implemented by bridge aware nodes.

2.4.4.6 Stream management

Three different entities participate in stream management, namely the controller, talker and listener as shown in figure 2.4.4.6. A stream path is established between a talker and one or more listeners. The resources used by the stream path are managed by the bridges that lie along the path. The talker could be a device such as a set top box that transmits an MPEG2 stream and the listeners could be devices that pick up the stream and display it e.g. an HDTV. The controller is an application that manages connections between talkers and listeners.

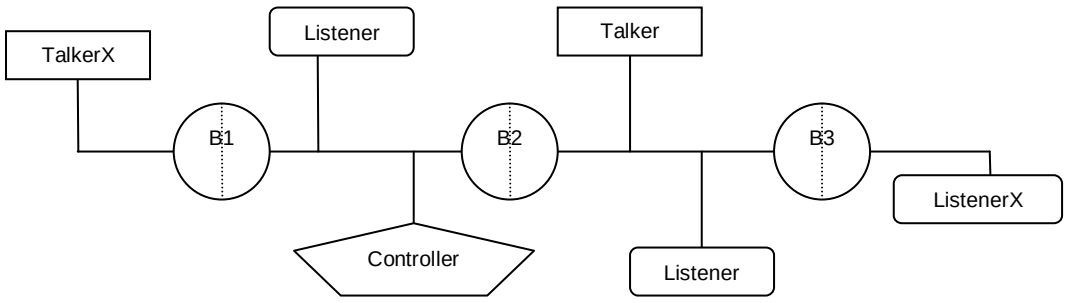


Figure 2.4.4.6 An IEEE1394 Multi-bus network. B1, B2 and B3 are bus bridges.

In order to create a stream connection between talkerX and listenerX, the controller shown in figure 2.4.4.6 sends a JOIN request addressed to listenerX. The packet containing the JOIN request would have its snarf field (see section 2.4.4.7 for explanation of the snarf field) set so that only bridge B3 can intercept it. B3 then reserves resources

i.e. channel and bandwidth for the bus on which listenerX is on and sends a JOIN request to TalkerX. The packet containing this JOIN request will have its snarf field setup so that it gets intercepted by B2 then B1. B2, then B1 will allocate resources for the stream as the packet travels towards TalkerX. A stream would then have been established between TalkerX and ListenerX.

2.4.4.7 The new Snarf field

A new snarf field is defined in block write packet headers that indicates which bridges may intercept a block write request to a MESSAGE_REQUEST/RESPONSE register (IEEE1394.1 section 6.5). Normally a bridge grabs all packets on the local bus and forwards those destined for remote buses without examining a packet‘s contents. However if it sees that the packet’s snarf field is set in such a way that it has to process the packet’s contents as a bridge command, it will treat it as such. Otherwise it will forward the packet. Block write handlers in the current driver need to be modified to enable the setting of the snarf field by bridge aware nodes.

2.5 Summary

This section has presented a comparison of the current device driver architectures employed by two of the most widely used operating systems, namely Microsoft Windows and Linux. Both architectures permit the creation of drivers as modular components that are added to the operating system kernel at runtime and enable the breaking up of driver functions into different components that can have a layered architecture. PnP and power management are an integral part of the Windows architecture while the Linux architecture has not yet fully integrated them.

As an example of a major device driver implementation the IEEE1394 driver stacks of both operating systems were compared. IEEE1394 is a high speed serial bus standard that will be used in home and office networks of the future. The IEEE1394 stacks on both operating systems are still in development. The requirements for support of IEEE1394.1 by the stacks, which standardises bridging of IEEE1394 buses, were also presented.

3 Designing Device Drivers

Device drivers are components that are added to a kernel to extend its functionality. They must be designed with stability in mind. Use of system resources such as memory, I/O ports and IRQ lines must be managed carefully. Since device driver code executes in supervisor mode (code that executes in supervisor mode has unrestricted access to system resources), care must be taken to make sure that the code executes appropriately without interfering with work carried out by other parts of the kernel. This section presents issues that must be considered when designing efficient and well behaved device drivers. It also provides a look at current Windows and Linux IEEE1394 driver designs and outlines the essential components of a generic IEEE1394 client driver.

3.1 Memory Management

One of the most important resources that a device driver must manage is memory. Memory requirements often change unpredictably during the life of a device driver. Maintaining the right size of memory requires knowledge of what sort of device the driver will be used for. Memory allocation issues, such as whether a driver must allocate all the memory it needs when it is first loaded up, or while servicing an I/O request must also be considered.

3.1.1 General Memory Usage

Both the Windows and Linux operating systems maintain two pools of memory. One pool always stays resident (non-paged) the other is swappable (paged). Non paged memory is necessary for operating system components that must be in main memory at all times. The page fault handler for example cannot be swapped out to disk because it is responsible for retrieving swapped out pages from a disk itself. In both environments, memory can be allocated dynamically as driver code is executing. It must be de-allocated when no longer needed. One of the most common causes of kernel instability is due to drivers not de-allocating memory they allocated, which leads to wasted memory. Similarly de-referencing a pointer without first allocating memory and initialising the pointer to point to the new piece of memory, often leads to kernel instability. Direct

access to user space memory addresses must also be done with care, to prevent users from maliciously crashing the kernel. Both operating systems provide routines for testing the validity of user space memory addresses.

3.1.2 Handling Fragmentation of Heap Memory

Drivers such as network drivers process large numbers of incoming data packets. These packets are usually kept in a queue. When the hardware signals the arrival of a data packet, the driver is responsible for allocating a chunk of memory to handle the request. If frequent memory allocation and de-allocation is performed by a driver it can lead to fragmentation of heap memory [Cant, 1999]. To solve this problem Windows and Linux provide lookaside list implementations. A lookaside list is a collection of fixed sized blocks of memory [Oney, 1999]. At driver load time a driver can allocate a lookaside list and allocate memory from this chunk when needed. At de-allocation time the freed memory is returned to the lookaside list. This ensures that the next request for memory from the lookaside list is more likely to be satisfied than if it were to be requested from the main heap.

3.1.3 Use of Data Structures

Both operating systems provide data structures for representing lists, stacks and queues that are implemented as linked lists. System implementations of these data structures should be used instead of custom ones, as this helps reduce development time. Only the essential operations such as adding and removing items to and from the above structures are implemented. Specialised operations such as searching for specific items in lists must be implemented by the driver writer.

3.1.4 Synchronising Accesses to Shared Memory

Whenever access is made to shared memory locations, synchronisation primitives must be used. For example if an item is added to the tail of a queue, this operation must be protected using a multi-processor safe synchronisation primitive such as a spin lock. Both operating systems provide implementations of spin locks, semaphores, mutexes and signalling.

3.1.5 Modular Design

Creating drivers as modular components that get loaded at runtime ensures that system memory gets used only when needed. If drivers for every possible hardware device were part of the kernel, the size of the kernel would end up being very large and use more system memory than necessary. Instead, Windows loads drivers when it identifies devices through PnP and Linux allows manual loading of drivers at runtime.

3.2 Functional Design

Driver code consists of functions that perform a specific task. A function should not perform more than one task. This simplifies modifications of the code later on in the driver's life cycle. Under Windows, drivers can be written using an object orientated language such as C++. On that platform object orientated design techniques such as producing class diagrams and sequence diagrams can be employed, as well as representation of driver components using object templates (classes).

3.2.1 Re-entrant code

Functions must be written to be re-entrant. Re-entrant code is function code that can be executed by multiple threads at the same time, without the danger of race conditions occurring. No part of a function's code should contain shared data, if possible, since function code may be executing in multiple threads at the same time. This includes the use of static and global variables. Both Windows and Linux provide a facility whereby they allow a driver to specify a location for shared global data. This location is then passed to drivers when the kernel calls a driver's functions to perform some task. In this way data can be shared by all functions. Two separate functions may still be called with the same data, in which case use of a synchronisation structure such as a semaphore is necessary.

3.2.2 Recursion

Care must be taken when using recursion in code, as the kernel may run out of stack space. Very deep recursive calls require a lot of system resources, which are limited at the kernel level.

3.3 Hardware management

A device driver is created to control a piece of hardware so that it can be used by the operating system and applications that run on the operating system. The operating system contains device drivers for every piece of hardware present on a machine. It must provide facilities to allow devices to ask for CPU time, as well as allowing a device driver to communicate with a piece of hardware.

3.3.1 Input Output (I/O) Ports and I/O Memory

I/O ports and I/O mapped memory addresses are used to read and write data to a device. A piece of hardware uses specific I/O addresses, which may not be shared by two devices. Depending on the CPU architecture there might be separate address spaces for hardware devices and software or a single address space. Accesses to devices can be I/O mapped or memory mapped. Figure 3.3.1 shows how I/O addressing is performed on Windows [Oney, 1999]. Once a read or write request has been made by a CPU, the device then maps the requested address to a bus address space. The figure is also applicable to Linux, as it supports both methods of accessing devices.

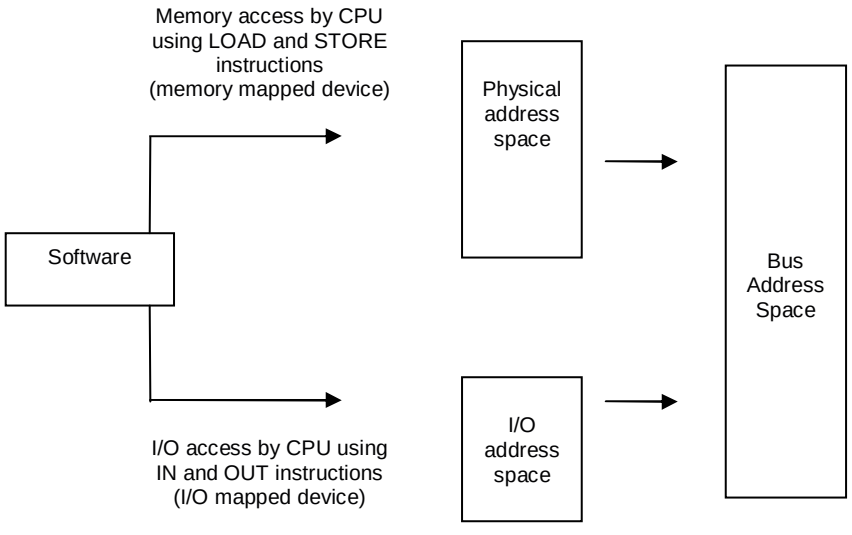


Figure 3.3.1 I/O addressing on Windows

Specialised CPU instructions are used to read and write to and from I/O mapped devices. Memory mapped devices do not require the use of specialized device I/O CPU instructions when performing reads and writes, instead ordinary memory access instructions such as MOV are used. Generally the use of assembly language instructions is not necessary, as both Windows and Linux provide HAL routines to perform device I/O operations, which are specific to the particular CPU architecture on which the kernel is running.

3.3.2 Interrupt Requests (IRQs)

Hardware present on devices can raise interrupts to inform a device driver of the occurrence of some event, such as the arrival of new data. The interrupt is sent to the CPU, which in response executes an interrupt handler. Interrupt handlers are installed by device drivers, usually at load time. Interrupt request lines (IRQs) available to a CPU are limited. Two devices may not use the same interrupt, as the CPU would not be able to tell which device raised the interrupt. During the time that an interrupt handler is being executed, all other interrupts that have lower priority than the current interrupt are disabled so the interrupt handler must return as soon as possible. On both Windows and Linux any form of waiting such as using semaphores is prohibited in an interrupt handler. Similarly, access to user space addresses is not possible as the interrupt handler does not execute in the user's process context nor will the operating system be able to handle page faults.

3.3.3 Direct Memory Access (DMA)

When a device performs the reading and writing of data, it is necessary to interrupt the CPU to service I/O requests, which may delay the execution of other tasks. For devices that frequently read and write large amounts of data an alternate way of performing I/O operations exists whereby a device may perform I/O directly to system memory, termed direct memory access (DMA), thereby freeing the CPU for other system tasks. Rubini [Rubini *et al*, 2001] identifies two ways in which DMA can occur. The first is software initiated, through read and write requests, and the second is device initiated by raising an interrupt. Figure 3.3.3a and 3.3.3b illustrate the two ways.

With the software initiated approach a process makes a read request, the device driver puts the data requesting thread to sleep, builds a DMA buffer, installs a DMA interrupt handler and initiates the DMA request. The hardware transfers data to the DMA buffer and on completion raises an interrupt. The interrupt handler wakes the data requesting thread which will pass the data from the DMA buffer to the data requester.

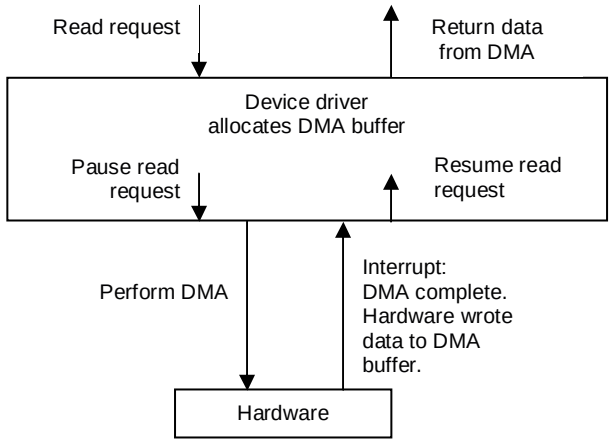


Figure 3.3.3a Performing a read operation using DMA, an example of software initiated DMA.

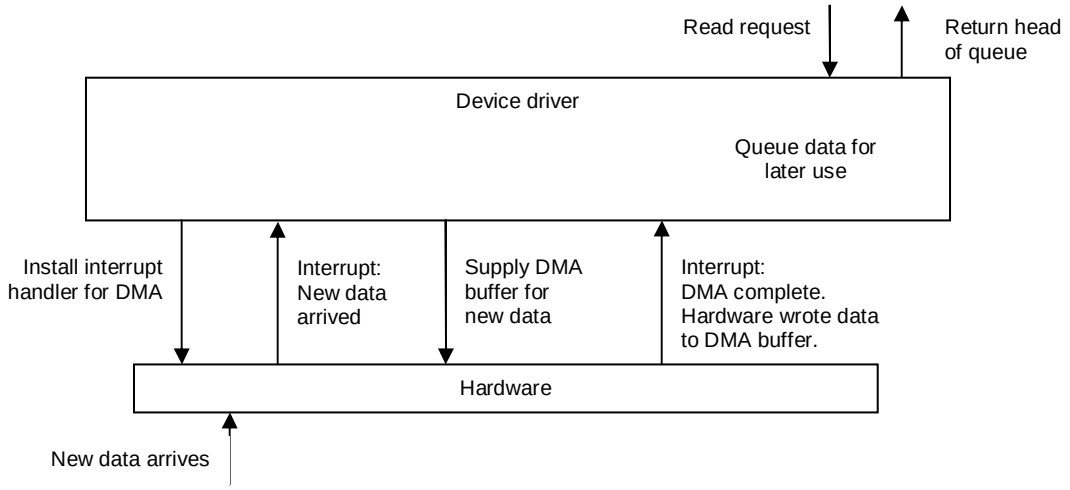


Figure 3.3.3b A device pushing data to DMA buffers, an example of hardware initiated DMA.

The device driver installs a DMA interrupt handler. When new data becomes available to the hardware, it raises an interrupt to inform the device driver of the arrival of new data. The driver supplies a DMA buffer to the hardware which transfers data to the buffer and raises an interrupt to signal data transfer completion. The driver's interrupt handler then queues the new data. When a request is made to the driver for data it removes data from its queue and passes it to the requestor.

3.4 Layered Driver Design

A single device can be constructed such that it performs all actions required to communicate with and control a device. Creating this kind of device driver is not recommended as it often leads to complications such as the driver code being difficult to understand for people other than those that created it. Operating systems such as Windows and Linux provide standard drivers on top of which other drivers can be layered.

Writing a single monolithic driver will duplicate functionality already available from these drivers. The monolithic driver may not be able to take advantages of future operating system upgrades. Layered design involves breaking parts of a device driver into separate components. This approach is encouraged by the Windows driver model, which specifies the breaking up of drivers into bus, functional and filter drivers as discussed in Chapter 2.

3.5 Windows and Linux IEEE1394 Driver Stack Designs

IEEE1394 is a fairly new standard that is only getting widespread industry usage now. It offers a high speed serial bus with transfer speeds of 100, 200 and 400 Mbps second. Theoretically, at the maximum speed, it is possible to transfer data at a rate of 50 MB per second compared to 100Mbps Ethernet which has a theoretical transfer rate of 12.5 MB per second. Chapter 2 section 2.4 discusses IEEE1394 in more detail.

There are a number of protocols that make use of the IEEE1394 bus. These include IP over 1394, which specifies IP networking requirements over a 1394bus, Serial Bus Protocol 2 (sbp2) which transports SCSI over IEEE1394 [Lohmeyer, 2002], 61883 which

specifies communication and control protocols [Microsoft DDK, 2002] (Table 3.5 lists the various 61883 protocols), and the new protocol still in draft phase, IEEE1394.1, which defines bridging of multiple IEEE1394 busses allowing up to 64K nodes on the bus instead of the current 63 nodes.

IEC 61883-1	General
IEC 61883-2	SD-VCR Data Transmission
IEC-61883-3	HD-VCR Data Transmission
IEC-61883-5	MPEG2-TS Data Transmission SDL-DVCR Data Transmission
IEC-61883-6	Audio and Music Data Transmission Protocol

Table 3.5 The various IEEE 61883 specifications

The driver stacks implemented by Windows and Linux are still in a state of development as new additions are still being made to the IEEE1394 specification suite, such as the IEEE1394.1 bridging standard. This section presents the Windows and Linux IEEE1394 Driver stack designs as they appear in their current state.

3.5.1 The Windows IEEE1394 Driver Stack

The Windows IEEE1394 stack is shown in figure 3.5.1. Chapter 2 presented the Windows driver model (WDM), where it was pointed out that the WDM contains three layers of drivers: functional, bus and filter. In figure 3.5.1, there is a functional driver consisting of a root FDO that manages all system drivers. Below it there is a bus driver for the PCI bus, which consists of a root PDO (Physical Device Object) and a PCI FDO (Functional Device Object). An FDO is an instance of a device driver that gets created at driver load time. At FDO creation time a driver object that contains all driver implemented standard routines, such as read and write, is associated with an FDO. When a client obtains a handle to that FDO and directs I/O requests towards it, IRPs for that I/O request will be passed to the routines implemented by the driver object along with a reference to the FDO. One of the fields contained in the FDO is called *DeviceExtension*, where global driver data is stored.

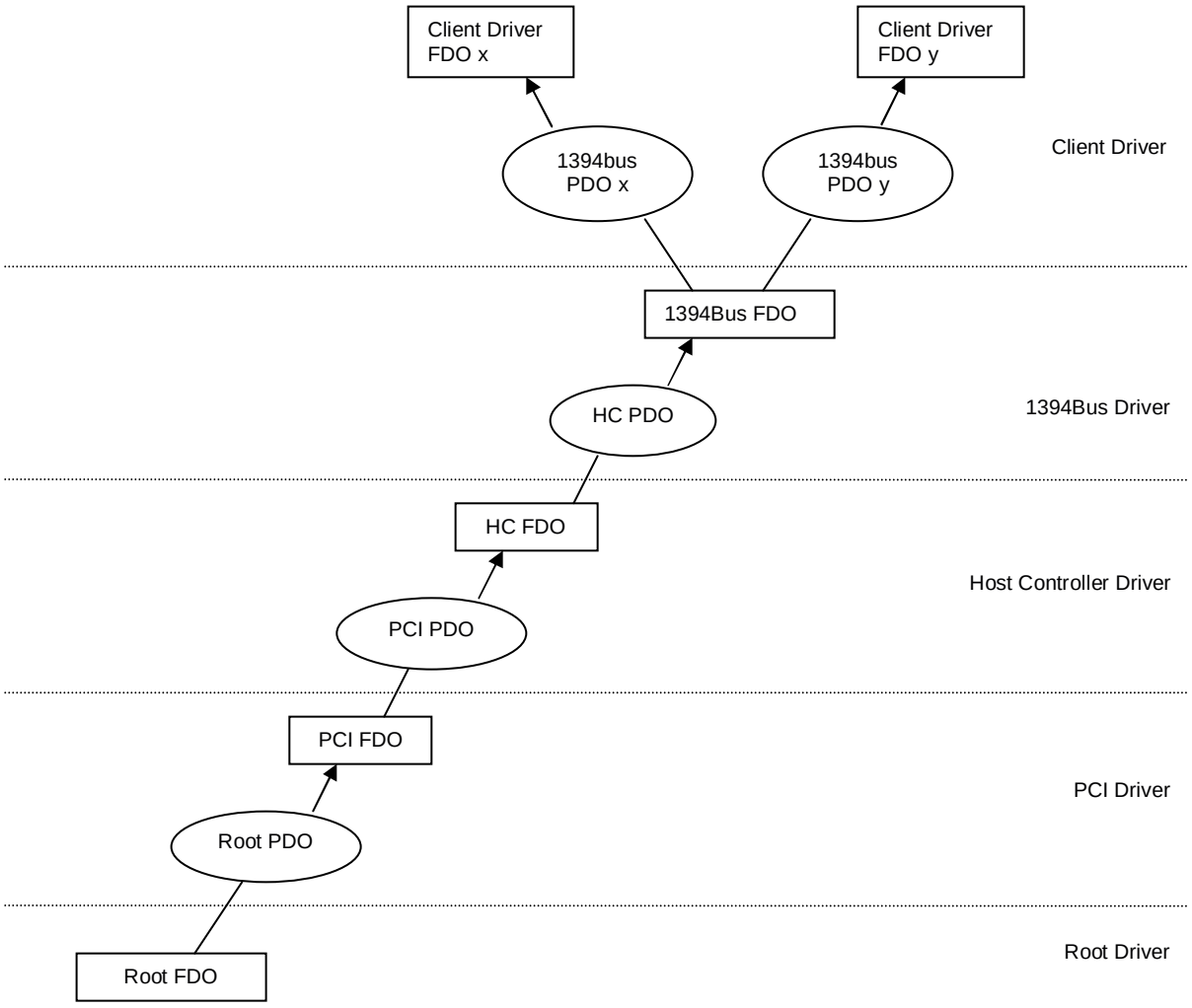


Figure 3.5.1 Windows IEEE1394 Stack

A PDO is a device object that is below the current FDO in a stack of device objects. For example in figure 3.5.1 there is a stack of devices objects i.e. root PDO->PCI PDO->HC PDO->1394bus PDO-> client driver FDO (a PDO and FDO for a driver are the same objects except for drivers that manage multiple device objects). If the current device object is the client FDO then the PDO below it would be the 1394bus PDO. Device objects are called FDOs when they exist within a driver where they were created and they are called PDOs when they are used outside of a driver where they were created, as happens for the 1394bus FDO. In figure 3.5.1, the 1394bus FDO is created in the 1394Bus driver. For each new IEEE1394 device, the 1394bus driver creates a new device

object and that device object, now called a PDO, is made available to the client driver so that the client driver's FDO can be stacked on top of this PDO.

As illustrated in figure 3.5.1, each time a new layer is added to the driver stack, an FDO is created for the current driver. A PDO is also supplied by the layer on top of which the current device object will be stacked, to enable the FDO to pass IRP's down to that PDO. From figure 3.5.1, the previous layer's FDO, and the PDO passed to the next layer may not be the same e.g. in the figure the 1394bus FDO and 1394bus PDO are not the same. IRPs sent to the 1394bus PDO are managed by the 1394bus driver, since its driver routines were used to create the 1394bus PDO. One of the functions of a bus driver is to manage child PDOs. For example, the 1394bus driver contains an FDO which manages the IEEE1394 bus. When a new node is inserted into the bus the 1394bus driver creates a new child PDO for that node and requests the PnP manager to load a client driver for it. When the client driver gets loaded it creates its own FDO and stacks that FDO on top of the new PDO created by the 1394bus driver.

The stack shown in figure 3.51 is built up for a machine that has a PCI bus, an IEEE1394 bus and a device for which the client driver was written. For multiple devices, there would be multiple 1394bus PDOs created and managed by the IEEE1394 bus driver. At FDO creation time a reference count is incremented within the PDO below, so the device object one layer below cannot be unloaded until this reference count decreases to zero.

3.5.2 The Linux IEEE1394 Driver Stack

In Linux there is no counterpart to the PDO and FDO classification that Windows uses. The Linux driver does not automatically build up a stack. At most, only the PCI driver is loaded, which will identify the IEEE1394 bus but will not build the stack shown in figure 3.5.2. The dotted box signifies that the host controller and client driver layers are built in response to user action i.e. loading of an appropriate driver for an IEEE1394 compliant device. Similarly to Windows, whenever a new layer is added a reference count is incremented on the layer below. This means that the layer below does not get unloaded while the new layer is still in use, as this could lead to system instability.

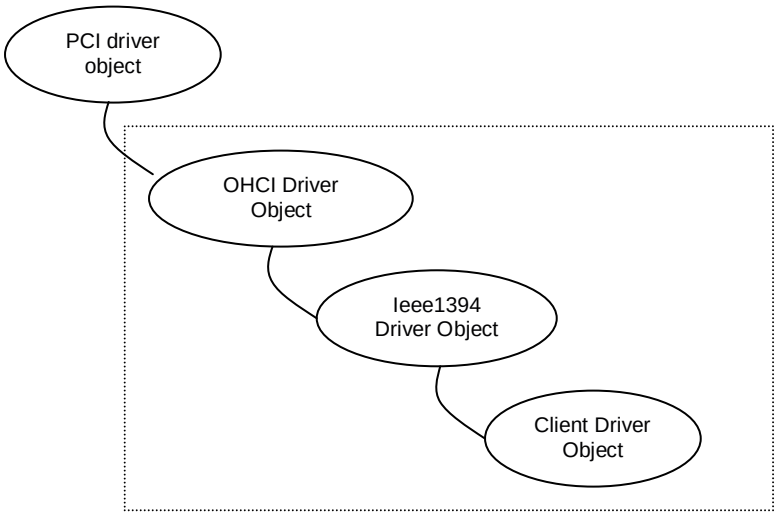


Figure 3.5.2 The Linux IEEE1394 Stack

3.5.3 Designing IEEE1394 Client Drivers

On both Windows and Linux, bus and host controller drivers already exist. The most frequent need for IEEE1394 driver development is at the client driver level. A driver for a specific device, such as 61883-6 protocol aware digital audio device, or a generic driver that performs all IEEE1394 operations can be created. The advantage of creating a generic driver is that specialised operations such as those for 61883-6 audio transmission can be done at the application level by using an application level API. This eliminates the need for kernel level driver development. For some implementation approaches, such as the creation of virtual device drivers e.g. a sound driver that converts audio data to 61883-6 format and outputs to an IEEE1394 bus, a kernel driver is preferred.

3.5.4 A Generic IEEE1394 Client Driver

Generic IEEE1394 client drivers should support the standard set of IEEE1394 operations, namely reads, writes and lock operations. In addition to read, write and lock operations a driver would need to be able to perform isochronous operations such as listening and talking on a channel. It should also be able to handle incoming asynchronous transactions for specified addresses. Given below is a listing of operations that a generic IEEE1394 driver should perform.

Asynchronous Read	*Register Asynchronous Listening*
Asynchronous Write	*Deregister Asynchronous Listening*
Asynchronous Lock	*Register Bus Reset Handler*
Asynchronous Listen	*Generate Soft Bus Reset*
Isochronous Listen	*Obtain Bus Information (e.g. node count)*
Isochronous Talk	*Configuration ROM manipulation*
Asynchronous Streaming	*Pinging of nodes (network troubleshooting)*

3.5.4.1 Read, Write and Lock operations

Reads and writes can be quadlet (4 bytes) sized or block sized. A lock operation is a special type of read and write transaction. A read would normally get the current value from an address, while a write would modify the current value of an address. With a lock transaction, operations such as additions and masked swaps can be performed on a value stored at an address. The bus drivers of both Windows and Linux support these operations.

3.5.4.2 Isochronous Talk and Listen

These operations allow the streaming of bandwidth dependent data, e.g. audio and video over an IEEE1394 bus. There are 64 channels available with channel 63 being allocated as the broadcast channel. To be able to receive data from a channel, an isochronous listen operation must be performed. Similarly, to be able to transmit data on a channel an isochronous talk operation must be performed. Both these operations are supported by the current Windows and Linux drivers.

3.5.4.3 Asynchronous Streaming

Asynchronous write transactions sent on an isochronous channel during the asynchronous phase of the bus are termed asynchronous streams [Microsoft1394, 2002]. Both the Windows bus driver and Linux core driver provide asynchronous streaming support. Asynchronous streams are received using normal isochronous listen operations. Asynchronous stream packets are useful for transmitting broadcast packets. They are currently used in IP over 1394 drivers.

3.5.4.4 Address Range Registration /De-Registration

A client driver performs asynchronous listen requests for specified address ranges by registering its intent to receive data for that address range with the bus driver. For example, for listening to data written to a node's FCP (Functional Control Protocol) register, which is memory mapped, the driver would register the FCP start address and 508 bytes of addresses thereafter with one request. The driver will then get notified, by the bus driver, whenever new data gets written to any of the registered addresses. Figure 3.5.4.4 illustrates the process. The driver should deregister any registered addresses when it no longer requires them. Both the current Windows and Linux drivers support the above operations. Once a client driver registers an address range, another client driver will not succeed on any attempts it makes to register the same address range since address range sharing is not supported by the drivers on either of the operating systems.

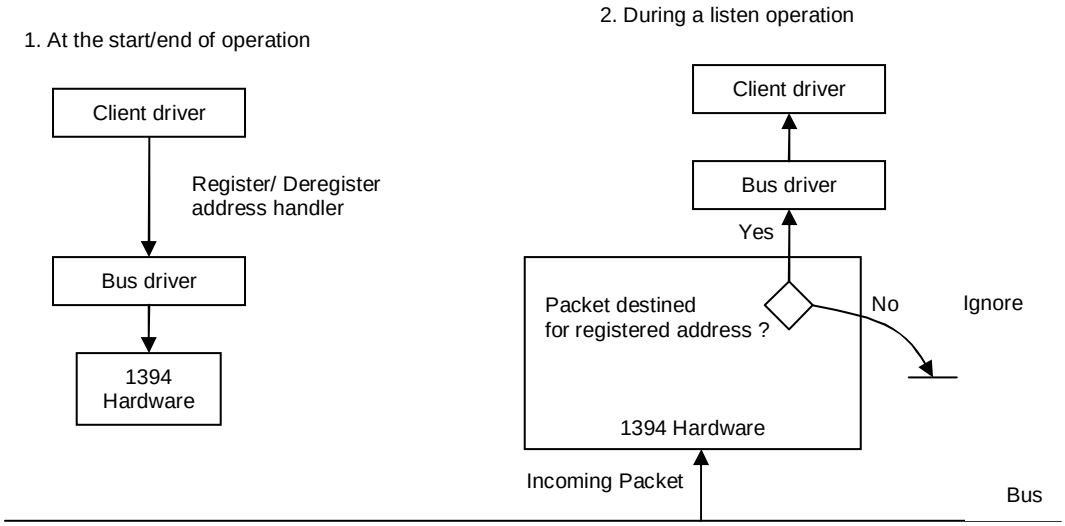


Figure 3.5.4.4 Asynchronous listen handling

3.5.4.5 Bus Reset Operations

A bus reset occurs whenever a new node is inserted into or removed from the bus, at which time node enumeration occurs. It can also be generated by the device driver (soft bus reset) to perform node re-enumeration. The client driver should notify its users

whenever a bus reset occurs. The client driver must register its intent to receive bus reset notifications with the bus driver.

3.5.4.6 Providing Bus Information

An operation to allow the gathering of bus information, such as the number of nodes currently on the bus, isochronous channel usage, the current isochronous resource manager (IRM) and host controller information, should be provided by a client driver. For example, knowing the number of nodes on the bus is critical to an application to be able to carry out node enumeration. Since node count starts at 0, an application can read the configuration ROMs of node 0 to node_count-1 and build a map of nodes currently on the bus. Support for this operation is not available in the current Windows bus driver so applications have to determine node count from the number of successful reads of configuration ROMs of all possible nodes (nodes 0 to 63) that can populate a bus.

3.5.4.7 Network Troubleshooting

Operations that allow the collection of network status information should also be provided. One such operation is the Ping operation specified in the IEEEP1394a standard, which sends a packet to a node, waits for a reply, and provides the round trip time to the requester, allowing it to ascertain current network speeds and remote node status.

3.5.4.8 Configuration Rom Manipulation

The configuration ROM provides critical information about a node, such as the node's unique identifier, its manufacturer, whether it is an IEEE1394 standard compliant node, and the capabilities of the node e.g. whether it implements an AVC subunit. Operations should be provided to allow manipulation of a node's configuration ROM.

3.5.4.9 IEEE1394 Bridge Operations

The IEEE1394.1 specification is still in draft phase, but from the current version of the specification a number of bridge-aware node requirements and operations have been identified. Modification of current drivers is needed to make them bridge aware. These modifications should be kept to the minimum and moved to the application level where possible. Modifications of the configuration ROM, as well as the bridge message request

and response registers have to be implemented at the driver level, but bridge command sending and receiving can be done at the application level.

Driver level changes are what prevent the implementation of bridge awareness to the Windows IEEE1394 drivers since they are proprietary drivers. Modifications that need to be made to an IEEE1394 node's configuration ROM to make it appear as a bridge aware node are necessary in a driver where the configuration ROM is initialised. Fields in packet headers used for communication with bridges are inaccessible to applications and need to be made accessible. Since no driver source code is available from Microsoft these modifications cannot be made to that operating system's drivers. Ideally no special bridge operations should need to be provided by client drivers, as all interactions with bridges can be done at the application level with a client driver that provides the operations presented in sections 3.5.4.1 to 3.5.4.8.

3.6 The Application Level

A device driver will ultimately be required to service requests from the application level. Presentation of an API for applications to utilise is necessary. A major problem that faces IEEE1394 drivers is the number of context switches performed while servicing application requests. IEEE1394 is a high speed serial bus specification. Data is transmitted at speeds approaching 40MB per second. If a context switch is done to notify an application of the arrival of new data for every new packet only a small fraction of the 40 MB per second transfer rate will materialise. Buffering can speed up transfer rates but a better solution is to employ DMA to user space buffers. Currently the Windows API supports DMA transfers to a user space buffer, while the Linux IEEE1394 API does not.
The API provided should hide all the details of interaction with the device driver from the application. This way the application will be decoupled from the driver and will only be dependent on the API as shown in figure 3.6. Future modification to the driver can then be performed with no modifications to the application.

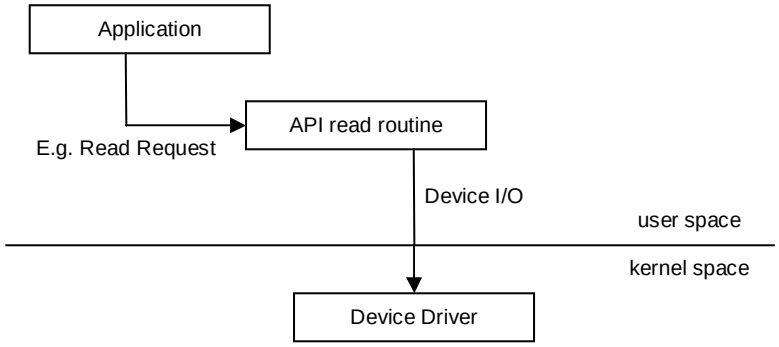


Figure 3.6 Application interaction with a device driver through an API routine

3.6.1 Driver Installation and Packaging

Once a device driver has been developed it must be presented to users and application developers in a way that will enable them to make use of services offered by a device driver with the least amount of effort. If the operating system supports it, an installation wizard should be provided. Installation wizards are often used in Windows to install new applications and drivers. It can be argued that the same facility exists in Linux but it is often not provided by driver writers. API components are usually packaged as libraries which are then added to the system. Sample applications that demonstrate use of the device driver's capabilities should also be provided.

3.7 Summary

Designing device drivers requires a careful examination of facilities offered by the operating system for which the driver will be developed. This section presented the design issues such as memory management and code design that must be considered when designing devices drivers for both the Windows and Linux operating systems. Essential driver components that must be provided when designing IEEE1394 client drivers for both operating systems were also presented.

4 Implementing Device Drivers

The process of creating a device driver requires knowledge of how the associated hardware device is expected to operate. For example, just about every device will allow its clients to read data from and write data to it. In this section, the steps required to create a generic driver for a memory device that supports the most commonly used driver operations are described. Two different versions of this driver will be presented, the first for Windows and the second for Linux so that the similarities and differences of drivers in the two operating systems are highlighted. The chapter concludes by presenting the driver development environments used to create drivers for the two operating systems.

4.1 Implementing Windows Device Drivers

Drivers in Windows consist of various routines. Some are required, others optional. This section presents the routines that every driver must implement. A device driver in Windows is represented by a structure called a *DriverObject*. It is necessary to represent a driver with a structure such as a driver object because the kernel implements various routines that can be performed for every driver. These routines, discussed in the following sections, operate on a driver object.

4.1.1 Driver Initialisation

Every device driver in Windows contains a routine called *DriverEntry*. As its name suggests, this routine is the first driver routine executed when a driver is loaded and is where initialisation of the device driver's device object is performed. This device object should not be confused with the driver object mentioned in section 4.1. Microsofts's DDK [Microsoft DDK, 2002] states that a driver object represents a currently loaded kernel driver whereas a device object represents a physical, logical or virtual device. A single loaded kernel driver (represented by a driver object) can manage multiple devices (represented by device objects). During initialisation, fields in the device object that specify the driver's *unload* routine, *add device* routine and *dispatch* routines are set. The *unload* routine is a routine that is called when the driver is about to be unloaded so that it can perform cleanup operations e.g. freeing up memory allocated off the kernel heap. *add*

device is a routine that is called after the *DriverEntry* routine when a PnP driver is loaded, while the *dispatch* routines are routines that implement driver I/O operations.

4.1.2 The AddDevice Routine

PnP drivers implement a routine called *AddDevice*. In this routine a device object is created, at which time space for storing global data for a device is allocated. Device resource allocation and initialisation is also performed.

4.1.2.1 Creating a device object

A device object corresponding to a functional device object (discussed in chapter 3) for a device is created using an I/O Manager routine called *IoCreateDevice* inside the *add device* routine. The most important requirements for *IoCreateDevice* are a name for the device object and device type. The name allows applications and other kernel drivers to gain a handle to the driver, in order to perform I/O operations. The device type specifies the type of device the driver is used for e.g. a Storage device.

4.1.2.2 Global Driver Data

When a device object is created it is possible to associate with it a block of memory, called *DeviceExtension* in Windows, where custom driver data can be stored. This is an important facility as it eliminates the need to use global data structures in driver code, which can be difficult to manage. For example, in the case where a local variable with the same name as a global variable is declared in a routine mistakenly, the driver writer may find it difficult to track a bug in the driver. It also makes it easier to manage device object specific data, when more than one device object exists in a single driver, as is the case when a bus driver manages child physical device objects for devices present on its bus.

4.1.2.3 Device naming

The name for the device object can be used for accessing a handle to it. The handle is then used for performing I/O. Microsoft recommends not naming functional device objects created in filter and functional drivers. As pointed out by Oney [Oney, 1999], if a device object is named, any client can open the device object and perform I/O for non-disk device drivers. This is because the default access control status Windows gives to

non-disk device objects is an unrestricted one. Another problem is that the name specified does not have to follow any naming protocol, so the name specified may not be a well chosen one. For example two driver writers may come up with the same name for their device objects, which would cause a clash.

Windows supports a second device object naming scheme using device interfaces. Device interfaces are constructed using 128 bit globally unique identifiers [Open Group, 1997]. A GUID can be generated using a utility provided by the Microsoft DDK. Once generated, a GUID can be publicised for drivers written for a specific device. A driver registers its GUID in the add device routine through a call to the I/O manager routine *IoRegisterDeviceInterface*. Once registered, it must enable the interface through a call to the I/O manager routine *IoSetDeviceInterfaceState*. The registration process will add an interface data entry to the Windows registry file, which is accessible by applications.

4.1.2.4 Driver Access from an Application

An application that wants to perform I/O operations with a device driver must obtain a handle to a device driver through the *CreateFile* Win32 API call. It requires a path to a device such as \\device\devicex. Named devices will have their names appear in the name space called \\device, thus the previous path is for a device named *devicex*. *CreateFile* also requires access mode flags such as read, write and file sharing flags for the device.

Accesses to unnamed devices that have registered a device interface are performed differently as shown in figure 4.1.2.4. This requires obtaining a handle to a device information structure using the driver's GUID, and calling the *SetupDiGetClassDevs* Win32 API routine. This is only possible if the driver registered an interface through which applications can access it (called a device interface class), with the I/O manager routine *IoRegisterDeviceInterface* and enabled that device interface with the I/O manager routine *IoSetDeviceInterfaceState* at load time.

Each time a driver calls the I/O manager routine *IoRegisterDeviceInterface*, a new instance of the device interface class is created. Once a device information handle is

obtained by an application, multiple calls to the Win32 API routine *SetupDiEnumDeviceInterfaces* will return device interface data for each instance of the device interface class. Lastly, a device path for each of the driver instances can be retrieved from the interface data obtained from the previous call with another Win32 API routine, *SetupGetDeviceInterfaceDetail*. *CreateFile* can then be called with the device path for the device instance of interest, to obtain a handle for performing I/O.

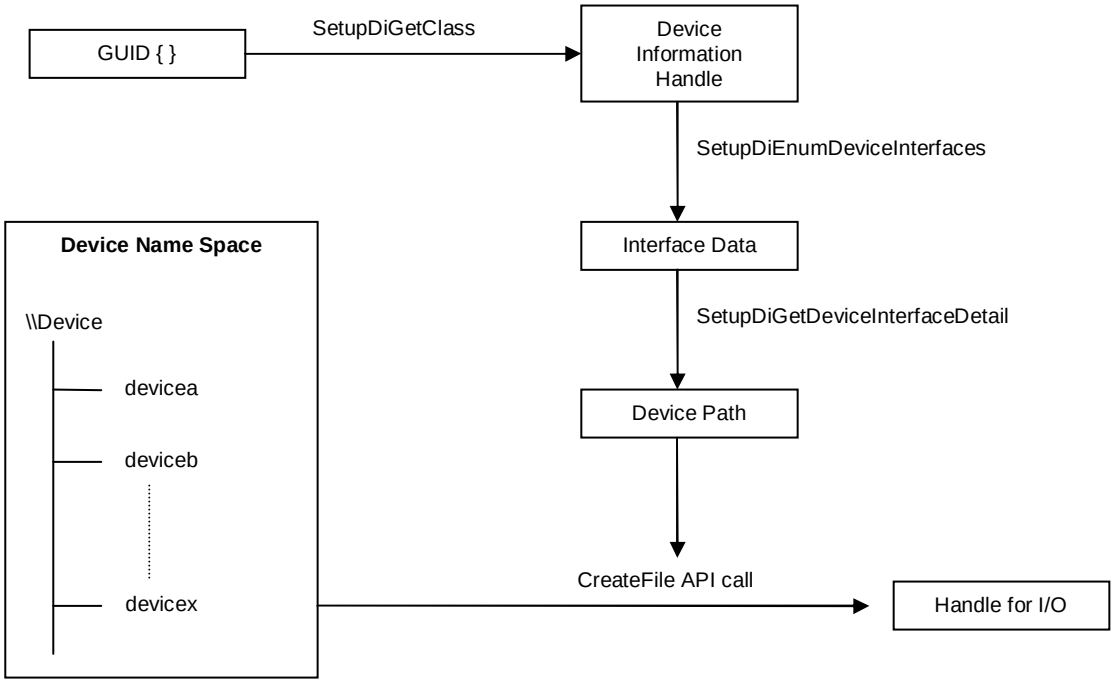


Figure 4.1.2.4 Obtaining a handle an application can use for I/O from a device GUID.

4.1.2.5 Device Object Stacking

Device object stacking is performed in the *add device* routine, so that IRPs sent by drivers in the layer below the driver being loaded can be received by it. This is achieved by a call to the I/O Manager routine *IoAttachDeviceToDeviceStack*. The routine attaches the specified device object to the top of the driver stack as shown on figure 4.1.2.5 and returns a device object that is one below the new one e.g. in the example shown on figure 4.1.2.5 this would be lower device object X. A physical device object is required, which is lower in the stack than the new device object. The lower physical device object can be

any number of layers below the new device object but *IoAttachDeviceToStack* returns the device object one below the current one.

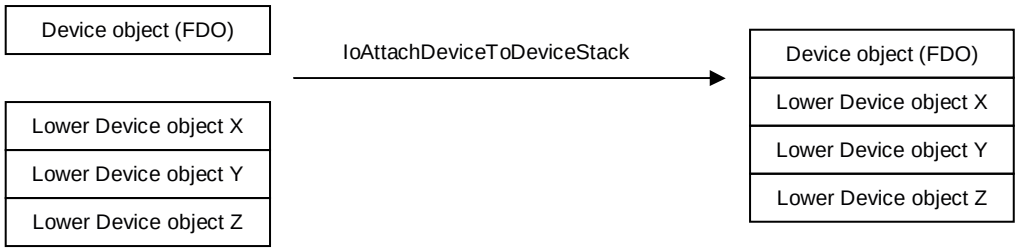


Figure 4.1.2.5 Attaching a device object to the top of a device object stack.

4.1.2.6 User to Kernel and Kernel to User Data Transfer Modes in Windows

The mode used to transfer data from kernel space to user space and vice versa is specified in the flags field of a device object. There are three modes: buffered I/O, direct I/O and I/O that does not use any of the latter methods termed “method neither I/O”. Figure 4.1.2.6 illustrates the three modes. In buffered I/O mode the operating system allocates a kernel buffer that can handle a request. In the case of a write operation, the operating system validates the supplied user space buffer and copies data from the user space buffer to the newly allocated kernel buffer and passes the kernel buffer to the driver. In the case of reads, the operating system validates the user space buffer and copies data from the newly allocated kernel buffer to the user space buffer. The kernel buffer is accessible to drivers as the *AssociatedIrp.SystemBuffer* field of an IRP. Drivers read from or write to this buffer to communicate with applications when buffered I/O is in use.

Direct I/O is the second I/O method that can be used for data exchanges between applications and a driver. An application-supplied buffer is locked into memory by the operating system, so that it will not be swapped out, and a memory descriptor list (MDL) for the locked memory is passed to a driver. An MDL is an opaque structure. Its implementation details are not visible to drivers. The driver then performs DMA to the user space buffer through the MDL. The MDL is accessible to drivers through the *MdlAddress* field of an IRP. The advantage of using direct I/O is that it is faster than

buffered I/O since no copying of data to and from user and kernel space is necessary and I/O is performed directly into a user space buffer.

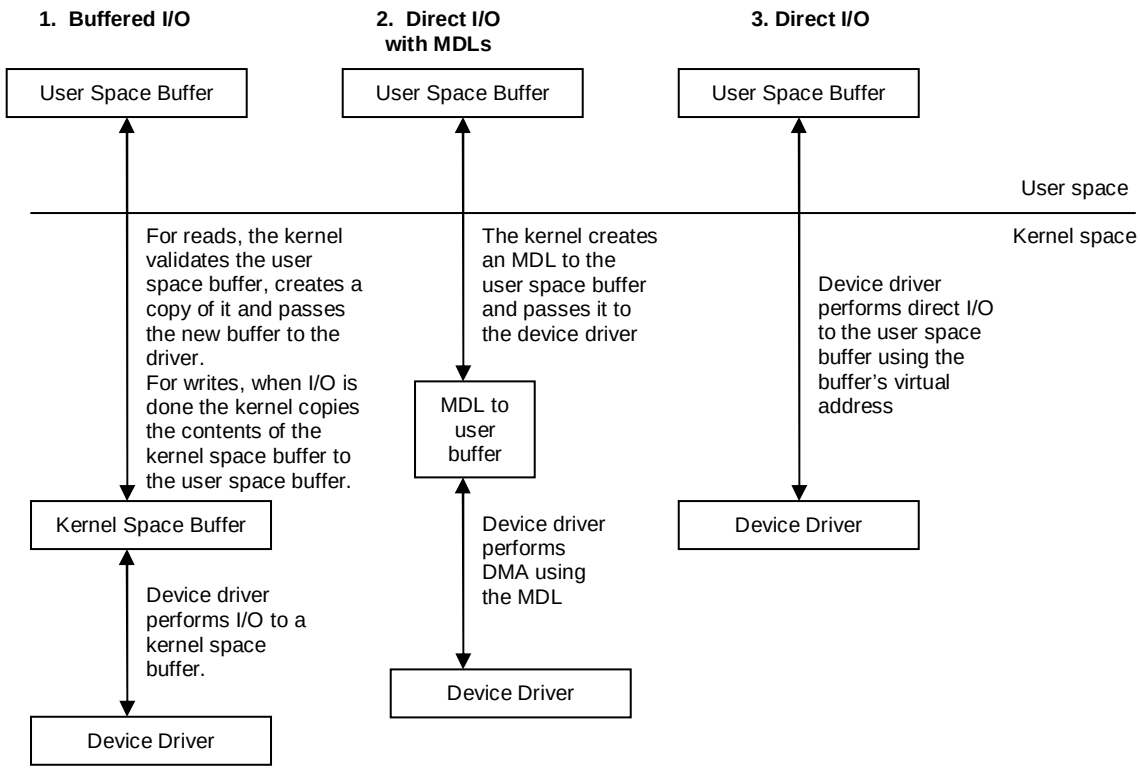


Figure 4.1.2.6 The three ways in which data from kernel to user and user to kernel space is exchanged.

The third method for I/O is neither buffered nor uses MDLs. Instead the operating system passes the virtual address for a user space buffer to the driver. The driver is then responsible for checking the validity of the buffer before it makes use of it. In addition, the user space buffer is only accessible if the current thread context is the same as the application's, otherwise a page fault will occur since the virtual address is valid only while that application's process is active.

4.1.3 Dispatch Routines

Dispatch routines are routines that handle incoming I/O requests packaged as IRPs (I/O request packets). When an IRP arrives (e.g. when an application initiates I/O), an appropriate dispatch routine is selected from the array of routines specified in the

MajorFunction field of a driver object as shown in figure 4.1.3. These dispatch routines are initialised in the driver's entry routine. Every IRP is associated with an I/O stack location structure (discussed in chapter 3) when created. This structure contains a field, which specifies the dispatch routine the IRP is meant for and the relevant

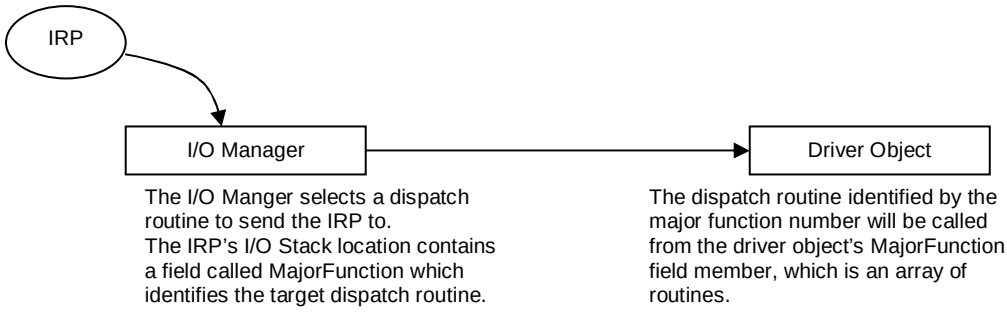


Figure 4.1.3 dispatching IRP's to dispatch routines.

parameters for that dispatch routine. Thus IRPs are routed to an appropriate driver supplied routine so that they can be handled there. Required dispatch routine IDs are shown in table 4.1.3. They are indexes for the array of routines specified by the MajorFunction field of a device object. The dispatch routines have custom driver supplied names that are implemented by the driver. They all accept an IRP and a device object to which the IRP is being sent.

IRP_MJ_PNP	Handles PnP messages
IRP_MJ_CREATE	Handles the opening of device to gain a handle
IRP_MJ_CLEANUP	Handles the closing of the device handle gained above
IRP_MJ_CLOSE	Same as clean up, called after cleanup
IRP_MJ_READ	Handles a read request to a device
IRP_MJ_WRITE	Handles a write request to a device
IRP_MJ_DEVICE_CONTROL	Handles a I/O control request to a device
IRP_MJ_INTERNAL_DEVICE_CONTROL	Handles driver specific I/O control requests
IRP_MJ_SYSTEM_CONTROL	Handles WMI requests
IRP_MJ_POWER	Handles power management messages

Table 4.1.3 Required Windows driver dispatch routines

4.1.4 Windows Driver Installation

Windows uses installation information contained in a text file called an INF file to install drivers. The creator of a driver is responsible for providing an INF file for the driver, without which the driver is useless. A GUI application that is provided with the Windows DDK called *GenInf* allows the generation of an INF file for a driver. It requires a company name and a Windows Device class under which the driver will be installed. Windows has various pre-defined device classes for installed drivers. The Windows device manager applet, accessible through the system control panel applet, shows all installed drivers categorised using these device classes. Examples of existing classes are the 1394 and PCMCIA device classes. A custom device class can be added by adding a *ClassInstall32* section in the INF file.

The hardware ID for a PnP-aware device must also be specified in the INF file since it will be used by the system to identify the device when the device is inserted into the system. A hardware ID is an identification string used by the PnP manager to identify devices that are inserted into the system. Microsoft publishes PnP hardware IDs for the various devices that are usable with the Windows operating system. This hardware ID is stored on the hardware device and read off the device by the system when that device is inserted into the system. Once an INF file for a new device is successfully installed into the system, the driver for that device (which has a specific hardware ID) will be loaded each time the device is inserted into the system and unloaded when the device is removed from the system.

4.1.5 Obtaining Driver Usage Information in Windows

The device manager found in the control panel system applet provides driver information for users. It lists all currently loaded drivers and information on the providers of each driver and their resource usage. It also displays drivers that failed to load and their error codes.

4.2 Implementing Linux Device Drivers

Device drivers in Linux are similar to those in Windows in that they too are made up of various routines that perform I/O and device control operations. There is no driver object visible to a driver, instead drivers are internally managed by the kernel.

4.2.1 Driver Initialisation

Every driver in Linux contains a register driver routine and a deregister driver routine. The register driver routine is the counterpart to the Windows driver entry routine. Driver writers use the *module_init* and *module_exit* kernel defined macros to specify custom routines that will be designated as the register and deregister routines.

4.2.1.1 Driver Registration and Deregistration

The routine designated by the *module_init* macro as the registration routine is the first routine executed when a driver is loaded. The driver is registered here with a kernel character device registration routine called *register_chrdev*. The important requirements for this routine are a name for the driver, a driver major number (discussed later in section 4.2.2) and a set of routines for performing file operations. Other driver-specific initialisation should take place in this routine. The deregistration routine gets executed when the driver is being unloaded. Its function is to perform cleanup operations before a driver is unloaded. A call to the kernel routine *unregister_chrdev* with a device name and major number is necessary when deregistering a driver that was previously registered with a *register_chrdev* call.

4.2.2 Device Naming

In Linux, devices are named using numbers in the range 0 to 255, called device major numbers. This implies that there can be a maximum of 256 usable devices i.e. devices that an application can gain a handle to, but each driver for such a major device can manage as many as 256 additional devices. These driver-managed devices are numbered using numbers in the range 0 to 255 called device minor numbers. It is therefore possible for applications to gain access up to 65535 (256x256) devices. Major numbers are assigned to well known devices e.g. major number 171 is assigned to IEEE1394 devices. The file *Documentation/devices.txt* in the Linux kernel source tree contains all major

number assignments and a contact address for the device number registration authority. Currently, major numbers 240-254 are available for experimental use. A driver can specify a major number of 0 to request automatic assignment of a major number for a device if one is available. The use of major number 0 for this purpose does not cause problems, as it is reserved for the null device and no new driver should register itself as a the null device driver.

4.2.2.1 Driver Access from an Application

Drivers are accessed by applications through file system entries (nodes). By convention, the drivers directory is */dev* on a particular system. Applications that want to perform I/O with a driver use the *open* system call to obtain a handle to a particular driver. The *open* system call requires a device node name such as /dev/tty and access flags. After obtaining a handle, the application can use the handle in calls to other system I/O calls such as read, write and IOCTL.

4.2.3 File operations

In Windows, dispatch routines were set up in the driver entry routine of a driver. In Linux, these dispatch routines are known as file operations and are represented by a structure called *file_operations*. A typical driver would implement the file operations listed in table 4.2.3.

Open	Handles the opening of device to gain a handle
Release	Handles the closing of device handled gained above
Read	Handles a read request to a device
Write	Handles a write request to a device
Llseek	Handles a seek operation on a device
IoCtl	Handles a device control request for a device

Table 4.2.3 Most commonly defined driver file operations in Linux

These file operations are specified during driver registration. A structure called *file* is created by the kernel whenever an application requests a handle to a device and is supplied to the driver whenever one of the file operation routines is called. The file operation routines serve many clients, each represented by the *file* structure. The structure has a field named *f_op*. This field is a pointer to the original set of file operations that

were specified at registration time of a major driver. It is therefore possible to change the original file operations during a call to any of the file operation routines by changing the value of the field named *f_op* to point to a new set of file operations.

4.2.3.1 Global Driver Data

Whenever an application issues the system *open* call on a device file node in */dev*, the application gets back a handle to a device from the operating system. At this time the drivers *open* function is called with a *file* structure created for that open call. This file structure is passed by the kernel to the driver whenever any of the file operations routines are executed. The *private_data* field of the file structure can be any driver-supplied custom data structure. Driver private data is usually set up in the *open* file operations function by allocating memory, and freed in the release file operations function. The private data field can be used to point to data that is global to a driver instead of using global variables.

4.2.4 How Driver Major and Minor Numbers Work

Consider a driver that is registered with major number 4 and has the name *testdriver*. Let there be another driver with the name *driverx*, which is not registered with any major number. Also let there be a device node called */dev/devicex* on the files system with major number 4 and minor number 1 managed by driverx. Figure 4.2.4 illustrates this setup.

The creator of *testdriver* is responsible for routing I/O requests from applications to *driverx*. For example *testdriver* might keep a 256 element array of *file_operation*s structures. It can export a routine to the kernel's symbol table, accessible by other drivers, which sets element x of the array to a *file_operations* structure supplied by a driver with minor number x. *driverx* would make a call to this routine in its driver entry routine, supplying it the minor number it manages and the file operations it implements. This is necessary because in Linux only one driver can register itself for a device major number i.e. device registration uses only a major number. The kernel does not support registration using a major and minor number combination, which would have been a better approach.

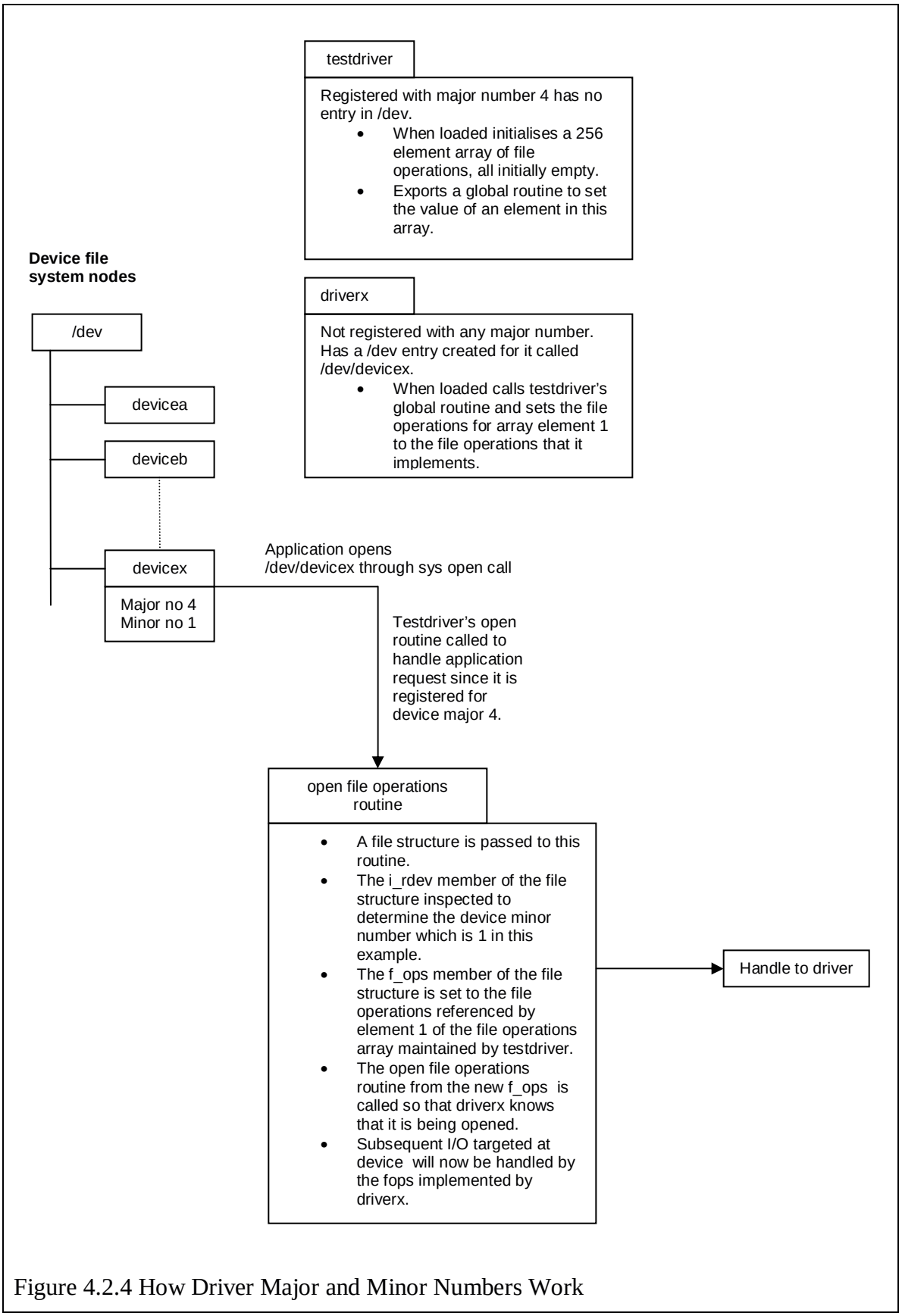


Figure 4.2.4 How Driver Major and Minor Numbers Work

When an application requests a handle to devicex by opening */dev/devicex*, the *open* routine for the driver called *testdriver* is called by the kernel with a *file* structure that represents the open handle, since it is the driver registered with major number 4.

A structure called *inode* is also passed to the *open* routine. This structure contains a field named *i_rdev*, which specifies the major and minor numbers for the device the open operation was targeted at. The kernel macros MINOR and MAJOR can be used to extract these values from the *i_rdev* field. In this example the MAJOR number would be 4 since the open request was routed to *testdriver*. The minor number, which can be any value from 0 to 255, is more interesting to *testdriver*, which should now set up the *f_op* member of the file structure passed to it to point to the file operation routines implemented by *driverx*.

4.2.5 User to Kernel and Kernel to User Data Transfer Modes in Linux

In Linux, three ways exist for exchanging data to and from kernel and user space. These are buffered I/O, direct I/O and mmap. In buffered I/O mode, data is copied from user space to a kernel space buffer before it is used inside a driver. Unlike Windows the Linux kernel does not perform buffering for I/O automatically. Instead kernel user space access routines are made available that allow copying of data to and from user space. In direct I/O mode, data can be written to and from user space buffers directly. This is achieved through the *kiobuf* interface, which involves mapping a user space buffer to a *kiobuf* defined structure through a *kiobuf* kernel call. The operation locks a user space buffer so that it does not get swapped out and is always available for device I/O. The third method, called mmap, involves mapping a chunk of kernel memory to user space, so that applications can use the mapped kernel memory for I/O. [Rubini et al, 2001].

4.2.6 Linux Driver Installation

Drivers are installed in Linux by transferring the driver files into a system specific directory. In the RedHat distribution, modules are located in the directory */lib/modules/kernel_version* where *kernel_version* specifies the version of the kernel currently loaded e.g. 2.4.19. A configuration file called modules.conf located in the system's configuration file directory e.g. */etc*, is used by the kernel while loading

modules. It can be used to override the location for a particular driver. It is also used for defining other module loading options, such as defining parameters to be passed to a driver when loading it.

Module loading and unloading is performed using programs that come with the kernel module utilities package called insmod, modprobe and rmmod. *Insmod* and *modprobe* load the binary image of a driver into the kernel and rmmod removes it. Another program called *lsmod* lists all currently loaded modules. Insmod will attempt to load a module and return an error if the module being loaded depends on other modules. *modprobe* will try to satisfy module dependencies by attempting to load any modules the current module may depend on before loading it. The module dependency information is obtained from a file called modules.dep located in the system's modules directory. The location of the modules directory is dependent to the version of the running kernel, for example for the 2.4.19 kernel it is */lib/modules/2.4.19*.

Before a driver can be accessible to applications a device node for that driver, with the devices major and minor numbers, must be created in the devices directory */dev*. A system program called *mknod* is used for this purpose. When creating a device node, it is necessary to specify whether the node is for a character device or a block device.

4.2.7 Obtaining Driver Usage Information in Linux

It is often necessary to check the status of loaded drivers in a system. In Linux, the proc file system is used to publish kernel information for application usage. The proc file system is just like any other file system. It contains directories and file nodes that applications can access and perform I/O operations with. The difference between files on the proc file system and ordinary files, is that data from I/O operations on a proc file entry gets passed to and from kernel memory instead of disk storage. The proc file system is a communication medium for applications and kernel components. For example, reading data from the file */proc/modules* will return currently loaded modules and their dependencies. The proc file system is very useful for obtaining driver status information and publishing driver specific data for application use.

4.3 The Windows and Linux Driver Implementations Compared

Drivers in both Windows and Linux are dynamically loadable modules that consist of various routines that perform I/O. When loading a module, the kernel will locate a the routine designated by the particular operating system as the driver entry routine and It will start driver code execution from there.

4.3.1 Driver Routines

Drivers in both systems have initialisation and de-initialisation routines. In Linux, both these routines can have custom names. In Windows, the initialisation routine name is fixed (called *DriverEntry*) but the de-initialisation routine can be a custom one. Windows manages a driver object structure for each loaded driver. Multiple instances of a driver are each represented by a device object. In Linux, the Kernel maintains information for each driver registered to manage a device major number. i.e. for each driver that acts as a major device driver.

Both operating systems require drivers to implement standard I/O routines, which are called dispatch routines in Windows and file operations in Linux. In Linux, a different set of file operations can be provided for each device handle returned to an application. In Windows, dispatch routines are defined once in the *DriverEntry* routine inside a driver object. Since there is one driver object for each loaded driver, it is not advisable to modify the dispatch routines assigned to it when an application requests a handle through an open call. Windows drivers have an *add device* routine that gets called by the PnP manager for PnP aware devices.

There is no PnP manager in Linux so that routine does not exist in Linux. Dispatch routines in Windows operate on device objects and IRPs. In Linux, file operations operate on a *file* structure. Custom global driver data is stored in device objects in Windows and in the *file* structure in Linux. A device object is created at load time in Windows where as in Linux a *file* structure is created when an application requests a handle to a driver with a system *open* call. An important implication of this is that in Linux global data per application can be kept in the *file operations* structure. In

Windows, global data can only be present in the FDO that the driver manages. Global data per application in Windows has to be kept in a list structure contained within the FDO's custom data structure.

4.3.2 Device Naming

Drivers in Windows are named using driver-defined strings and are found in the \\device namespace. In Linux, drivers are given textual names but applications are not required to know these. Driver identification is performed through the use of a major-minor number pair. Major and minor numbers are in the range 0-255, since a 16 bit number is used to represent the major-minor pair thus allowing a maximum of 65535 devices to be installed in a system.

Devices in Linux are accessible to applications through file system nodes. In most Linux distributions the directory */dev* contains device file system nodes. Each node is created with a driver's major and minor number. Applications obtain a handle to a driver, for performing I/O, through the *open* system call targeted at a device file system node. In Windows, another driver naming method exists whereby a 128 bit GUID is registered by each driver and applications access the Windows registry to obtain a textual name in the \\device namespace using the GUID. This textual name is used to obtain a handle for performing I/O with a driver through the *CreateFile* Win32 API call.

4.3.3 User-Kernel Space Data Exchanges

Data exchanges to and from user space are performed similarly by both operating systems, enabling buffered data transfer, performed by the I/O Manager in Windows and by the driver in Linux. Direct I/O to a user space buffer is achieved in both operating systems by locking the user space buffer so that it stays in physical memory. This arises from the fact that drivers cannot always access user space buffers directly, since they will not always be executing in the same process context as the application that owns the user space buffers. The application has its own virtual address space, which is only valid in its own process context. Thus when the driver accesses a virtual address from some application outside of that application's process context, it will be accessing an invalid address.

4.3.4 Driver Installation and Management

Driver installation is through a text file called an INF file in Windows. Once installed, the driver for a device will be automatically loaded by the PnP manager when the device is present in the system. In a Linux system, programs are used to load driver binary images into the kernel. Entries need to be inserted manually into system start up files, so that driver loading programs like *modprobe* are executed with a driver image path or an alias to the driver as a parameter.

Driver aliases are defined in the file */etc/modules.conf*, which programs like *modprobe* inspect before loading a driver. An example of an alias entry in *modules.conf* would be "alias sounddriver testdriver", which aliases the name *sounddriver* to the driver binary image testdriver. Executing *modprobe* with *sounddriver* as a parameter would make *modprobe* load *testdriver*. In this way, users can use a standard and simpler name such as *sounddriver* to load a sound driver without having to know what the name of a specific driver for a sound card is.

Driver status information is available in Windows through the device manager applet, or directly reading the data from the system registry. In Linux, driver information is available through entries in the proc file system. The file */proc/module* for example contains a list of loaded modules.

4.4 A Generic Memory Device Driver

This section presents the implementation of a simple driver that performs I/O to blocks of kernel space memory. A discussion of the various components needed to make the driver work on both Windows and Linux is presented, so that the similarities and differences of driver components of each operating system can be highlighted. Applications can perform device I/O operations on the block of kernel memory. The simple memory device is shown in figure 4.4. It consists of a number of blocks of memory. The driver will be able to select the memory bank it wants to access and the offset within a memory bank's block.

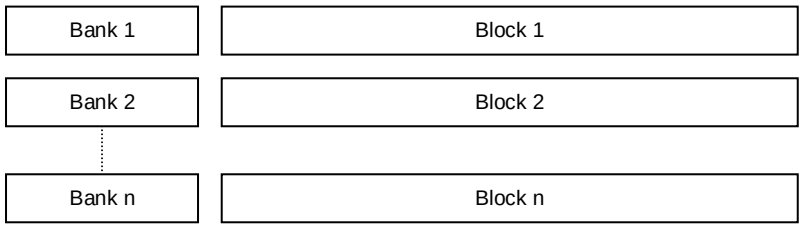


Figure 4.4 A simple memory device

4.4.1 Required Kernel Components.

Both the Windows and Linux drivers will implement the read, write and IOCTL driver routines. Required driver routines for each operating system are shown in figure 4.4.1. Naming of the driver routines is flexible. In figure 4.4.1 the routines for the different operating systems could have been given the same names, instead the conventional platform names have been utilised.

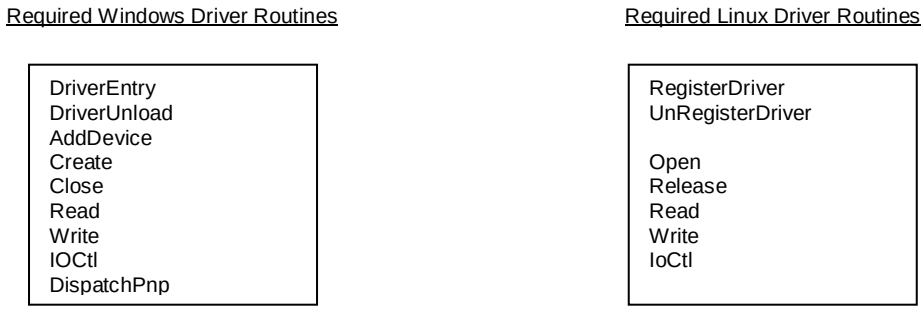


Figure 4.4.1 The Windows and Linux generic driver routines

4.4.2 Driver Load and Unload Routines

In Windows, the step performed in the driver load routine, *DriverEntry*, is the setting up of I/O dispatch routines as shown in figure 4.4.2a. In Linux, the step performed in the driver load routine, *RegisterDriver*, is the registration of a driver major number as shown in figure 4.4.2b. The tagged file operation initialisation, specific only to the GCC compiler, is shown in figure 4.4.2b in the declaration of the structure *fops*, which is not valid ANSI C syntax.

```
        driverObject->DriverUnload = DriverUnload;
        driverObject->DriverExtension->AddDevice = AddDevice;
        driverObject->MajorFunction[IRP_MJ_READ] = Read;
        driverObject->MajorFunction[IRP_MJ_WRITE] = Write;
        driverObject->MajorFunction[IRP_MJ_CREATE] = Create;
        driverObject->MajorFunction[IRP_MJ_CLOSE] = Close;
        driverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
        driverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Ioctl;
```

Figure 4.4.2a Initialisation of a driver Object in the driver entry routine

The compiler will initialise the various fields of the *file_operations* structure (fops) with the supplied driver implemented routine names i.e. *open* is a field name in the structure and *Open* is a routine implemented by the driver, and the compiler will assign a function pointer for *Open* to *open*.

```
struct file_operations fops
{       open: Open,
        release: Release,
        read: Read,
        write: Write,
        ioctl: IoCtl,
}

result = register_chrdev(major_number,"testdriver",&fops);
if(result < 0) PRINT_ERROR("driver didn't load successfully");
```

Figure 4.4.2b Registration of a driver major number in Linux

In the driver unload routine for the Linux driver the registered driver must be unregistered as shown in figure 4.4.2c.

```
unregister_chrdev(major_number,"testdriver");
```

Figure 4.4.2c Driver major number deregistration in Linux

4.4.3 Global Driver Structure

A structure must be defined for storing global driver data that will be operated on by the driver in its routines. For the memory device the same structure will be used for both Windows and Linux versions of the driver. It is defined as shown in figure 4.4.3.

```
#define MAX_BANK_SIZE 4
typedef char byte_t;
#define BLOCK_SIZE 1024;

typedef struct _DEVICE_EXTENSION
{       byte_t * memoryBank[MAX_BANK_SIZE];
        int currentBank;
        int offsets[MAX_BANK_SIZE];

}DEVICE_EXTENSTION, *PDEVICE_EXTENSION;
```

Figure 4.4.3 Structure used to store global data for generic driver

memoryBank is an array of 4 blocks of memory were the size of a block is 1K. *currentBank* indicates the currently selected block of memory and *offsets* records offsets within each of the blocks of memory.

4.4.4 Add Device Routine

The *add device* routine is only specific to Windows. Linux does not have an add device routine. All initialisation must be done in the driver load routine instead. The operations performed in the Windows add device routine are shown in figure 4.4.4. A device object is created with a call to the I/O manager routine *IoCreateDevice*.

An interface that applications will use to gain access to handle for the driver is then created with the I/O manager routine call *IoRegisterDeviceInterface*. One of the arguments to this routine is a GUID manually generated with the system *guidgen* application. The functional device object's *flags* field is set so that it performs buffered I/O. Space for each of the blocks of memory to be used by the memory device is then allocated with one of the kernel memory allocation routines called *ExAllocatePool*. The memory is allocated from the kernel's non-paged pool of memory, thus the device's memory will always be in physical memory.

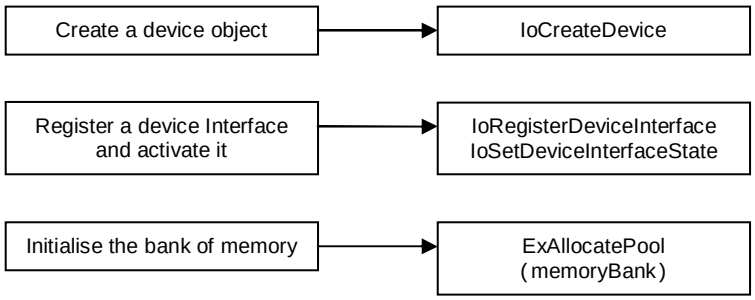


Figure 4.4.4 Operations performed in the Windows driver's add device routine

4.4.5 Open and Close Routines

Most of the initialisation required for the Windows driver has already been done in the add device routine so there is nothing to be done in the open routine. The *Open* routine in Linux performs the operations shown in figure 4.4.5a. Firstly memory for storing global driver data is allocated and stored in the file structure's *private_data* field. Space for the memory device is then allocated in exactly the same way as for Windows. Only the name of the memory allocation routine differs. *ExAllocatePool* in Windows and *kmalloc* in Linux.

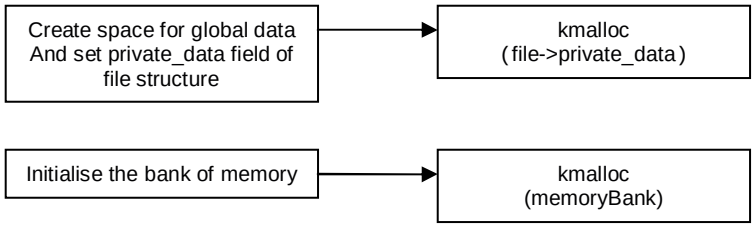


Figure 4.4.5a Operations performed in Linux's generic driver open routine

In Linux's close routine, the space allocated for global driver data as well as space allocated for the memory device are freed as shown in figure 4.4.5b. In Windows, the freeing up of allocated memory is done in response to the PnP *remove device* message, which is discussed later in this section.

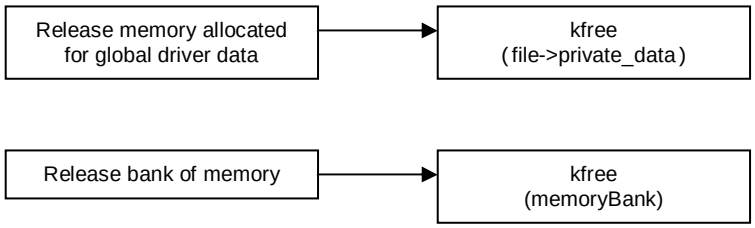


Figure 4.4.5b Operations performed in Linux's generic driver close routine

4.4.6 Read and Write Routines

The *read* and *write* routines will transfer data to and from the currently selected kernel memory bank. In Windows, the read routine is performed as shown in figure 4.4.6a. The length of the data to be read is obtained from an IRP's I/O stack location, in the field named *Parameters.Read.Length*. Data of the requested size is read from the currently selected bank of memory (applications perform memory bank selection through the driver's IOCTL routine discussed later) using the kernel runtime routine called *RtlMoveMemory*, which moves the data from the memory device's space to the buffer allocated for buffered I/O by the I/O manager i.e. the *AssociatedIrp.SystemBuffer* field of the IRP. The IRP is then completed, which informs the I/O manager that this driver has finished processing the IRP and that the I/O manager should return the IRP to the originator of the IRP.

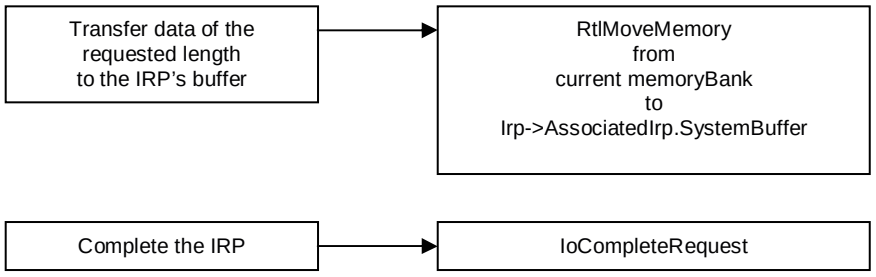


Figure 4.4.6a Performing a read operation in the Windows driver

The write routine performs the opposite of the above memory move operation as shown in figure 4.4.6b.

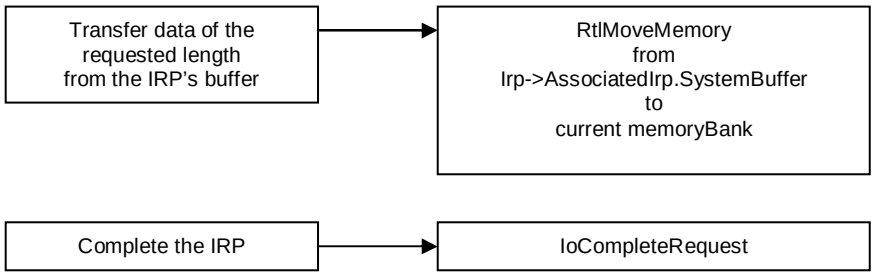


Figure 4.4.6b Performing a write operation in the Windows driver

In Linux, the read routine appears as shown in figure 4.4.6c. A reference to the global driver data is obtained from the *private_data* member of the file structure. From the global data, a reference to the *memoryBank* is obtained. Data is then transferred from this memory bank to user space using the user space access kernel routine called *copy_to_user*.

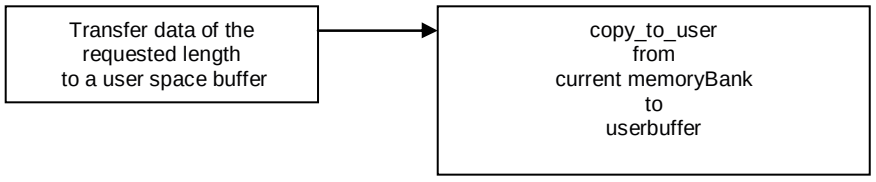


Figure 4.4.6c Performing a read operation in the Linux driver

The write routine performs the same operations as above, except this time data is transferred from user to kernel space as shown in figure 4.4.6d.

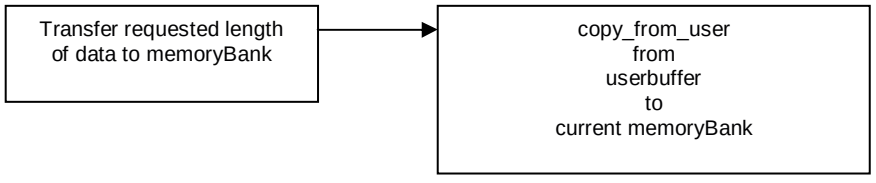


Figure 4.4.6d Performing a write operation in the Linux driver

4.4.7 Device Control Routines

Device control routines are used to set various states of a device. In this example, the IOCTL routine is used to select the current bank number. Driver-specific IOCTL codes

must be defined prior to use. In Windows, IOCTL codes are defined using numbers 0-65535. Codes 0-32767 are reserved for the operating system. Codes 32768-65535 are available for custom use. The code chosen should be the same as the device code specified during the *CreateDevice* call in the drivers *add device* routine.

The IOCTL codes in Windows are defined as shown in figure 4.4.7a. The device ID 61000 is chosen since IDs greater than 32768 are available for custom device use. The first argument to the CTL_CODE macro indicates the device ID, the second argument indicates the function code and is 12 bits long. Codes 0 to 2047 are reserved by Microsoft so a function code greater than 2047 and less than 2^12 is used. The third argument specifies the method used to pass parameters from user space to kernel space, and the fourth argument indicates the access rights that applications have to the device.

```
#define MEMORY_DEVICE 61000

#define IOCTL_SELECT_BANK \
    CTL_CODE(MEMORY_DEVICE, 3000, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_STRING \
    CTL_CODE(MEMORY_DEVICE, 3001, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Figure 4.4.7a IOCTL code definition in Windows

In Linux, IOCTL codes are specified in the file *Documentation/ioctl-numbers.txt*, which can be found in the Linux kernel source tree. Experimental drivers select an unused code, currently 0xFF and above. The IOCTL codes for this driver are defined as shown in figure 4.4.7b. Macro _IOWR indicates that data will be transferred to and from kernel space. Other macros available are _IO which indicates no parameters, _IOW which indicates that data will be passed from user space to kernel space only and lastly _IOR which indicates that data will be passed from kernel space to user space only. The above macros require the size of the parameter that will be exchanged between kernel and user space. Rubini et al [Rubini et al, 2002] suggest that for the driver to be portable, this size should be set to 255 (8bits) although current architecture dependent data ranges from 8-

14 bits. The second argument is similar to the Windows function number. It is eight bits wide, i.e. ranges from 0-255.

```
#define IOCTL_PARAM_SIZE 255
#define MEMORY_DEVICE 0xFF

#define IOCTL_SELECT_BANK \
    _IOWR(MEMORY_DEVICE, 1, IOCTL_PARAM_SIZE)
#define IOCTL_GET_VERSION_STRING \
    _IOWR(MEMORY_DEVICE, 2, IOCTL_PARAM_SIZE)
```

Figure 4.4.7b IOCTL code definition in Windows

Once the IOCTL codes have been selected, the IOCTL routines can be defined. In Windows, the IOCTL routine is defined as shown in figure 4.4.7c.

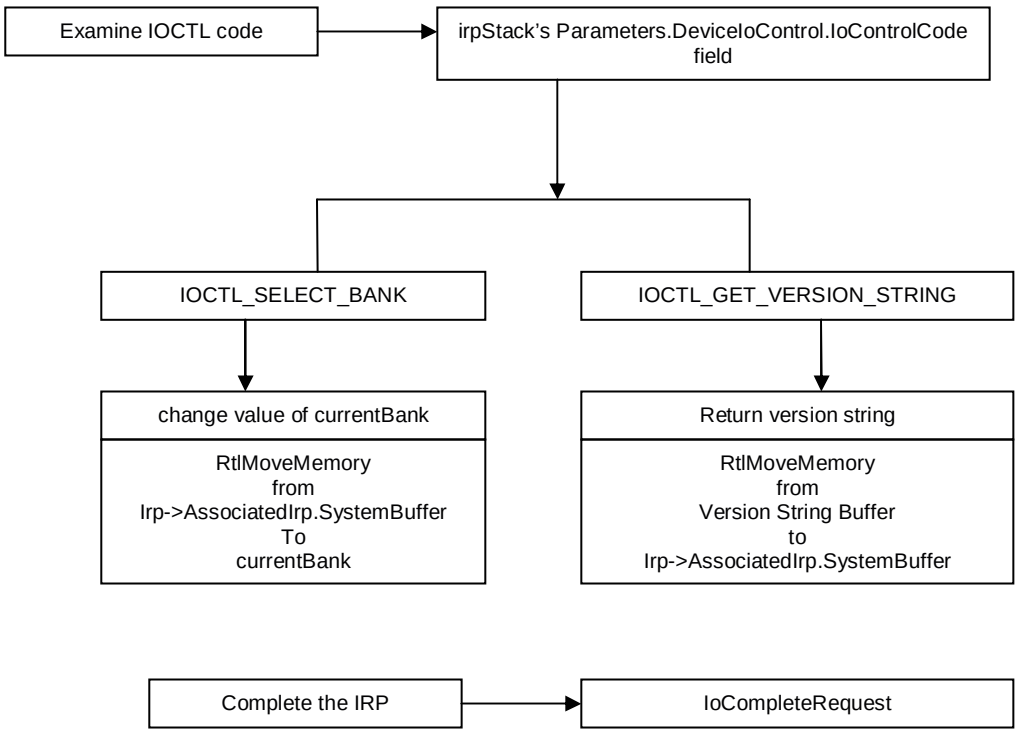


Figure 4.4.7c IOCTL routine definition in Windows

Two IOCTL codes are handled. The first, IOCTL_SELECT_BANK, sets the current bank number. The second, IOCTL_GET_VERSION_STRING, returns a driver version

string. Data returned to callers of the IOCTL routine is handled in the same way as for the read and write requests.

The Linux IOCTL routine definition is shown in figure 4.4.7d. The IOCTL codes handled are the same as for Windows. The only difference is that of syntax specific to each kernel. Data handling is performed in the same way as for reads and writes.

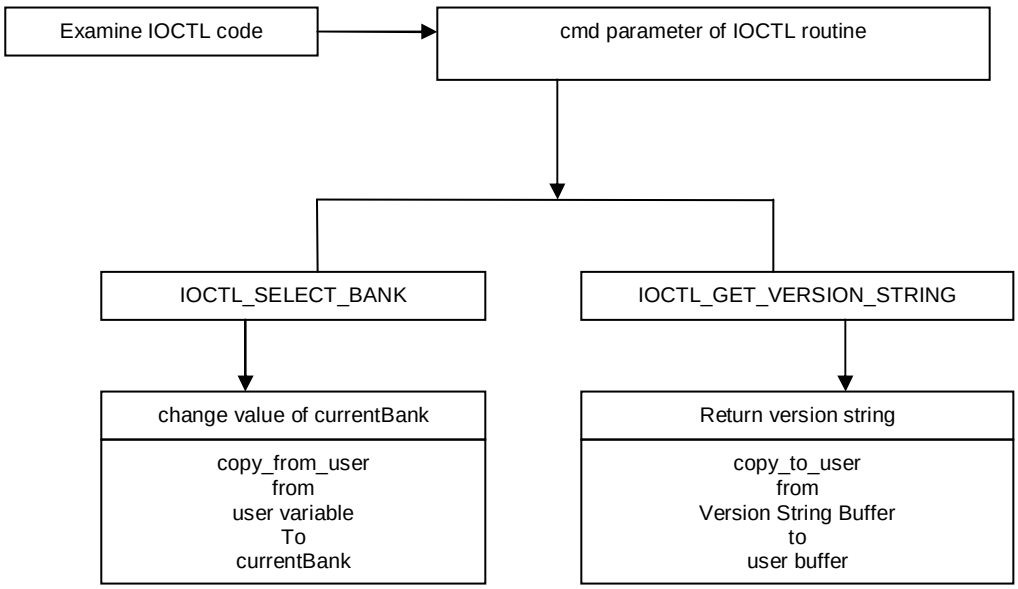


Figure 4.4.7d IOCTL routine definition in Linux

4.4.8 PnP Message Handling Routines

In Windows, PnP messages are dispatched to the driver at appropriate times. E.g. when a device is inserted into the system or removed from the system. These messages are handled by the PnP driver routine. In Linux, the kernel does not send PnP messages to the driver thus a PnP routine does not exist in the Linux driver. The Windows PnP message handler is shown in figure 4.4.8a.

Only one of the PnP messages is handled for the memory device. The *remove device* message is sent when the driver is unloaded from the system. At this time, the driver's interface is disabled with a call to the I/O manager routine *IoSetDeviceInterface* and the

driver's functional device object is deleted as well as the generic driver's blocks of memory.

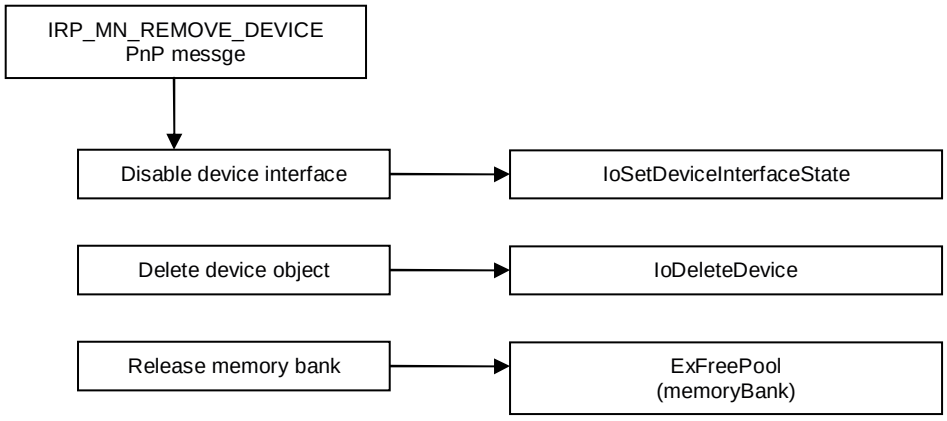


Figure 4.4.8a PnP Message handler routine

4.5 Driver Development Environments

Developing drivers for the two operating systems discussed so far, namely Microsoft's Windows and Linux, requires the use of software development tools specific to each platform. The operating system kernel on both Windows and Linux is constructed using the C Programming Language. It follows that drivers for both operating systems are created using the C programming language. Windows supports driver construction using the object oriented programming language C++, whereas Linux does not.

4.5.1 The Windows Driver Development Environment

Microsoft Windows is a proprietary, commercial operating system i.e. it must be purchased for a fee. A number of commercially available driver development environments aimed at Windows exist. One such example is the NuMega DriverStudio™ Suit [Compuware, 2001] which comes with class libraries and driver construction wizards that aid in the development of drivers, as well as an integrated debugger that allows debugging of driver code.

4.5.1.1 The Windows Device Driver Development Kit

The standard route for driver development on Windows is to obtain a Device Driver Kit (DDK) from Microsoft and use its facilities to build drivers. The latest version of the DDK is available to Microsoft Software Development Network (MSDN) subscribers. The DDK contains programs required to build drivers. The DDK installation program will install batch files that set up a shell window to enable building of drivers for each of Microsoft's operating systems. The DDK release used in this investigation of the Windows driver architecture, Windows DDK 3590, has build environments for Windows ME, 2000, XP and .NET drivers. There are two build environments for each platform. The first is called a checked build environment i.e. debugging symbols are added to the driver code. The second is called a free build environment i.e. drivers built in this environment are not built with debugging symbols. The latter environment is where production drivers are built.

4.5.1.2 Windows Driver Makefiles

Once a DDK shell window is active the simple command *build* will build a driver. A *Makefile* is used to define driver source files from which a driver is to be built. The entries for the *Makefile* are specified in a file called *sources* which resides in the directory where the *build* command is issued. Figure 4.5.1.2 shows the format of a *Makefile* used to build a simple driver.

```
TARGETNAME=mydriver
TARGETPATH=obj
TARGETTYPE=DRIVER
DRIVERTYPE=WDM

INCLUDES=$(BASEDIR)\inc;

SOURCES= mydriver.c
```

Figure 4.5.1.2 a *Makefile* used for building a WDM driver with the Windows DDK

The environment variable TARGETNAME specifies the driver output filename. For this example the final driver will be called mydriver.sys as drivers in Windows are assigned a .sys extension. TARGETPATH specifies where the object code for the driver will be produced. There is a file called *_objects.mac* in the directory called *obj* where additional

paths for object file dump directories are defined. On Windows 2000, the default object file dump directories are *objchk_w2k* for checked builds and *objfre_w2k* for free builds. INCLUDES specifies the path for include files necessary to build drivers and finally SOURCES specifies the driver source file from which the driver will be built.

4.5.1.3 Windows DDK Documentation and Tools

The Windows DDK contains well organised API documentation as well as example driver source files from which newcomers to driver writing can learn to create drivers. The DDK also contains utility programs that aid in the driver development process. One of the utility programs is the device tree application that lists all currently loaded drivers listed in the \\device namespace in a hierarchical manner, showing any existing driver stacks as well as driver implemented routines and driver object memory addresses. A screen shot of the device tree application showing the *raw1394* IEEE1394 client driver constructed during the investigation of the Windows IEEE1394 stack, is shown in figure 4.5.1.3.

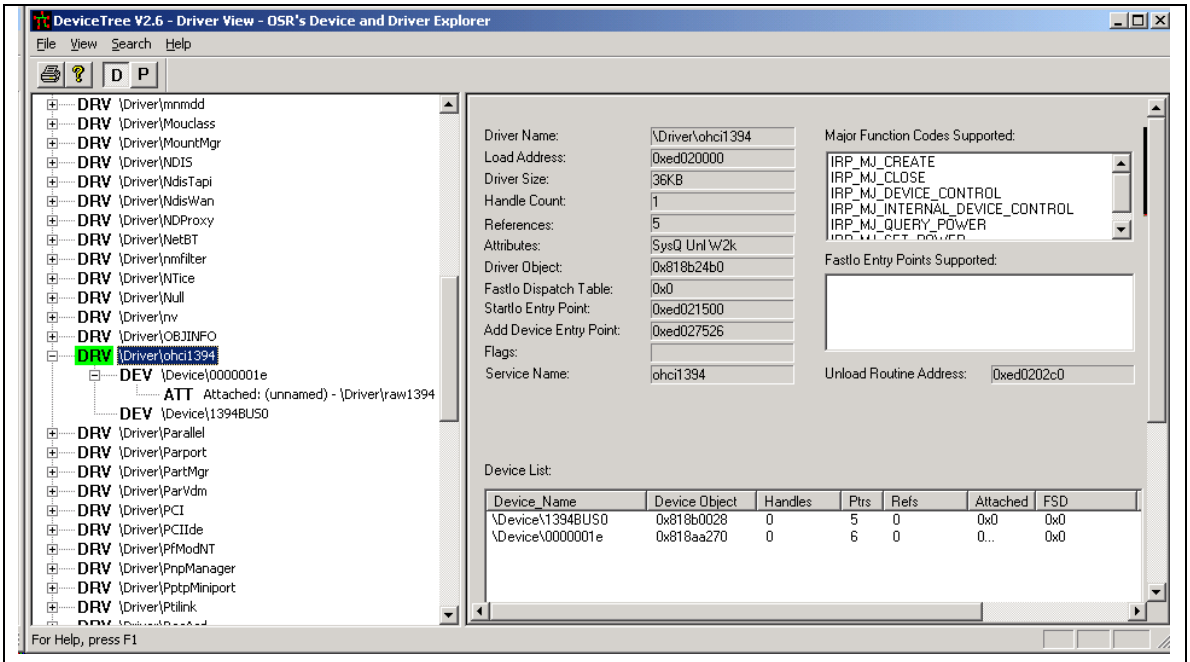


Figure 4.5.1.3 The device tree application showing the IEEE1394 stack

From the figure it can be seen that the *raw1394* driver is attached to a 1394bus driver device object which in turn is stacked on top of an ohci1394 device object. Other

programs that are provided with the Windows DDK are an INF file generation application called *geninf* used to generate INF files for driver installation and a PnP driver test application used to test if drivers are PnP compliant.

4.5.1.4 Windows IEEE1394 driver documentation, Example Drivers and Applications

The Windows DDK provides complete documentation for its IEEE1394 API. Sample drivers and applications are also provided. However the supplied IEEE1394 diagnostic application and drivers do not function as expected. This issue has been raised a number of times in the Windows driver development mailing list, *microsoft.public. development.device.drivers,* but has not yet been addressed by Microsoft and was verified on DDK build 3590 used during the investigation of the Windows IEEE1394 stack. There is also no literature currently available that fully describes the implementation of the IEEE1394 driver stack on Windows.

4.5.2 The Linux Driver Development Environment

The driver development environment on Linux is different from that on Windows. There is no counterpart to the Windows DDK on Linux i.e. there is no such thing as a Linux Device Driver Kit supplied by the kernel's creators. Instead the kernel's creators make all the kernel source code available to everyone. The kernel header files are all that are required for creating drivers. Drivers are built using the GNU C compiler, GCC, which is also used to build applications. Similarly to Windows, a *Makefile* is used to specify how a driver is to be built.

4.5.2.1 Linux Driver Makefiles

Once a *Makefile* is defined, the simple command *make* is used to build the driver. Figure 4.5.2.1 shows an example *Makefile* for building a driver, called *mydriver,* in Linux with a source file name *mydriver.c.* The first entry, KERNELDIR, defines an environment variable that specifies where the kernel header files are located. The next line includes the current kernel configuration. Before a kernel is built, externally definable kernel variables are specified in a file called .config which is stored in the kernel source tree's root directory. These are included so that the kernel's header files can make use of them. CFLAGS is used to set additional flags to the compiler (GCC). "-O" sets code

optimisation on, "-Wall" prints out all code warnings. The "all" section is the default section examined when the "make" command is executed. It points to a target called *mydriver*, which depends on the object file called *mydriver.o*, which is built by using GCC. The environment variable LD specifies the GNU linker to be used to build the final driver module. The option "-r" specifies that output should be relocatable i.e. memory locations within it will be offsets to some base address not known at compile time. "$^" is an alias for *mydriver.o* and "$@" is an alias for *mydriver* i.e. it requests the linker to produce relocatable code from the object file mydriver.o and produce an output called mydriver.

```
KERNELDIR=/usr/src/linux
include $(KERNELDIR)/.config
CFLAGS=-D__KERNEL__ -DMODULE –I $(KERNELDIR)/include –O –Wall

all: mydriver

mydriver: mydriver.o
        $(LD) –r $^ -o $@
```

Figure 4.5.2.1 *Makefile* used to build a driver in Linux

Kernel module management programs such as *insmod* and *lsmod* can then be used to load the driver into the kernel and to observe currently loaded modules, respectively.

4.5.2.2 Linux Driver Development Documentation, Example Drivers and Applications

There is some documentation on the various parts of the Linux kernel in the directory called "Documentation" found under the kernel source tree, but is not as complete and descriptive as the Windows DDK documentation. The Linux driver book written by Rubini *et al* [Rubini *et al*, 2001] is a better source of information for device driver writers. There are no example drivers that come with the Linux kernel, but code for existing production drivers is available, and can be used as a base for starting a new device driver. The problem with this is that it may be more difficult or may take longer for new driver developers to learn to develop new drivers.

4.5.2.3 Linux IEEE1394 Driver Documentation, Example Drivers and Applications

There is no documentation nor example programs available currently that fully describe and demonstrate the implementation of the Linux IEEE1394 driver stack. Libraw1394 is a library that allows raw IEEE1394 I/O on an IEEE1394 bus. It has some documentation and companion test programs. A GUI application called *gscanbus* also exists that represents IEEE1394 nodes on the bus, and allows simple diagnostic operations to be performed on them. Other than what has just been described, IEEE1394 driver writers have to read through the IEEE1394 driver sources, understand the layout of the code there, and create their drivers modelled on existing IEEE1394 drivers. An important resource for Linux IEEE1394 device driver developers is the Linux IEEE1394 development mailing list, which is used by the current developers to discuss implementation changes and bug fixes to their IEEE1394 stack. New driver writers can direct their queries about the stack to this list. Information for joining the mailing list is available at the Linux IEEE1394 project website [Linux 1394, 2002].

4.5.3 Debugging drivers

The creation of just about every piece of software requires it to be debugged sometime during the time of its development as there are always obscure bugs that cannot be discovered by examining source code manually. This is especially true for device drivers. At worst, bugs in applications might cause the application's process to become unstable. Serious bugs in drivers will cause the entire system to become unstable.

Debugging applications is a straightforward process. A break statement is set at a place of interest in source code using a debugger's debugging facilities. This is usually debugger specific. In Windows, using Microsoft's Visual Studio debugger, setting break points is as simple as clicking a line in the source code editor. The same applies to DDD (a GUI debugger found on Linux that uses the popular command line debugger GDB). When a program is executed in debug mode and a break point is reached, the execution of that program is paused and the program can be single stepped i.e. instructions from it executed one at a time and their effects observed. A debugger will usually allow the values of variables and variable memory addresses in a running program to be observed.

The debugging facilities discussed thus far are also available for debugging device drivers, to a certain extent, on each operating system.

4.5.3.1 Debugging Drivers on Windows

Debugging drivers on Windows can be performed using a number of different methods. The simplest of these is to use the *DbgPrint* debugging routine which allows printing of messages to the Windows debugger buffer. If a Windows debugger such as WinDbg is running then the messages can be observed from there, otherwise a special application that can retrieve messages from the debugger's buffer has to be used. One such application is *DebugView*, a free application provided by the SysInternals Corporation [Russinovich, 2001]. The *DbgBreakPoint* routine sets a break point in a program. When executed, the system pauses and driver code execution is passed to the system debugger. The *Assert* macro transfers driver execution to the system debugger based on the value of a test condition.

The Microsoft kernel debugger, *WinDbg*, requires two PCs for operation. The first PC is where the driver code is developed and tested. The second PC is connected to the driver development PC via a serial port. A developer can interact with the debugger running on the first PC through a serial console from the second PC. The NuMega DriverStudio ™ [Compuware, 2001] provides a debugger that allows drivers to be debugged from a single PC, which can be the driver development machine, and acts like an application debugger. It provides a console Window from which command line instructions can be issued to control it.

4.5.3.2 Debugging Drivers on Linux

In the same way as Windows, debugging of drivers in Linux can be performed by using debug routines provided by the kernel such as *printk*, which is the equivalent of the Windows *DbgPrint* routine. It behaves in the same way as the C standard I/O routine *printf* except that it takes an additional argument that specifies where the message will be printed to. A kernel debugger is also available as a patch that can be applied to the kernel sources. The patch for the built-in Linux kernel debugger (kdb) can be obtained from the

KDB project page [KDB, 2002]. It allows the same operations as a standard debugger i.e. setting break points, single stepping driver code and, examining driver memory.

4.6 Other Development Models: Creating device drivers in C++ using object orientation

In the past drivers, for example those written for MSDOS, used to be created with assembly language programming using a host architecture's CPU instructions. This approach to creating drivers meant that, once created, a driver was confined to a particular machine architecture e.g. x86, Sparc or PowerPC. Modern operating systems improved this by introducing Hardware Abstraction Layer (HAL) routines for the different machine architectures, which implement the functionality for accessing hardware. Drivers could then be created using a programming language such as C and be compiled for multiple machine architectures.

When a device driver is constructed in the C programming language, it consists of a set of routines and a standardized structure that gets passed to each of these routines. This structure contains instance data, on which the driver's routines operate. One example of instance data that a driver stores is a FIFO queue that the driver uses for handling requests from a single application. Device driver development using the C++ language allows programmers to use object orientation. A class can represent a driver. The methods of this class would perform the same functionality as the routines implemented by a driver coded in C. The attributes of this class would contain the data stored in the standardized structure for storing driver instance data of a driver coded in C. Asche in [Asche, 1995] gives the following as advantages of driver development in C++:

- In a driver written in C, memory allocation/de-allocation operations can be scattered in many areas of the driver's source code. This could make identifying memory leaks difficult. C++ offers the constructor/destructor facility for its objects, where memory allocation/de-allocation operations can be made. If memory allocation/de-allocation operations are performed within constructors and destructors consistently throughout driver code, when a memory leak is detected the driver's creator would know where to look first.

- C++ classes designed to perform a generic task can be reused with other drivers with no modification, even on other operating systems.
- Object orientation facilities such as inheritance, overloading and polymorphism can be used at the driver level. For example there could be a parent input device driver from which other more specific drivers like keyboard and mouse drivers can inherit functionality. These drivers can overload (re-implement) methods defined by their parent to implement their own functionality.

The disadvantage of creating drivers in C++, is that C++ method calls are more expensive than their C counterparts. The overall driver performance will be slower than an equivalent C driver. C++ code also uses more memory than C code. Asche [Asche, 1995] provides implementation of drivers in C++ on Microsoft Windows. The Linux kernel does not support drivers written in C++ [Gooch, 2002].

4.7 Summary

This section examined the components required to implement drivers on the Windows and Linux operating systems. The material is presented in such a way that the differences and similarities of driver implementation on the two operating systems were highlighted. Required driver routines, device naming, driver installation and the application to driver link were discussed. An implementation of a generic memory device driver, with a side by side definition for each operating system's required driver routines was presented. The driver development environment used to create drivers for each operating system was also presented.

5 IEEE1394 Driver Implementations

The Windows and Linux operating systems both have host controller and bus drivers for the IEEE1394 bus. In this chapter, the API for these drivers is presented, as well as new IEEE1394 client drivers that were implemented during this project to improve on an existing IEEE1394 client driver in the case of Linux, and implementation of a new IEEE1394 client driver in the case of Windows. Test applications and the user space APIs developed for the above client drivers are also presented.

5.1 An Overview of the Current Windows and Linux IEEE1394 Drivers

Linux has the various client drivers shown in Figure 5.1a. *raw1394* allows generic IEEE1394 operations and exposes a user space library called *libraw1394* that applications can utilise. *raw1394-2* was implemented during this project. It performs the same operations that raw1394 can perform and offers enhancements. *video1394* is a driver that allows video transmission and receipt on the IEEE1394 bus. It is now succeeded by *dv1394*. *amdtp* is a driver that allows transmission of audio over IEEE1394 in 61883-6 format. The *spb2* driver implements a SCSI like interface for IEEE1394 storage devices. Lastly, the *eth1394* driver implements Ethernet over 1394, but is not IP over IEEE1394 compliant.

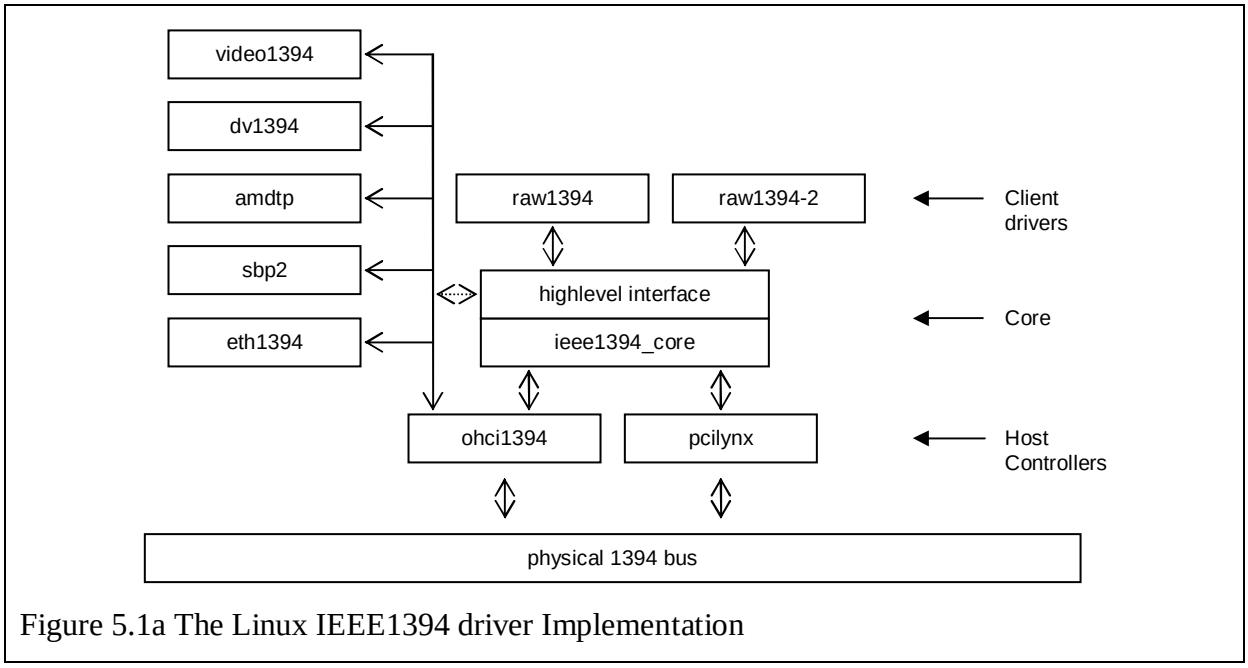


Figure 5.1a The Linux IEEE1394 driver Implementation

Windows has client drivers *61883.sys* and *nic1394.sys* as shown in Figure 5.1b. The Windows DDK supplies two example IEEE1394 drivers that expose the bus driver's API to applications and a test application called *win1394*, but these drivers do not work properly. When compiled, installed and when the test *win1394* application is run, an error message that indicates that a valid IEEE1394 device object was not found on the system is displayed. The drivers only serve as examples that can be used as a basis for creating other drivers. Another client driver called *raw1394* is shown in figure 4.2b. *raw1394* was implemented during this project. *61883.sys* is a client driver that implements the 61883-1 and 61883-6 protocols, which specify audio transmission over IEEE1394. *Nic1394.sys* is a client driver that implements the IP over IEEE1394 protocol. *1394bus.sys* is the IEEE1394 bus driver and exposes a kernel space API that client drivers can use.

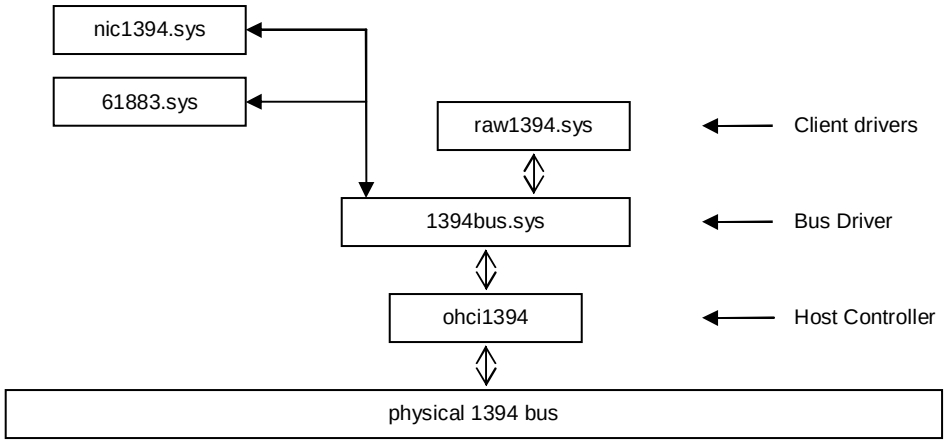


Figure 5.1b The Windows IEEE1394 driver Implementation.

5.1.1 raw1394 for Windows

raw1394 is a driver that performs generic IEEE1394 operations and can be used to implement the functionality offered by kernel client drivers such as *61883.sys* in user space. The advantage of this is that the complexity involved in development of drivers for kernel space is greatly reduced. Debugging code in kernel space is more difficult, since errors in driver code could affect the kernel and may cause the entire system to be unstable. In user space on the other hand, only the process belonging to the application being executed will be affected. As mentioned in section 5.1, Windows does not ship

with a driver with the functionality of *raw1394*. *raw1394* performs asynchronous read, write, lock and listen operations in addition to isochronous transmission and receipt. The following sections present the various components of this driver.

5.1.1.1 General Driver Components

Throughout the driver implementation, IEEE1394-defined data types are used. This makes the driver code easier to understand. The standard IEEE1394 data types are defined as shown in table 5.1.1.1.

nibble_t	4 bit
byte_t	8 bit
doublet_t	16 bit
quadlet_t	32 bit
octlet_t	64 bit

Table 5.1.1.1 Standard IEEE1394 defined data types.

The definitions used in the *raw1394* driver, for the data types in table 5.1.1.1, use Windows specific data types. *octlet_t* is 64 bit (DWORD64 in Windows), *quadlet_t* is 32 bit(DWORD32 in Windows), *doublet_t* is 16 bit (USHORT in Windows), *byte_t* is 8 bits (UCHAR in Windows), and *nibble_t* is 4 bits (UCHAR in Windows) defined here as 8 bits since a 4bit primitive does not exist on Windows. The custom driver structure used to store global driver data is another essential component of the *raw1394* driver. It is shown in figure 5.1.1.1.

```
typedef struct _DEVICE_EXTENSION {
   PDEVICE_OBJECT       deviceObject;
   PDEVICE_OBJECT       lowerDeviceObject;
   PDEVICE_OBJECT       portDeviceObject;;

   PADDRESS_RANGE_DATA pAddressRangeDataListHead;
   KSPIN_LOCK   pAddressRangeDataSpinlock;

   ISOC_DATA isocdata[64];
   PKEVENT pBusResetDataEvent;

   BOOLEAN shutdown;

   ULONG                generation;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Figure 5.1.1.1 Structure used for storing global driver data by raw1394

The *deviceObject* is a functional device object for raw1394 that gets created in the *add device* driver routine, the *lowerDeviceObject* is a physical device object representing the *1394bus.sys* driver and the *portDeviceObject* is a physical device object representing the host controller driver. The ADDRESS_RANGE data structure is used to store registered address information for asynchronous listen operations. The ISOC_DATA structure is used to store data specific to each of the 64 possible isochronous channels. g*eneration* is used to store the current bus generation number. This number is updated every time there is a bus reset. The KSPIN_LOCK structure is used to perform multiprocessor safe accesses to the *address range* data structure, which is composed of a linked list of registered address data. The KEVENT structure is a kernel event object used for making the bus reset listener thread block, to make it wait for a bus reset event to occur. The boolean variable *shutdown* indicates the state of the device.

5.1.1.2 Add Device Routine

In the *add device* routine, *raw1394* creates a functional device object, registers a device interface and attaches itself to the top of the IEEE1394 stack. It keeps a reference to the physical device object that represents the 1394bus driver. It also obtains a pointer to the host controller physical device object through the operation shown in figure 5.1.1.2.

```
PDEVICE_EXTENSION pdx;
…
PNODE_DEVICE_EXTENSION pnodeExt = pdx->lowerDeviceObject->DeviceExtension;
pdx->portDeviceObject = pnodeExt->PortDeviceObject;
```

Figure 5.1.1.2 Obtaining a reference to the host controller device object in Windows

The above operation is a non-documented way for getting a reference to the host controller physical device object. It was obtained from the sample IEEE1394 drivers in the Microsoft DDK. Getting a reference to the host controller PDO allows performing read and write operations without interception from the 1394bus driver, which overwrites packet destination address fields. For example if a packet is addressed to node id 299:0, the 1394bus driver will alter the node id to 1023:0 (local bus). This behaviour is not desirable in multi-bus environments. The add device routine also sets the power state of the device to *PowerDeviceD0*, which specifies that the device is fully powered up.

5.1.1.3 Interaction with the 1394bus driver

IEEE1394 operations are performed by sending an IRB encapsulated within an IRP to the Windows IEEE1394 bus driver using the I/O manager routine *IoCallDriver*. The IRP is allocated using the I/O Manager routine *IoAllocateIrp.* An IRB is encapsulated within an IRP by setting the *Parameters.Others.Argument1* field of the IRP's I/O Stack location to point to the IRB structure. The IRP-IRB relationship is shown in figure 5.1.1.3a. An IRP's I/O stack location is a block of memory which contains parameters that will be used by clients while processing an IRP.

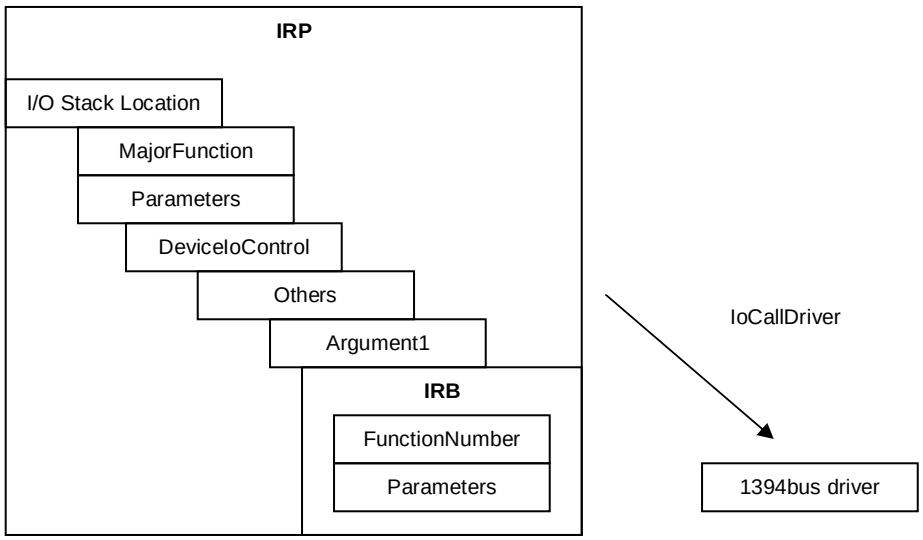


Figure 5.1.1.3a Components of an IRP sent to the 1394bus driver by raw1394

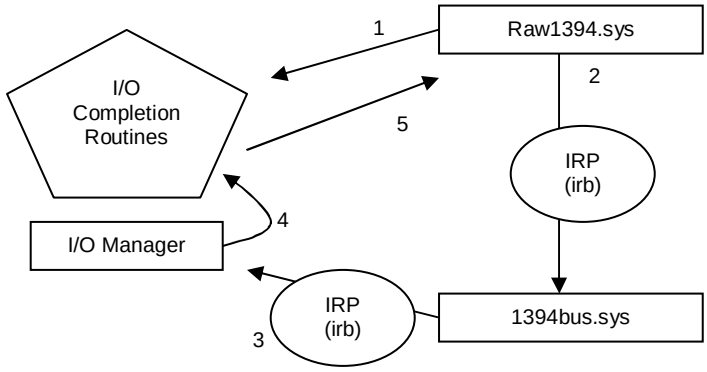
The *MajorFunction* and *Parameters.DeviceIoControl.IoControlCode* fields of the IRP's I/O stack location together specify the I/O control code that the *1394bus* driver handles. For all requests sent to the *1394bus* driver, the IRP stack's *MajorFunction* field is set to IRP_MJ_DEVICE_CONTROL, which tells the *1394bus* driver that this is a device control request and the *Parameters.DeviceIoControl.IoControlCode* field is set to IOCTL_1394_CLASS, which is the IOCTL code that the *1394bus* driver is implemented to handle.

An IRB has a field called *FunctionNumber* which selects the appropriate operation to be performed by the 1394bus driver. The operation may require input parameters or produce

results which are saved in output parameters. For example, the asynchronous read operation requires a destination node address as one of its input parameters.

When sending an IRP down to the 1394bus driver using the I/O manager routine, *IoCallDriver*, that routine returns immediately. This does not mean the operation to be performed by the 1394bus driver with the supplied IRP was completed. If the *raw1394* driver requires notification of completion of the operation, e.g. when performing an asynchronous read, it should set an I/O completion call-back routine using the I/O manager routine *IoSetCompletionRoutine*.

It can then initialise a kernel event object and wait on the object after a call to *IoCallDriver*. Inside its I/O completion call-back routine it can signal the kernel event object. The thread waiting on this event object will then know that the operation is complete. The driver will then extract output values from the IRB contained within the IRP. The above process is illustrated in Figure 5.1.1.3b. All operations performed by the *raw1394* driver using the *1394bus* driver are carried out in this way.



1- *raw1394* registers an I/O Completion routine.
2- It then sends an IRP to *1394bus.sys* and waits on an event object.
3- *1394bus.sys* completes the requested operation and notifies the I/O manager
4- The I/O manager calls the registered I/O completion routine which signals the event object *raw1394* is waiting on.

Figure 5.1.1.3b Raw1394 and 1394bus driver interaction

5.1.1.4 PnP Message Handling

IEEE1394 devices are fully PnP aware, so it is important that PnP messages are handled properly. Table 5.1.1.4 shows the PnP messages sent by the PnP manager and how they

are handled by the raw1394 driver. If any of the messages are not handled properly the loading of the driver into the kernel may not be successful.

PnP Dispatch Routine Message	How the message is handled by Raw1394
IRP_MN_START_DEVICE	Message sent to a driver after it is enumerated or the when the device is restarted. This IRP associated with this PnP message is passed down to the 1394bus driver to notify it of the arrival of this device. The device shutdown flag is set to FALSE. A bus reset handler is installed and the interface (created in the add device routine) for the device is activated.
IRP_MN_STOP_DEVICE	Message sent to a device so that it can reconfigure its hardware resources. The device is stopping, the shutdown flag is set to TRUE, the bus reset handler is removed, the interface to the device is disabled, and the 1394bus driver is notified of the removal of this device by passing it down this IRP.
IRP_MN_QUERY_STOP_DEVICE	Message sent to request driver if the device can be stopped for resource rebalancing. Passed down to the 1394bus driver.
IRP_MN_CANCEL_STOP_DEVICE	Message sent to inform the driver of the cancellation of an IRP_MN_STOP_DEVICE PnP message. The device is restarted as in IRP_MN_START_DEVICE
IRP_MN_QUERY_REMOVE_DEVICE	Message sent to the driver to find out if the device can be removed safely from the system. Passed down to the 1394bus driver.
IRP_MN_SURPRISE_REMOVAL	Message sent when the device is unexpectedly removed. Same operation as IRP_MN_STOP_DEVICE is performed.
IRP_MN_REMOVE_DEVICE	Message sent to inform the driver of the removal of the device from the system. The operations specified in IRP_MN_STOP_DEVICE are carried out if they have not already been performed. The device object is detached from the Windows IEEE1394 stack and deleted.
IRP_MN_CANCEL_REMOVE_DEVICE	Message sent to inform the driver of the cancellation of the IRP_MN_REMOVE_DEVICE message. The device is restarted as in IRM_MN_START_DEVICE.
IRP_MN_QUERY_CAPABILITIES	Message sent after a device is enumerated. Passed down to the 1394bus PDO which will set up default capabilities for a 1394 device.

Table 5.1.1.4 Raw1394 driver PnP message handling

5.1.1.5 Performing I/O Operations

Device I/O operations are usually handled through the *read* and *write* driver implemented routines. IEEE1394 devices differ from normal I/O devices, such as storage devices, in that each request requires a node destination or source address to be supplied in addition to a buffer and its length for an I/O operation. For this reason all I/O operations performed by the raw1394 driver are implemented in the IOCTL driver routine, which can accept input parameters from clients and provide output parameters to them. The I/O control codes shown in table 5.1.1.5a were defined for the Windows raw1394 driver.

IOCTL_ASYNC_READ	IOCTL_UNSET_EVENT_HANDLE
IOCTL_ASYNC_WRITE	IOCTL_GET_ASYNC_DATA
IOCTL_ISO_START_LISTEN	IOCTL_GET_GENERATION
IOCTL_ISO_STOP_LISTEN	IOCTL_ASYNC_LOCK
IOCTL_ISO_START_SEND	IOCTL_GENERATE_BUSRESET
IOCTL_ISO_STOP_SEND	IOCTL_ASYNC_STREAM
IOCTL_ISO_SEND	IOCTL_GET_CYCLETIME
IOCTL_ASYNC_START_LISTEN	IOCTL_GET_AVAILABLE_RESOURCES
IOCTL_ASYNC_STOP_LISTEN	IOCTL_GET_LOCAL_NODEID
IOCTL_SET_EVENT_HANDLE	

Table 5.1.1.5a Windows Raw1394 IOCTL codes

Each IOCTL code corresponds to a request that an application can send to *raw1394* which services the request as shown in figure 5.1.1.5b. The *raw1394* API decouples applications from the *raw1394* driver implementation.

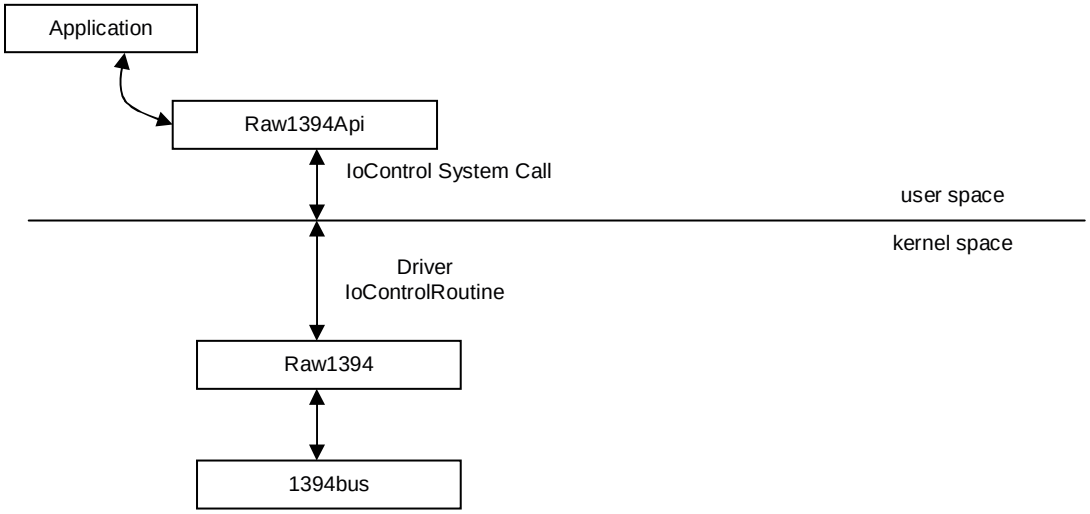


Figure 5.1.1.5b The sequence of calls from an application down to the 1394bus driver for Read, Write and Lock requests

Some of the IOCTL operations require the raw1394 driver to notify the application of some event. Since it is impossible for a kernel driver to call a user space routine from kernel space an event mechanism is used instead. For example the IOCTL_SET_EVENT_HANDLE IOCTL code is used to obtain an event handle that an application thread can use to wait for some event.

5.1.1.6 Read, Write and Lock Operations

Read and write operations are specified by the IOCTL_ASYNC_READ and IOCTL_ASYNC_WRITE codes. Each transaction requires a 16 bit destination or source node ID containing a bus ID and node ID. The transaction is also targeted towards a specified 48 bit CSR address in the node's private address space. The IRB that is sent to the 1394bus driver has its *FunctionNumber* member set to REQUEST_ASYNC_READ, REQUEST_ASYNC_WRITE or REQUEST_ASYNC_LOCK and other required members such as *destination address* set appropriately.

The structures used to communicate with user space applications are shown in figure 5.1.1.6a. An application would pass, a reference to, an instance of one of the structures to the raw1394-2 driver, which would make a copy in kernel space (using the user-kernel space memory copy function *copy_from_user*) and access the various fields within the structure. When an operation is complete, raw1394-2 copies updated data to the user space structure (using the user-kernel space memory copy function *copy_to_user*).

```
typedef struct _USER_ASYNC_DATA
{       octlet_t address;
        byte_t * buffer;
        size_t buffer_size;
}USER_ASYNC_DATA,*PUSER_ASYNC_DATA;
```

```
typedef struct _USER_ASYNC_LOCK_DATA
{       octlet_t address;
        quadlet_t arguments[2];
        quadlet_t dataValues[2];
        quadlet_t returnedData[2];
        doublet_t lockTransactionType;
}USER_ASYNC_LOCK_DATA,*PUSER_ASYNC_LOCK_DATA;
```

Figure 5.1.1.6a Structures used for read, write and lock operations by raw1394

The *address* field contains a bus ID (first 10 bits), node ID (second 6 bits) and a CSR address (last 48 bits). The *buffer* is used for storing I/O data and the buffer size specifies the size of the data buffer. An MDL (Memory descriptor List) is created with the I/O manager routine *IoAllocateMDL* using this *buffer* to/from which the 1394bus driver will transfer data using DMA. Lock requests differ from read and write transactions in that a maximum of 8 bytes of data can be operated on by a lock request whereas reads and writes can operate on much larger blocks of data. Therefore a structure specific to lock operations is used instead. A number of different types of lock operations exist (shown in table 5.1.1.6b). Each lock operation may have arguments that it uses to operate on input data and returns a result. The fields in the USER_ASYNC_LOCK structure correspond to

these. The sequence of calls that occur when an applications makes one of the above I/O operations is shown in Figure 5.1.1.5b.

ASYNC_LOCK_MASK_SWAP	For each bit in the original value and the matching argument, reset the bit to be the same as the corresponding bit in the data value.
ASYNC_LOCK_COMPARE_SWAP	If the original value and argument match, replace the original value with the data value.
ASYNC_LOCK_FETCH_ADD	Add the data value to the original value. Big-endian addition is performed.
ASYNC_LOCK_LITTLE_ADD	Add the data value to the original value. Little-endian addition is performed.
ASYNC_LOCK_BOUNDED_ADD	If the original value and the argument differ, add the data value to the original value.
ASYNC_LOCK_WRAP_ADD	If the original value and the argument differ, add the data value to the original value. Otherwise, replace the original value with the data value.

Table 5.1.1.6b Lock operation types from the Microsoft DDK.

5.1.1.7 Asynchronous Listen Operations

Asynchronous listen operations are performed for a specified address. Whenever a node performs a write operation targeted at an address, the 1394bus driver notifies raw1394 of the arrival of new data. An address range must be registered with the 1394bus driver before notification can take place.

Function code REQUEST_ALLOCATE_ADDRESS_RANGE is specified in the FunctionNumber member of the IRB sent to the 1394bus driver. Other input parameters that need to be supplied in the IRB are the address being registered, e.g. the FCP_COMMAND address where FCP command data is written to, and the size of the address range being registered e.g. for the FCP_COMMAND register the address size is 508 bytes long.

A buffer to which incoming data is written and a call-back function that is used to notify the raw1394 driver when new data arrives also needs to be specified. The buffer is not supplied directly instead an MDL (Memory descriptor List) structure is built using the buffer and passed to the 1394bus driver so that the host controller driver can perform DMA to the buffer. The 1394bus driver also allows the supplying of a FIFO list of MDLs (each built for a buffer) where incoming data is stored by the driver. This is advantageous because a large list of MDLs can be used as a back store for incoming packets instead of handling every packet separately as is done when a single MDL is supplied. The

structure shown in figure 5.1.1.7a is used by raw1394 to allow applications to perform multiple asynchronous listen operations.

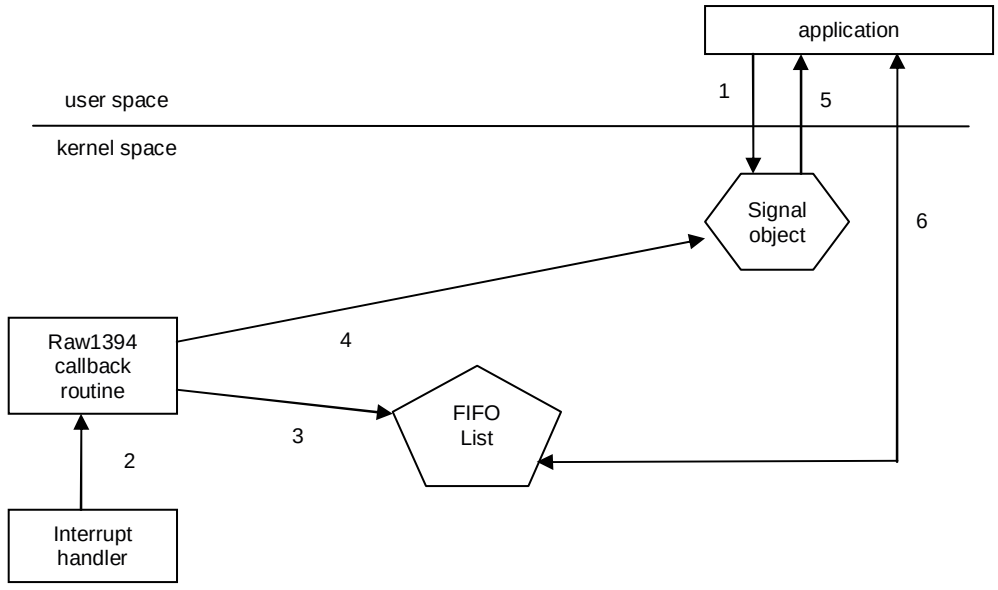
```
typedef struct _ADDRESS_RANGE_DATA
{           PADDRESS_RANGE   paddress_range;
            HANDLE    handle;
            ULONG    address_returned_count;
            nodeaddr_t start_address;
            size_t address_size;
            byte_t * data_buffer;
            size_t buffer_size;
            size_t new_data_size;
            PVOID callback;
            PMDL mdl;
            PKEVENT pAsyncDataEvent;
            ULONG packetcount;
            PUSER_ASYNC_DATA  asyncDataListHead;
            PUSER_ASYNC_DATA  asyncDataListTail;
            KSPIN_LOCK   asyncListSpinLock;
            void * next;
}ADDRESS_RANGE_DATA,*PADDRESS_RANGE_DATA;
```

Figure 5.1.1.7a Structure used for asynchronous listen operations by raw1394

paddress_range is a Windows-specific handle to a registered address. *handle* is a handle for a kernel event object that is given to an application so that the kernel can use it to notify the application when new data arrives. *callback* is the call-back routine that is called by the 1394bus driver to notify raw1394 when new data arrives. This routine executes in an interrupt context. *mdl* is an MDL for a buffer where new data will be written to. *asyncDataListHead* is the head of a linked list of incoming asynchronous packets. *asyncDataListTail* is the tail of this list. It is used to quickly add items to the end of the linked list instead of traversing the linked list every time to find its end when adding new items.

When new data comes in it is added to the tail of the list. Then a signal is sent to the application which will remove items from the head of the list as long as there are items at the head of the list. *asyncListSpinLock* is a spin lock used to ensure that accesses to the list is multi-processor safe. The *next* field is used to manage a linked list of ADDRESS_RANGE_DATA structures for each registered address range. Figure 5.1.1.7b shows how raw1394 handles incoming asynchronous packets using a queue. When raw1394 registers an address for asynchronous listening, the 1394bus driver passes the

request down to the host controller driver which will install an interrupt handler for handling incoming packets.



1- The application waits on a signal object for incoming data.
2- A new packet arrives.
3- Raw1394's call-back routine adds the new packet to the queue.
4- Raw1394 signals the application to notify it of the arrival of new data.
5- The Application removes data packets from the queue.

Figure 5.1.1.7b Incoming asynchronous packet handling by raw1394

When performing an asynchronous listen operation, an application issues an IOCTL_ASYNC_START_LISTEN code to the raw1394 driver. Next it obtains a kernel event object with IOCTL_SET_EVENT_HANDLE for new data arrival notification. It then waits on this object. When the object is signalled it uses IOCTL_GET_ASYNC_DATA to retrieve new data packets. It performs the last two steps continuously during the life of the listen operation. An application stops an asynchronous listen operation by sending the IOCTL_ASYNC_STOP_LISTEN code to the raw1394 driver and releases the handle to the kernel event object with IOCTL_UNSET_EVENT_HANDLE.

5.1.1.8 Isochronous Listen and Talk Operations

A series of steps have to be followed by a client driver for isochronous talk or listen operations to be performed with the Windows 1394 stack. An IRB setup with an appropriate function number is submitted to the *1394bus* driver as discussed in section 5.1.1.3.

The steps required for performing isochronous transactions utilising the Windows 1349 stack are:

1- Selection of an appropriate speed for the operation (100 Mbps, 200 Mbps or 400Mbps) with function number REQUEST_GET_SPEED_BETWEEN_DEVICES.
2- Allocation of a channel to perform the talk on (0 to 63) with function number REQUEST_ISOCH_ALLOCATE_CHANNEL.
3- Allocation of bandwidth with function number REQUEST_ISOCH_ALLOCATE_BANDWIDTH.
4- Allocation of a resource handle with function number REQUEST_ISOCH_ALLOCATE_RESOURCES.
5- Attaching of buffers with function number REQUEST_ISOCH_ATTACH_BUFFERS where incoming data will be written to. These buffers are MDLs to user space buffers that the host controller driver will perform DMA to.
6- Performing the talk with function code REQUEST_ISOCH_TALK in the case of a talk operation and performing a listen with function code REQUEST_ISOCH_LISTEN. This step requires the resource handle obtained in 4.

For isochronous listen operations steps 2 and 3 are not necessary. When the client driver has finished performing an isochronous talk or listen operation it should free up allocated resources as follows:

1- Stop its talking or listening with function number REQUEST_ISOCH_STOP.
2- Detach any attached buffers with function number REQUEST_ISOCH_DETACH_BUFFERS.
3- Free the resources allocated in step 4 during resource allocation with function number REQUEST_ISOCH_FREE_RESOURCES.
4- Free allocated channels with function number REQUEST_ISOCH_FREE_CHANNEL.
5- Free allocated bandwidth with function number REQUEST_ISOCH_FREE_BANDWIDTH.

To perform isochronous talking with *raw1394*, an application would issue the IOCTL_ISO_START_SEND code to the *raw1394* driver (which performs steps 1,2,3,4 and 5 from the steps required for performing Windows isochronous transactions). It then fills up its buffers and issue the IOCTL_ISO_SEND as many times as necessary (which performs step 6). When done, it issues the IOCTL code IOCTL_ISO_STOP_SEND (which performs steps 1,2,3,4 and 5 from the steps required to free up isochronous resources), so that allocated resources are freed. To perform isochronous listening, an application would issue an IOCTL_ISO_START_LISTEN code to *raw1394* (which performs steps 1, 4 and 5 from the steps required for performing Windows isochronous transactions) and listen for data. To stop listening it issues the code IOCTL_ISO_STOP_LISTEN (which performs steps 1, 2 and 3 from the steps required to free up isochronous resources). This greatly reduces the number of operations applications have to perform in order to receive or transmit isochronous data.

5.1.2 Raw1394 for Linux

A *raw1394* driver already exists in the Linux 1394 stack. This driver uses the services of the highlevel layer in the stack to perform I/O operations. It exposes a user space library called *libraw1394* to applications and implements the *read* and *write* I/O driver routines. All requests from user space to kernel space are done through the *write* system call and are asynchronous. Requests are exchanged to and from kernel space using a structure called *raw1394_request*. This structure is passed down to the kernel *raw1394* driver

when an application performs asynchronous or isochronous operations. Polling for completed requests is performed, by applications, through the *read* system call using *libraw1394*.

When the kernel *raw1394* driver completes its operations on a request, the request is placed in a completed request queue. The next time polling for completed requests is performed by an application through *libraw1394*, the application obtains the completed request from kernel space, which also contains the completion call-back routine inside it. This routine is then called to notify applications of the completion of their request. The process is shown in figure 5.1.2.

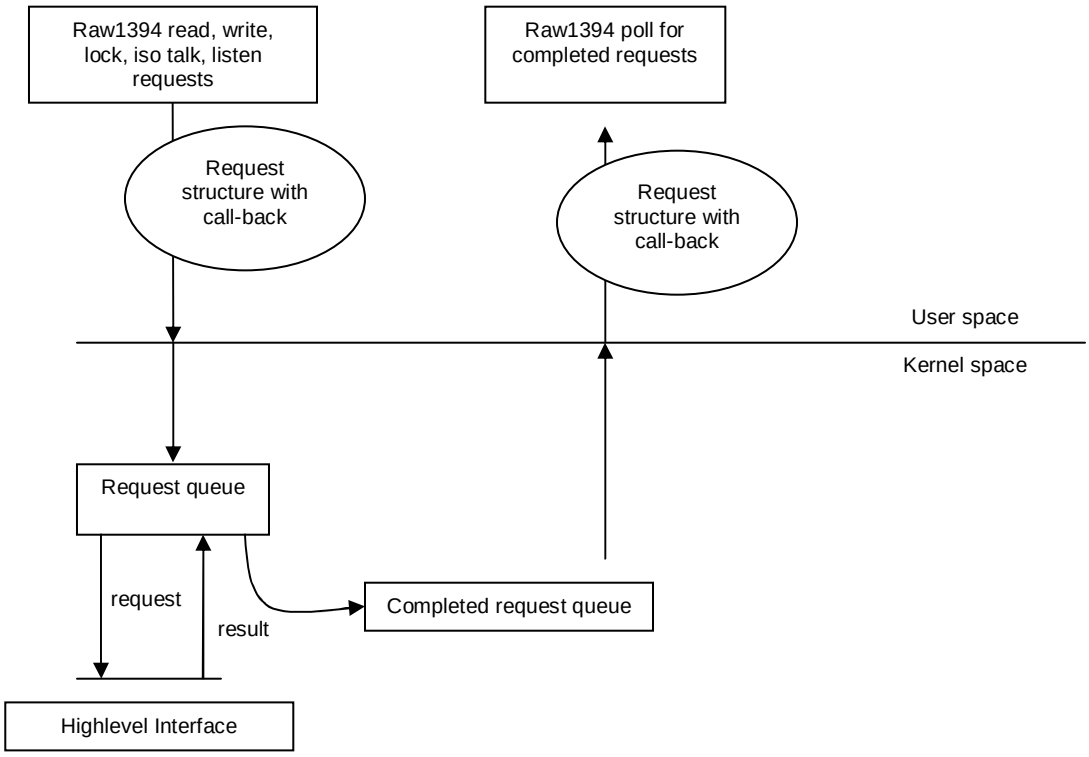


Figure 5.1.2 Raw1394's handling of requests from user space

Requests are created in user space and are sent to kernel space where they are completed and returned back. Some requests, such as isochronous listen, asynchronous listen and bus reset handler setting requests, put the *raw1394* driver in a mode whereby it has to

notify the application of incoming asynchronous listen data, isochronous listen data or bus resets. When these events occur in kernel space, requests representing the events, are created in kernel space and placed in a queue of completed requests. The application would then retrieve these requests when it next polls for completed requests.

5.1.2.1 Improving Raw1394

There are two features that are not present in raw1394 that would make it operate more efficiently, as well as make it more flexible to use. The first is the ability to perform asynchronous listen operations for any specified asynchronous address. Currently raw1394 only handles asynchronous listens for the FCP register. The second is buffering of requests so that a context switch (to and from user-kernel space) does not have to be performed for every request made. Each request is approximately 50 bytes long. Filling a large buffer e.g. a 1MB buffer, with multiple requests in user space and sending the buffer to kernel space at once, is much more efficient than sending each individual request by itself. For sending a small number of packets, the API could introduce a buffer flushing feature whereby the application can initiate sending of requests to kernel space before the buffer is filled, thereby enabling it to send requests less than the buffer size.

5.1.2.2 raw1394-2

raw1394-2 is a driver that was implemented for this project. It attempts to improve on the *raw1394* driver, described in section 5.1.2, that was developed by the Linux IEEE1394 community. *raw1394-2* is similar to the Window's *raw1394* implementation described in section 5.1.1. All of its I/O operations are implemented through the driver's IOCTL routine.

5.1.2.3 Device registration and deregistration

In its registration routine, *raw1394-2* registers its minor number with the IEEE1394 core driver, so that dispatching of I/O requests for that minor number are routed to the *raw1394-2* driver. *raw1394-2* is a driver written for IEEE1394 devices managed by the IEEE1394 core driver, which would have registered itself with the IEEE1394 driver major number. In its driver deregistration routine, *raw1394-2* deregisters its minor number with the IEEE1394 core driver. The original *raw1394* performs the same

operations as above in its registration and deregistration routines. In addition, it obtains a *highlevel* handle that it will utilise to communicate with the IEEE1394 core driver. *raw1394-2* obtains this handle at a later more appropriate time, as discussed in the next section.

5.1.2.4 Open and Release Driver Routines

In its open routine, which gets called when the *open* system call is made by an application, *raw1394* sets up application session information. It sets up two queues for the client. The first for I/O requests and the second for completed requests. In its release routine, which corresponds to the user space *close* system call, it empties its queues and frees up used resources e.g. resources used for isochronous operations.

raw1394-2 also sets up application session information. As one of its session data items it saves the *highlevel* handle that it obtains from the *highlevel* interface in its open routine. This implies that there will be one highlevel handle assigned to each application as opposed to raw1394's single highlevel handle for all applications. *raw1394-2* uses a dedicated highlevel handle for each application since it makes the handling of requests per application simpler. An application would be able to open the driver and perform I/O independent of other applications, which is not the case with raw1394 since it makes applications share a single *highlevel* handle.

When obtaining a *highlevel* handle, the *raw1394-2* driver supplies call-back routines for the various IEEE1394 I/O operations. It also supplies a call-back routine called *add host*, which will be called by the *highlevel* layer for every host controller present on the system. In this way if the system has multiple 1394 interface cards, the *raw1394-2* driver can keep track of a host controller for each card and use the host controller that a client specifies. The *release* routine for *raw1394-2* frees up resources in use by the calling application.

5.1.2.5 Global Driver Data

The global driver data (driver-application session data) structure used by *raw1394-2* uses the same name as the Windows raw1394 driver but has different fields. It is defined as

shown in figure 5.1.2.5. The first entry *phost* is the currently selected host controller for which this session data is used. *hl* is a highlevel handle for communication with the ieee1394 core driver. *hlops* is a structure containing call-back functions and data required when obtaining a highlevel handle. *pard_list_head* is list of registered address ranges for asynchronous listen operations similar to the Windows *raw1394* address range data structure. *iqd* stores state data for each of the 64 isochronous channels. *Channels_inuse* is a 64 bit bitmap of channels in use. *host_list_head* is the head of a list of host controllers. A client can select one of these hosts for performing I/O and *host count* is a count of these hosts. *Bus_reset_wait_queue* is a kernel event object used to wait for a bus reset.

```
typedef struct _DEVICE_EXTENSION {
        struct hpsb_host * phost;
        struct hpsb_highlevel *hl;
        struct hpsb_highlevel_ops * hlops;
        struct list_head pard_list_head;
        ISO_QUEUE_DATA iqd[64];
        octlet_t channels_inuse;
        spinlock_t pard_list_lock;
        struct list_head host_list_head;
        spinlock_t host_list_lock;
        int host_count;
         wait_queue_head_t busreset_wait_queue;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Figure 5.1.2.5 *raw1394-2*'s global driver data

5.1.2.6 Interactions with the IEEE1394 core driver

As discussed earlier, a client driver has to obtain a handle to the highlevel interface for communication with the IEEE1394 core driver. This is performed with a call to the highlevel routine *hbsp_register_highlevel*. The handle is freed when not needed with a call to the *hbsp_unregister_highlevel* routine. When obtaining a highlevel interface handle, the client driver (in this case *raw1394* or *raw1394-2*) supplies a structure called *hpsb_highlevel_ops* which contains call-back routines for various operations to be implemented by the client driver. This structure has the fields shown in table 5.1.2.6a

Add_host	Repeatedly called with a reference for each host controller when a highlevel handle is obtained.
remove_host	Same as above, called when the highlevel handle is freed
host_reset	Same as above, called during bus resets
Iso_recieve	Called for servicing incoming isochronous data
Fcp_request	Called for servicing incoming fcp data

Table 5.1.2.6a Linux's IEEE1394 core driver highlevel operations

raw1394-2 introduces the use of a new field into the *hpsb_highlevel_ops*. This new field is called *private_data*. This field represents driver-supplied data (called DeviceExtension in Windows). Whenever the *ieee1394_core* driver calls any of the above *hpsb_highlevel_ops* it passes the private data to them so that the client driver can make use of it. Another operation that a client driver can perform is the registration of address ranges for handling incoming asynchronous writes to these addresses. This registration is performed with a call to a routine called *hpsb_register_addrspace*. This routine takes as a parameter a structure called *hpsb_address_ops* that contains the call-back routines shown in table 5.1.2.6b

Read	Called to handle incoming reads from a registered address
Write	Called to handle incoming writes to a registered address
Lock	Called to handle incoming lock transactions to a registered address.

Table 5.1.2.6b Linux's *ieee1394_core* driver address range operations

raw1394-2 also incorporates the *private_data* field discussed earlier, into *hpsb_address_ops*. Whenever any of the read, write or lock routines are called, this data is passed to them, so that the client driver that implements them can use the session data stored in this field. Other routines available to client drivers are shown on table 5.1.2.6c.

hpsb_read	Performs an asynchronous read from a specified node id and address
hpsb_write	Performs an asynchronous write to a specified node id and address
hpsb_lock	Performs an asynchronous lock operation to a specified node id and address
hpsb_listen_channel	Starts listening on a specified channel
hpsb_unlisten_channel	Stops listening on a specified channel
hpsb_send	Send a packet on to the IEEE1394 bus
hpsb_reset_bus	Generates a software bus reset

Table 5.1.2.6c Miscellaneous IEEE1394 routines available to client drivers

A parameter common to all of the routines presented above is a structure called *hpsb_host,* which is used to represent a host controller. *raw1394-2* introduces the *private_data* field to this structure as well. Node information such as the local node ID, node count on the bus and the current generation number are kept here. Another useful entry is a structure called *hpsb_host_driver* which give host controller driver information, such as the name of the driver.

5.1.2.7 Performing I/O operations with raw1394-2

raw1394-2 defines a similar set of IOCTL codes to those used by the Windows raw1394 driver. These codes are shown in table 5.1.2.7. In contrast, the Linux *raw1394* driver serialises the messages passed between user space and kernel space, i.e. all types of I/O requests such as reads, writes and isochronous talks, as well as events such as bus resets, are exchanged between user and kernel space in one request. For example, the raw1394 driver's read routine, which is called for message dispatching, will fetch a request such as a bus reset request or data from an asynchronous write request from the completed request queue, and pass it up to application layer (*libraw1394*), which will then notify the application concerned.

IOCTL_ASYNC_READ	IOCTL_ASYNC_LOCK
IOCTL_ASYNC_WRITE	IOCTL_GENERATE_BUSRESET
IOCTL_ISO_START_LISTEN	IOCTL_GET_BUS_RESET
IOCTL_ISO_STOP_LISTEN	IOCTL_ASYNC_STREAM
IOCTL_ISO_SEND	IOCTL_GET_CYCLETIME
IOCTL_GET_ISO_DATA	IOCTL_GET_AVAILABLE_RESOURCES
IOCTL_ASYNC_START_LISTEN	IOCTL_GET_BUS_INFO
IOCTL_ASYNC_STOP_LISTEN	IOCTL_SET_HOST
IOCTL_GET_ASYNC_DATA	IOCTL_GET_HOST_COUNT
IOCTL_GET_GENERATION	

Table 5.1.2.7 Linux Raw1394-2 IOCTL codes

raw1394-2 manages message exchanges between user and kernel space in a different manner from *raw1394*. It defines IOCTL codes for each different kind of message that can be retrieved from the kernel. These are IOCTL_GET_ASYNC_DATA used to retrieve asynchronous listen write packets, IOCTL_GET_ISO_DATA used to retrieve isochronous listen data packets and IOCTL_GET_BUS_RESET used to listen for bus resets that may occur. Three different application threads can then be used to listen for kernel events that signal each of the above events.

5.1.2.8 Performing Read, Write and Lock operations with raw1394-2

Asynchronous read, write and lock transactions are performed using the *ieee1394_core* driver routines listed in table 5.1.2.6c. The structures shown in figure 5.1.2.8 are used for requests exchanged between applications and *raw1394-2*. The field called *address* represents a 64 bit address containing a node number (6 bit), bus number (10 bit) and destination or source CSR address (48 bit). The *buffer* is a pointer to a user space virtual address, and buffer size is the size of this buffer. Lock transactions are different because they are of a fixed length, 32 bit for an ordinary lock transaction and 64 bit for a lock 64 transaction. Data in the field named *arguments* is one of the operands for the lock operation. The field named *returned_data* contains the results of the lock transaction. The transaction type specifies the lock transaction type and the *lock64* field indicates whether this is a 64 bit lock operation or not. Lock transaction types are discussed in section 5.1.1.6.

```
typedef struct _USER_ASYNC_DATA
{       octlet_t address;
        size_t address_size;
        byte_t * buffer;
        size_t buffer_size;

}USER_ASYNC_DATA,*PUSER_ASYNC_DATA;
```

```
typedef struct _USER_ASYNC_LOCK_DATA
{   octlet_t address;
    quadlet_t arguments[2];
    quadlet_t data_values[2];
    quadlet_t returned_data[2];
    int lock64;
    doublet_t lock_transaction_type;
}USER_ASYNC_LOCK_DATA,*PUSER_ASYNC_LOCK_DATA;
```

Figure 5.1.2.8 Structures used for read, write and lock operations by *raw1394-2*

Data to and from the user space buffer is transferred using two user-kernel space memory copy routines called *copy_to_user* and *copy_from_user*. Unlike the Windows 1394bus driver the Linux *ieee1394_core* driver does not yet perform DMA to user space buffers. *raw1394-2* also handles large read and write requests by breaking them up into smaller packets, whose size is determined by the maximum asynchronous packet size supported by the host controller.

5.1.2.9 Asynchronous Listen Operations

As mentioned in the previous section, DMA to user space buffers is not performed by the Linux *ieee1394_core* driver. In addition, as in the case of the Windows *raw1394* driver, asynchronous write packets are passed to the client driver in an interrupt context. At this

time, accessing a user space virtual address is not possible, therefore *raw1394-2* manages a queue of incoming asynchronous write packets. Figure 5.1.1.7b shows how incoming asynchronous write packets are handled for the Windows raw1394 driver and the same applies for *raw1394-2*.

The structure shown in figure 5.1.2.9 is used for managing registered addresses for which asynchronous write transactions from remote nodes are handled by *raw1394-2*. The structure is similar to the Windows *raw1394* PADDRESS_RANGE_DATA structure. *struct list_head* is an entry that makes this structure usable with the Linux kernel's linked list implementation. A list of *paddress_range_data* structures is kept inside each application's session data structure. *start_address* specifies the start of the registered address range and *address_size* its size. *hpsb_address_ops* is a structure containing driver implemented routines that will be called to handle write, read and lock operations targeted at the registered address ranges. *async_write_wait_queue* is a kernel event object used for signalling the arrival of new data to an application thread that waits on that event object. The field named *data_queue* is the queue where incoming packets are stored and the field named *data_queue_lock* is used to ensure multiprocessor safe accesses to this queue.

```
typedef struct _ADDRESS_RANGE_DATA
{       struct list_head list_index;
        nodeaddr_t start_address;
        size_t address_size;
        struct hpsb_address_ops * async_address_ops;
        wait_queue_head_t async_write_wait_queue;
        struct list_head data_queue;
        spinlock_t data_queue_lock;
}ADDRESS_RANGE_DATA,*PADDRESS_RANGE_DATA;
```

Figure 5.1.2.9 Structure used for asynchronous listen operations by raw1394

5.1.2.10 Isochronous Talk and Listen Operations

The Linux *ieee1394_core* driver exposes a much simpler interface for client drivers than the Windows *1394bus* driver. *raw1394-2* implements its own isochronous talk operations using the *ieee1394_core* driver's *hpsb_send_packet* routine in the same way as *raw1394*, except that *raw1394-2* allows applications to specify a large buffer which it will break into a frame size specified by the application. This feature is not present in *raw1394*.

Isochronous listening is performed with a call to *hpsb_listen_channel*, which requires a *highlevel* handle. This handle is obtained from the *raw1394-2* driver routine that handles the application's *open* call, at which time an isochronous call-back routine is supplied. This routine will be called to service any incoming isochronous requests in an interrupt context. Incoming packet handling is performed as in the Windows r*aw1394* driver as shown in figure 5.1.1.7b.

5.1.2.11 Linux1394's New Raw ISO API

As of (19/11/2002), the Linux 1394 developers released a new isochronous API called *rawiso*. Its operation differs from the old API described in section 5.1.2.10. Table 5.1.2.11 lists the new API routines. The main change is that isochronous transmission is handled a little differently. *hpsb_iso_xmit_init* is supplied with a callback routine, which when called provides the data to be transmitted when the client driver is ready to send a packet. Other parameters to this routine are the maximum packet size the driver will use, the channel, transmission speed and the number of packets to be buffered. A call to the *hpsb_iso_xmit_start* routine will start isochronous transmission. Isochronous listening is performed similarly.

hpsb_iso_recv_init is used to initialise isochronous listening. It requires a call-back routine that will be called to handle incoming isochronous packets, the maximum isochronous packet size, the number of packets to be buffered and the channel number to be used for the receipt. *hpsb_iso_recv_start is* used to begin isochronous listening. Isochronous resources used by the driver are freed with the *hpsb_shutdown* routine. Isochronous transmission can be stopped without freeing up driver allocated resources with the *hpsh_iso_stop routine*. The mainstream IEEE1394 Linux kernel driver (released with kernel version 2.4.19) does not have *rawiso* support in it, only the latest drivers available from the Linux1394 community do.

hpsb_iso_xmit_init	Initialises isochronous resources for transmission.
hpsb_iso_recv_init	Initialises isochronous resources for receipt
hpsb_iso_xmit_start	Starts isochronous transmission
hpsb_iso_recv_start	Starts isochronous receipt
hpsb_iso_stop	Stops isochronous transmission or receipt
hpsb_iso_shutdown	Frees up all isochronous resources used up by iso transmissions or receipts.

Table 5.1.2.11 The new Linux 1394 rawiso API

5.1.3 The Windows raw1394 and the Linux raw1394-2 driver Implementations Compared

The *raw1394* for Windows and the *raw1394-2* for Linux driver implementations discussed so far are very similar, except where the operating system's implementation of the core IEEE 1394 drivers require special interaction from client drivers.

When performing data transfers, the Windows *1394bus* driver can perform DMA to and from user space buffers and requires an MDL (memory descriptor list structure) for a specified user space buffer. Most operations are passed from the 1394bus driver to the host controller driver, which is specific to the 1394 interface hardware in use. The Linux IEEE 1394 driver performs DMA to a kernel space buffer, which is passed to the client driver handling the specific transaction for which the buffer is used for. The client driver then copies data from this buffer to an application buffer.

As new data arrives, the client driver on each operating system is informed of the event through a call-back routine. It is supplied with a buffer containing the data and other related information such as the source address, if it is an asynchronous transaction, or a channel number, if it is an isochronous transaction, and a buffer size.

On both operating systems, kernel event objects are used to communicate the new data arrival event to applications. In Windows, the application obtains a kernel event object (called KEVENT) handle from kernel space and returns to user space where it waits for events using this handle. When this event object is signalled, the application issues IOCTL requests to obtain the new data from kernel space. It then goes back to listening on the kernel event handle for more events. In Linux, event notification is handled differently, since the same facilities as Windows do not exist in Linux i.e. the ability to wait on kernel event objects in user space. The application issues an IOCTL call and waits on a kernel event object (called wait_queue) in kernel space. This waiting is interruptible if need be, e.g. when an application process is terminated. When new data arrives, the kernel event object is signalled. The *libraw1394-2* API thread retrieves the new data, returns to user space, passes the new data to the application, and returns to kernel space to listen for more events.

Isochronous operations with the Window *1394bus* driver require a set of steps such as the allocation of channel, bandwidth, and buffer attachment to be performed by the client driver. In the Linux *ieee1394* core driver these steps are not necessary. Buffering strategies for data packets are left for client drivers to implement.

5.2 Moving the Linux 1394 driver stack towards IEEE1394.1 bridge awareness

This section presents implementation changes that have been identified to make the current Linux 1394 driver stack IEEE1394.1 bridge aware.

5.2.1 Indication of bridge awareness

Node bridge awareness is indicated by setting bit number 28 of the bus information block inside a node's configuration ROM to 1. Configuration ROM initialisation takes place in the host controller driver. For the OHCI driver this is done in the routine called *ohci_initialize*.

5.2.2 Bus reset and quarantine

Quarantine periods are obeyed by bridge aware nodes after every bus reset, at which time each node must set bit 0 and 1 of its quarantine register to 1. At the end of the quarantine period a bridge will clear these bits by setting them to 0 at which time the node can continue its operations. The quarantine register is located at CSR address 0x214. Register definitions are done in csr.h and csr.c in the current Linux 1394 driver. In csr.h a new field entry for the quarantine register is made in a structure called *csr_control*, where all node implemented CSR registers reside. In csr.c, where reads and writes to these registers are handled, read and write request handling for the quarantine register is added. Figure 5.2.2a shows the values the quarantine register can attain when events such as read and write requests occur. Bit 0 is called the orphan bit and indicates whether a quarantine period exists, bit 1 is called the net updated field. It indicates whether or not a net update is in progress i.e. the network topology is changing.

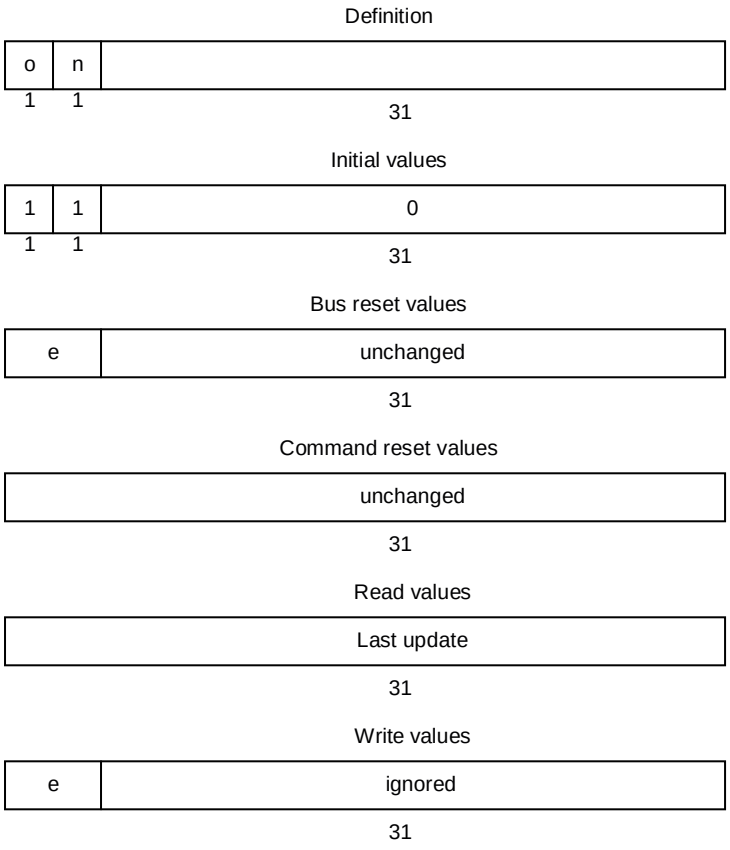


Figure 5.2.2a The quarantine register (32 bit), from section 5.2.1 of the IEEE1394.1 specification, and its different states.

The time when the values of the quarantine register can be modified directly by a node is when it picks up a self id packet, originating from a node that is a bridge, after a bus reset. Bits 19 and 20 indicate whether the self id packet was sent from a bridge or an ordinary node. The location of the bridge ID flag in self ID packets is shown in figure 5.2.2b. If the value in bits 19 and 20 is 10 (binary) then the self ID packet is from a bridge and the network topology is unchanged. If the value is 11 then the self ID packet is from a bridge and the network topology has changed. Referring back to figure 5.2.2a, during bus resets the field indicated by “e” will be set to either 10 or 11 respectively i.e. 10 would set the orphan bit to 1 and net update field to 0 ,11 would set the orphan bit to 1 and the net update field to 1.

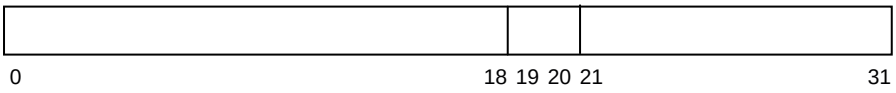


Figure 5.2.2b Location of bridge flag in the first quadlet of self ID packets.

5.2.3 Differentiation of remote from local packets

When packets are received, their destination ID fields are inspected and rejected if their bus ID is not equal to the local bus ID (1023). This ensures that local nodes will not pickup packets that are not sent to them. In the *ieee1394_core* driver, in file *ieee1394_core.c*, the *hpsb_packet_received* routine that gets called to handle all incoming packets is modified to perform the above check i.e. if the destination bus ID is not equal to 1023 the packet is dropped.

5.2.4 Bridge management messages

Bridge management messages are messages encapsulated in block write packets and targeted at the message response and request CSR registers (which are required to be implemented by all bridge aware nodes). The maximum size of a message is 64 bytes. Modifications to the current driver were made to make handling of write transactions to the message request and response registers possible from application space. This modification mirrors the existing implementation of write transaction handling for the fcp command and response registers.

5.2.4.1 Snarf Field

A new snarf field is introduced in block write packet headers as shown in figure 5.2.4.1. If the value of the snarf field is 0, bridges will forward the packet to its destination without modifying its contents. If the snarf field value is 1, the packet will be intercepted by the bridge located on the same bus as the destination node (terminal bridge) and the bus ID modified to be 1023 (local bus) then re-transmitted on that bus so that the local node can pick it up. If the snarf field value is 2, the bridge on the same bus as the source node will intercept it and change the source bus ID to be the same as the source bus ID instead of the local bus ID (1023). This is necessary since packets originating from nodes on the local bus have source bus ID of 1023. If the value of the snarf field is 3 the packet

is intercepted by all bridges on the route to the destination, and the message contained with the data payload processed.

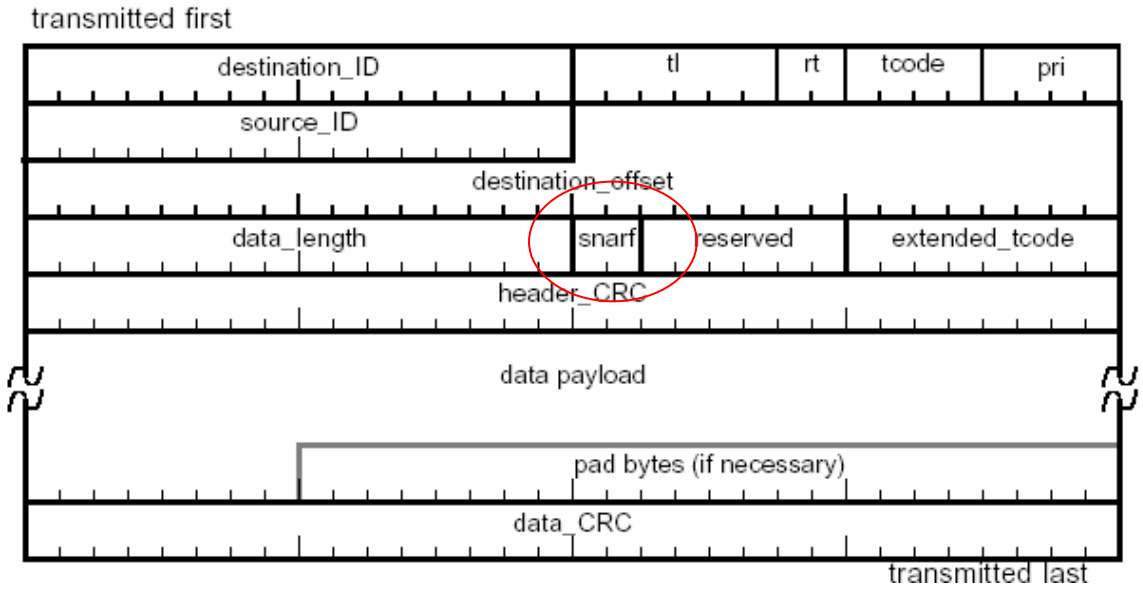


Figure 5.2.4.1 Snarf field definition in block write packet header.

No facility to set the value of the snarf field exists currently. Packets are formatted, before being sent, inside the *ieee1394_core* driver routine called *hpsb_make_writepacket*. A structure called *hpsb_packet* is used to represent packets before they are transmitted on the bus. A field named *header* within this structure contains packet header data. Setting of the *snarf* value in this field was added to the driver. Changes were also necessary in the *raw1394* driver in places where it makes calls to *hpsb_make_writepacket* to make it supply a value for the *snarf* field. In addition a snarf field was added to the structure used for exchanging I/O request data by *raw1394* with user space applications. This structure is called *raw1394_request*. *libraw1394* was modified to provide overloaded routines that enabled applications to specify snarf field values for block write requests.

5.2.5 IEEE 1394.1 implementation

The Linux *raw1394-2* driver, created during this investigation, allows the sending and receiving of IEEE1394.1 bridge messages. The modifications described in sections 5.2.1 to 5.2.3 and section 5.2.4.1 would have to be made to the Linux IEEE1394 drivers for full IEEE1394.1 compliance. Since the IEEE1394.1 specification is still in draft phase, which

implies that its requirements may change, and no hardware is available for testing, changes necessary to these drivers to make them bridge aware have not been implemented during this investigation.

5.3 IEEE1394 Software

During the investigation of the Windows and Linux IEEE1394 driver stacks, two test drivers, presented in section 5.1.1 and section 5.1.2, and a number of test applications were developed. This section presents a demonstration of each of these pieces of software and describes their functionality. Figure 5.3 presents a diagrammatic view of the software developed.

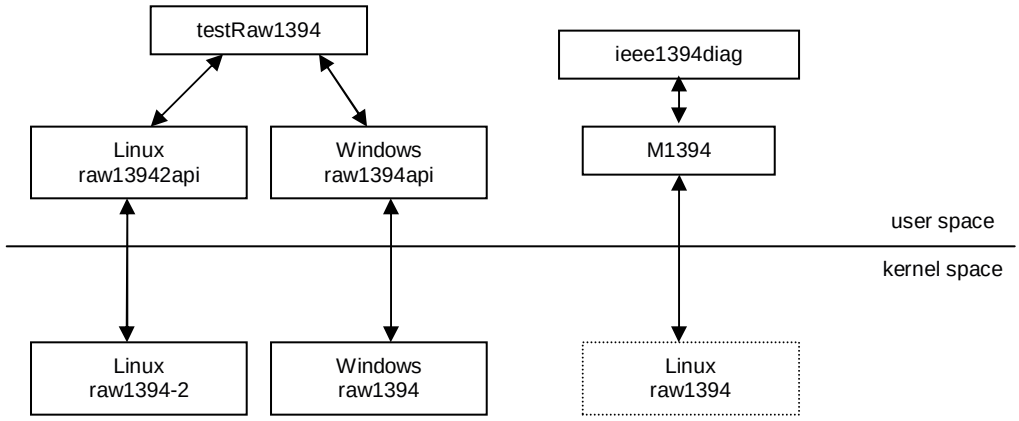


Figure 5.3 A Diagrammatic view of software produced during investigation of the Windows and Linux IEEE1394 driver stacks.

5.3.1 Linux libraw1394 Wrapper Class (M1394)

The first piece of software produced during this project was a wrapper class, called *M1394*, which simplifies the use of the publicly available Linux IEEE1394 C library, *libraw1394*. It offers a clear interface to *libraw1394* that applications can use. For example, *libraw1394* requires its clients to acquire a handle to be able to perform IEEE1394 operations. Operations such as reads and writes use a 16 bit number to represent a target node ID and a bus ID, thereby requiring application programmers to create the 16 bit target ID themselves. The *M1394* class offers application programmers

the benefits of object orientated programming. All the details of dealing with IEEE1394 hardware are handled with a single object. Once clients create an instance of an *M1394* class, the methods available from it (these closely mirror *libraw1394*) can be used to perform IEEE1394 operations. Target node IDs for operations are specified using two separate variables instead of a single 16 bit one, i.e. applications specify a bus ID and a node ID. The next two sections present examples that demonstrate how the *M1394* class is simpler to use than the *libraw1394* library.

5.3.1.1 An example IEEE1394 operation with libraw1394

The code listed in figure 5.3.1.1 shows how the string “Hello world!” can be written to the FCP command register of an IEEE1394 node using *libraw1394*. The program first obtains a *libraw1394* handle. This handle is necessary for making calls to libraw1394 routines, as these routines operate on the information contained in that handle, which is opaque to applications. Once a handle is safely obtained, the first 1394 interface card present on the PC is selected using the *raw1394_set_port* routine. If this call is successful, a buffer containing the string “Hello world!” is written to node 1023:0’s FCP command register. The destination node ID is represented with a 16 bit number. The first 10 bits are the bus ID, 0x3ff (1023) in the example, and the lower 6 bits specify the node ID, 0 in the example. A 10 bit left shift and an OR operation are used to construct the 16 bit node ID.

```
raw1394handle_t handle;
char * data = “hello world!”;
handle = raw1394_new_handle();

if (!handle)
{ perror("failed to get a raw1394 handle");
  exit(0);
}
if (raw1394_set_port(handle, 0) < 0) {
                perror("could not select a 1394 card to use for the test");
                exit(0);
        }
raw1394_write(handle, (0x3ff<<10) | 0,
                      FCP_COMMMAND_ADDRESS, sizeof(data),(quadlet_t*)data);
```

Figure 5.3.1.1 writing a short message to the FCP command register using *libraw1394*

5.3.1.2 An example asynchronous operation with the M1394 class

The same operation as described in section 5.3.1.1, performed using the *M1394* class, is shown in figure 5.3.1.2. An instance of the *M1394* class, *rd*, is created and the *isOk* member of *M1394* called to check if *libraw1394* was initialised successfully. The *fcp_send_command* routine is then called to perform the FCP write operation.

```
M1394 rd;
char * data = “hello world!”;

if(!rd.isOk())
{ perror("failed to initialise raw1394 device");
  exit(0);
}

rd.fcp_send_command(0,1023,data,sizeof(data));
```

Figure 5.3.1.2 Writing a short message to the FCP command register using the *M1394* class

5.3.1.3 An example isochronous operation with the M1394 class

Figure 5.3.1.3 shows how an isochronous operations can be performed using the Linux *M1394* class. Isochronous transmission requires a callback function that will be called to fill a buffer with data before the buffer’s contents are transmitted. The routine *set_iso_xmit_handler* sets this callback function. Transmisstion is then initialised with the routine *iso_xmit_init*. Finally data transmission is started using the routine called *iso_xmit_start*, which would stream the string “Hello World!” to an isochronous channel continuously.

```
// isochronous data sending callback function
int send_iso_packet(unsigned char *data,
                      unsigned int *len, …)
{
     char * msg = “Hello World!”;
     memcpy(data,msg,sizeof(msg));
     *len = sizeof(msg);
     return 0;
}
```

```
…

M1394 rd;
if(!rd.isOk())
{ perror("failed to initialise raw1394 device\n");
  exit(0);
}

rd.set_iso_xmit_handler(send_iso_packet);
rd.iso_xmit_init(packets_to_buffer,max_frame_size,channel,rd.bus_speed(speed));
if(!rd.isOk())
       {      printf("iso_transmit_init failed\n";
              exit(0);
       }

rd.iso_xmit_start();
if(!rd.isOk()) {
              printf("iso_transmit_start failed\n");
              exit(0);
       }
```

Figure 5.3.1.3 Performing an isochronous talk operation with the *M1394 class.*

5.3.2 Windows Raw1394 Driver and API

On the Windows platform, no application level IEEE1394 library, such as *raw1394*, exists apart from an IEEE1394 example API supplied with the Windows DDK, which is not functional. Applications are not able to perform IEEE1394 operations without creating their own client driver and user space library. To address this and at the same time demonstrate the use of the API exposed by the Windows IEEE1394 bus driver, a client driver called *raw1394* was constructed. Figure 5.3.2 shows the client driver loaded on a Windows system. A user space API that decoupled applications from the driver was also constructed. The API offers a much simpler interface than the Window's bus driver offers. For example, it hides from applications the complexities of performing isochronous operations with the Windows IEEE1394 stack (discussed in section 5.1.1.8).

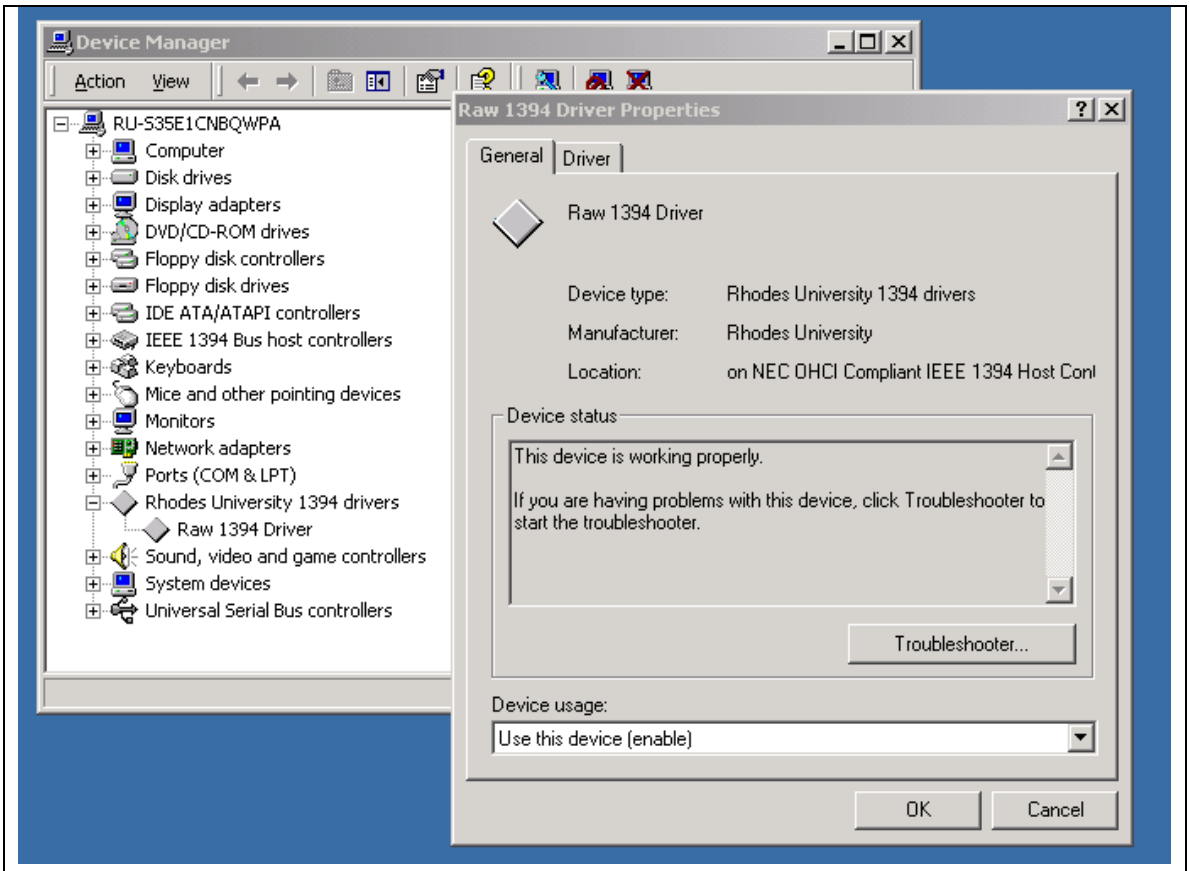


Figure 5.3.2 The newly constructed Windows raw1394 driver

5.3.2.1 An example asynchronous operation with the Windows raw1394 API

Figure 5.3.2.1 shows how the string "Hello world!" can be written to the FCP command register of an IEEE 1394 node using the Windows *Raw1394* API created during this investigation. An instance, rd, of the *Raw1394* class is created and the *isOk* member of *Raw1394* called to check if *Raw1394* was initialised successfully. The *async_write* routine is then called to perform the FCP write operation, given the destination node ID 0, bus ID 1023 and the FCP register address.

```
Raw1394 rd;
byte_t * data=  "Hello World";

if(!rd.isOk())
{ printf("Could not open raw1394 device \n");
  exit(0);
}
```

```
rd.async_write(0,1023,CSR_REGISTER_BASE+CSR_FCP_COMMAND,
                              data_buffer,sizeof(data));
if (!rd.isOk())
  { printf("raw1394 async write failed\n");
    break;
  }
```

Figure 5.3.2.1 Writing a short message to the FCP command register using the Windows Raw1394 class.

5.3.2.2 An example isochronous operation with the Windows raw1394 API

Figure 5.3.2.2 shows how an isochronous talk operation is performed with the Windows *raw1394* application API. An instance of the *raw1394api* class *Raw1394*, *rd*, is created. Three buffers each 320 bytes in size are initialised. Isochronous resources are allocated with the *iso_start_send* member of the *Raw1394* class. The talk operation is perfomed with a call to the *iso_send* routine, which transmits the data contained in the three buffers to a specified channel. Since the packet size specified is 32 bytes, there will be (3x320)/32=30 isochronous packets transmitted on the bus with a single *iso_send* routine call. The *iso_stop_send* member is used to halt isochronous transmission.

```
Raw1394 rd;
if(!rd.isOk())
        {       printf("Could not open raw1394 device \n");
                exit(0)
        }

doublet_t channel=0;
size_t packetsize=32;
size_t buffersize=320;
size_t buffercount=3;

byte_t ** buffer;
buffer = new byte_t* [buffercount];
size_t * sizes = new size_t[buffercount];
for(unsigned int i=0;i<buffercount;i++)
{       buffer[i]=new byte_t[buffersize];
        sizes[i] = buffersize;
}

rd.iso_start_send(channel,SPEED_1394_400,1,0,packetsize,buffer,sizes,buffercount);
if(!rd.isOk())
```

```
        {       printf("failed to on iso start talk\n");
                return;
        }
else printf("sending on iso channel %d\n",channel);

rd.iso_send(channel);
if(!rd.isOk())
        { printf("failed to on iso send\n");
          return;
        }

rd.iso_stop_send(channel);
if(!rd.isOk())
        { printf("failed to on iso stop send\n");
          return;
        }
printf("iso_stop_send complete\n");
```

Figure 5.3.2.2 Performing an isochronous talk operation with the Windows *raw1394* API.

5.3.3 Linux Raw1394 Driver and API

Examination of the existing Linux *raw1394* driver showed that although it handled most IEEE1394 operations, it lacked some useful features (discussed in section 5.1.2.1) essential to IEEE1394 applications. For example it does not offer an interface for applications to register address ranges and handle I/O operations targeted at the registered addresses. A second driver, called *raw1394-2*, was created to address this as well as demonstrate the use of the Linux IEEE1394 driver stack. Figure 5.3.3 shows the driver loaded on a Linux system. The output was obtained using the system program *lsmod*, which lists all modules currently loaded in the kernel. A user space API called *libraw1394-2* that closely matches the Linux user space *raw1394* API was also constructed for application use.

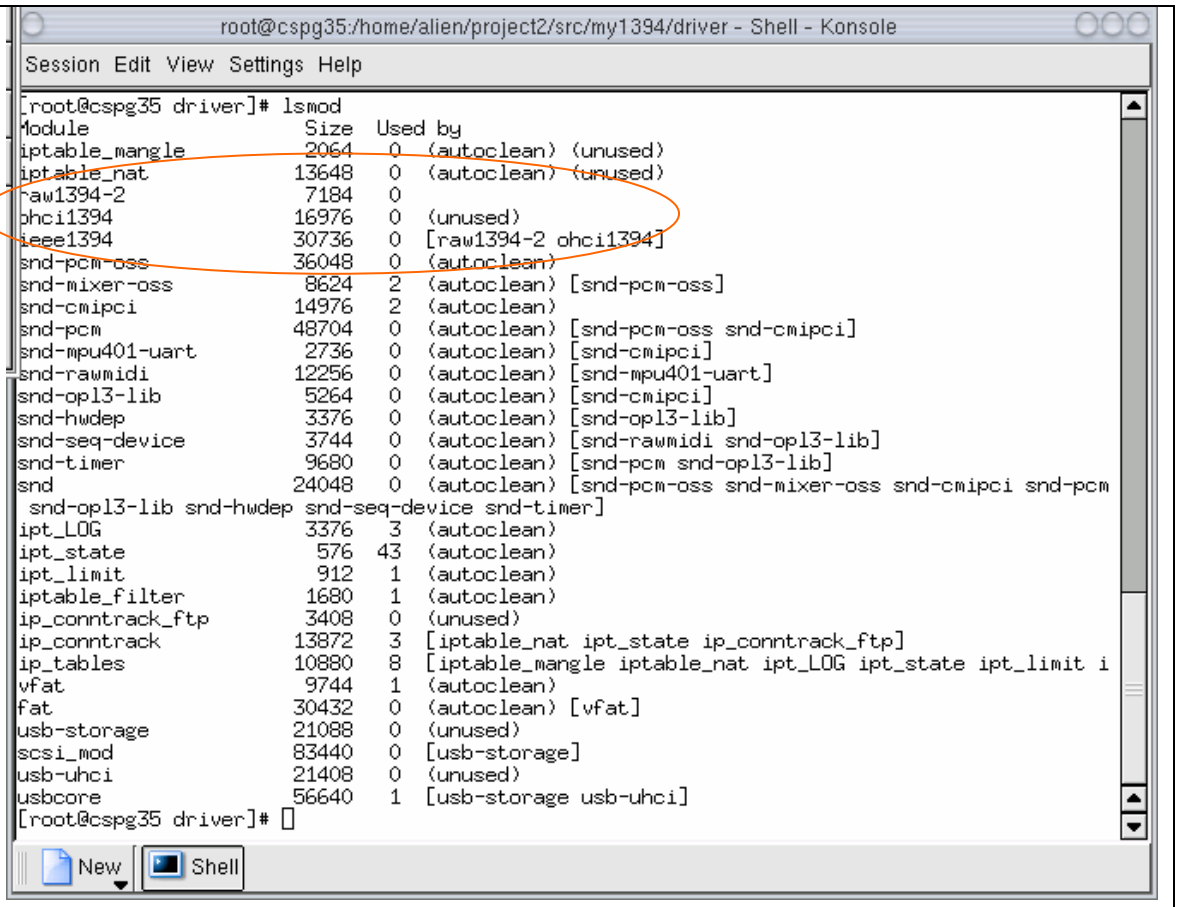


Figure 5.3.3 The newly constructed Linux *raw1394-2* driver

5.3.3.1 An example asynchronous operation with the Linux raw1394-2 API

Figure 5.3.3.1 shows how the *libraw1394-2* API can be used to perform the same operation performed by the Windows *Raw1394* example of section 5.3.2.1. The routine *raw13942_async_write* is used to send the string "Hello World!" to the FCP register of an IEEE 1394 node. The string is sent to node ID 0, bus ID 1023. *raw1394-2* supports buffering, which is one of the enhancements it offers over Linux's *raw1394* driver. In the example, the data size is the same as the buffer size, which means only a single frame is present in the buffer.

```
raw13942_handle_t handle = raw13942_create_handle();
byte_t  * data = "Hello World";

if(!raw13942_isok(handle))
{    printf("raw13942_create_new_handle failed");
     exit(0);
```

```
}

size_t datasize = sizeof(data);
size_t buffersize = datasize;
raw13942_async_write(handle,0,1023,CSR_REGISTER_BASE+CSR_FCP_COMMAND,
                                    data,buffersize,datasize);
if(!raw13942_isok(handle))
{   printf("raw13942_async_write failed");
    exit(0);
}
```

Figure 5.3.3.1 Writing a short message to the FCP command register using the Windows raw1394-2 API.

5.3.4 IEEE1394diag

The final piece of software produced during this project was a GUI application called *ieee1394diag*. This program is a diagnostic application that allows users to easily perform IEEE1394 operations on a 1394 bus. When run, *ieee1394diag* appears as shown in figure 5.3.4. It obtains the number of IEEE1394 nodes on the bus from the Linux IEEE1394 driver through *libraw1394*, reads the configuration ROM of each of the nodes, parses the configuration ROM data and presents it as a browse-able tree as shown in figure 5.3.4. For example, figure 5.3.4 shows two 1394 nodes on the bus, at the time the application was run. The first is a Linux PC (Linux-OHCI1394) and the second is a Windows PC (Microsoft).

Other operations *ieee1394diag* can perform are:

- Asynchronous reads, writes and locks to and from specified nodes on the bus targeted at a CSR address.
- FCP data and isochronous listening.
- Isochronous data transmission to a channel.
- Transmission of data from a data file on disk and saving of incoming data to a data file on disk.

ieee1394diag was released under the GNU public license (GPL) to the public as the first application of its kind. Project facilities provided by *sourceforge.net* were used to host it [Tsegaye, 2002]. As required by the GPL license all the source code for *ieee1394diag* as

well as the *M1394* class has been released to the public so that other developers can make improvements to them.

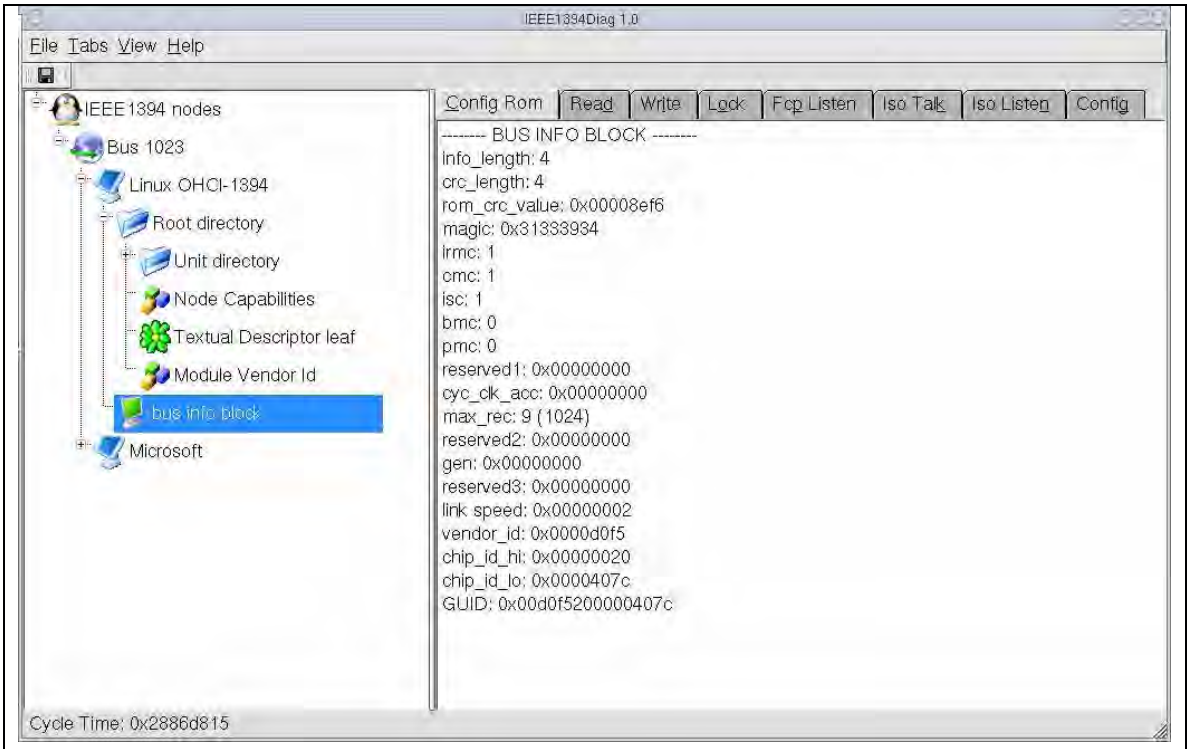


Figure 5.3.4 *ieee1394diag*, an IEEE1394 diagnostic application

5.4 Summary

This section presented the current implementations of the IEEE1394 driver stacks on the Linux and Windows Operating systems. Existing client driver implementations were discussed, as well as the implementation of two new client drivers. The first driver is called *raw1394*, implemented for the Windows IEEE1394 stack and allows generic IEEE1394 operations on an IEEE1394 bus. The second driver is called *raw1394-2*, implemented for the Linux IEEE1394 stack, and performs the same operations as the existing *raw1394* driver, but offers enhancements such as address range registration/ deregistration facilities per application, as well as I/O request buffering for improved I/O performance. Implementation changes that are required to make the Linux IEEE1394 stack IEEE1394 bridge-aware are also presented. The chapter concludes by providing a look at the user space APIs and test applications developed for the client drivers implemented during this project.

6 IEEE1394 Driver Performance Tests

The IEEE1394 device driver implementations of the Linux and Windows operating systems, as well as new IEEE1394 client driver implementations were discussed in chapter 5. When these drivers are used by applications to perform IEEE1394 operations, the performance and stability of each driver is different. Tests need to be conducted to measure the relative performance of each client driver, which in turn measures the performance of the IEEE1394 stack of each operating system since client drivers utilise the operating system's underlying IEEE1394 stack. The stability of each driver is measured by its ability to operate without problems during all the tests.

Most hardware benchmarking sites, such as [Tom's hardware, 2002], [Anandtech, 2002] run tests to evaluate the performance a new piece of hardware that is being introduced by a manufacturer. No matter how well the hardware has been designed, it needs drivers that make it operate at peak efficiency. Therefore, these benchmarks also test the performance of the device drivers supplied with the particular driver since they make the hardware function. Benchmarks, such as the one performed by Schimid [Shimid, 2002] use standard benchmarking programs such as Winmark [Winmark, 2002] and Sandra [Sandra, 2003].

The tests presented in this chapter, are designed to test the IEEE1394 bus utilisation time by the IEEE1394 drivers from the two operating systems and the experimental drivers created during this investigation using a dedicated IEEE1394 bus analyser. From the bus utilisation time, the maximum data transfer rate achieved by the driver can later be determined. The IEEE1394 bus analyser consists of hardware that snoops data from the IEEE1394 bus and performs precise timing of the time taken to perform asynchronous and isochronous data transfers. Standard benchmarks such as Sandra [Sandra, 2002] do not attain the level of accuracy the bus analyser provides. They are general purpose benchmarks and are not designed for specifically testing the IEEE1394 bus. The tests presented here were conducted to show how close to the theoretical maximum data transfer rate of 50MB/s, the data transfers performed with the IEEE1394 drivers are.

The tests in this section were conducted to show the performance of the current Linux and Windows IEEE1394 driver stacks as well as the newly created IEEE1394 client drivers, *raw1394* for Windows and *raw1394-2* for Linux. It must be noted that these drivers are experimental drivers and not production drivers.

6.1 Test Conditions

A number of IEEE1394 nodes and various combinations of IEEE1394 client drivers were used while conducting the tests. This section presents the hardware used, the versions of software (drivers and operating systems) used and the conditions under which the tests were run.

6.1.1 Hardware Used

Timing data for the tests were obtained by using a Firespy 400 IEEE1394 bus analyser (software version 2.1.1) manufactured by dapdesign. The timing data was precise within a nanosecond [dapdesign, 2002]. During the test, three IEEE1394 nodes were present. A Linux PC (256MB RAM, Redhat 7.2, Linux kernel 2.4.19, 1Ghz Celeron processor, OrangeLink S400 1394 card), a dual boot PC (256 MB RAM, Windows 2000 SP3/ Redhat 7.2 Linux kernel 2.4.19, Dual Pentium III 800Mhz, OrangeLink S400 1394 card) and an mLAN enabled Yamaha S80 synthesiser (S200 node, did not take part in the tests). All transmissions were made at S400 (400Mbps). The Windows PC was physically connected directly to the Linux PC and FireSpy bus analyser, so the speed between the three S400 nodes was always S400.

6.1.2 Software Used

On Windows, IEEE1394 drivers supplied with Windows 2000 service pack 3 were used. On Linux, IEEE1394 drivers supplied with kernel version 2.4.19 were used. Testing of the newly created Linux *raw1394-2* driver with the IEEE1394 driver supplied with kernel version 2.4.19 was not possible, as *raw1394-2* used facilities provided by the current CVS archive snapshot of the IEEE1394 drivers, such as address range deregistration. For tests that involved *raw1394-2*, Linux IEEE1394 driver version *ieee1394-550* obtained from the CVS archive was used. The changes described in Chapter 5 section 5.1.2.5 were made to this driver in order to make *raw1394-2* function.

6.1.3 The Tests

Command line programs were constructed to perform:

- Asynchronous transmission from both Windows and Linux with a specified frame size and buffer size, if buffering was supported by a driver. Transmissions were made to the FCP command register.
- Isochronous transmission from both Windows and Linux with a specified frame size and buffer size, if buffering was supported by a driver. Transmissions were made on channel 0.

6.1.3.1 Asynchronous Tests

A measure of asynchronous reception by each driver was evaluated by the time taken for each packet to be acknowledged by the receiving driver. The time recorded for the asynchronous transmission tests includes the time taken for a packet to reach its destination, plus the time taken for an acknowledgment packets (generated by the IEEE1394 driver of the receiver node) to be received by the transmitter node.

6.1.3.2 Isochronous Tests

The bus analyser could not be used to measure isochronous reception times by each driver since isochronous packets are not acknowledged by nodes that listen on a channel. Only the time taken for data to be completely transmitted by a driver on a channel was measured. Isochronous reception with the Windows driver could not be performed at the time of testing, even though the DDK API documentation states that the Windows driver is capable of isochronous reception. Correspondence with Microsoft indicated that the Windows XP operating system contained a later version IEEE1394 stack than Windows 2000, which suggested that isochronous reception might work with Windows XP. However isochronous reception tests conducted on Windows XP were unsuccessful.

6.1.3.3 Data Sizes

All data sizes indicated include packet header and CRC information. Although the client drivers were supplied data payload sizes of 32, 64, 128, 256, 512, 1024 and 2048 bytes (where a driver supports it), actual frame sizes including header and CRC data are 12 byte more for isochronous packets and 24 bytes more for asynchronous packets. These

frame sizes were chosen because IEEE1394 applications use frame sizes that are somewhere in this range (32-1024).

For the asynchronous tests, 1MB and 10MB of data were transmitted, each broken up into the frame sizes specified above. For isochronous tests, data sizes of 1MB, 10MB and 40MB were used. Larger data sizes could not be used because the Firesypy 400 bus analyser, used for the tests, could not capture all generated packet data if the transmitted data size was greater than 40MB.

6.1.3.4 The new Raw Isochronous API

A new isochronous API, called *rawiso* was released at the time of carrying out these tests by the Linux IEEE1394 development community. Isochronous tests were also conducted using this new *rawiso* API. The drivers used for testing the *rawiso* API were the latest IEEE1394 Linux drivers from CVS as of (19/11/2002) and *libraw1394* version 5.1.0, which contains the user space *rawiso* API. During the tests only a maximum frame size of 512 bytes (500 bytes data) was used, since the *rawiso* API did not operate as expected using larger frame sizes.

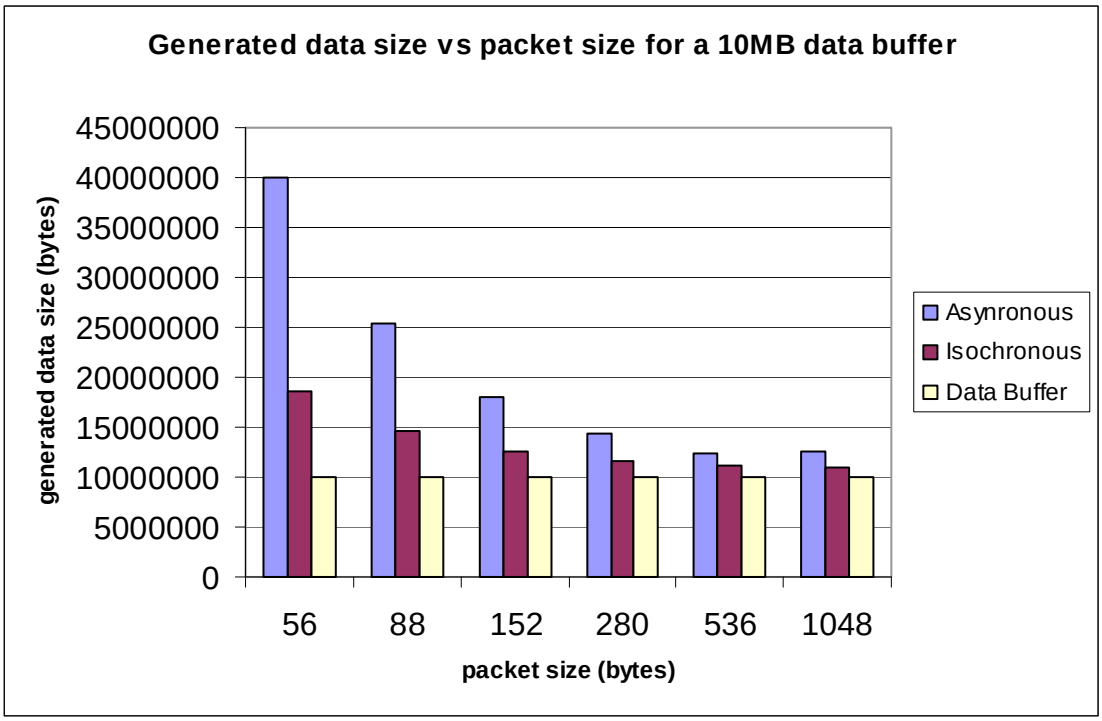
6.2 Tests and Results

Each test consists of a data buffer of a specified length, which is transmitted by an IEEE1394 node on a channel in case of isochronous tests, and using block writes targeted at a CSR register in the case of asynchronous tests. The data buffer is broken into a specified packet size, transmitted, and the time taken for the transmission to complete recorded from the FireSpy 400 bus analyser, which shows the amount of data transmitted and the time taken for the transmission. The data transmission rate is obtained by using the following formula

```
(data_transmitted(bytes)/time(secs))/1000000 -> transmission_rate (MB/sec)
```

6.2.1 Packet Size verses Data size

Before looking at the driver performance times, it's interesting to look at the packet sizes used compared to the overall data transmitted on the bus for both asynchronous and isochronous tests. Graph 6.2.1 shows this relationship.



Graph 6.2.1 A plot of actual data generated on the bus for a 10MB buffer vs. packet size.

For smaller packet sizes, it can be seen that the data transmitted on the bus is nearly twice as much as the actual data (yellow bar) transmitted by applications for isochronous transactions, and four times as much for asynchronous transactions.

The asynchronous data size includes acknowledgement packets, each 16 bytes in size, in addition to packet header and CRC data. For example, using packet sizes of 44 bytes, transmitting 10MB of data on the bus in isochronous mode will generate approximately 18MB of data on the bus, the 8MB of extra data coming from the packet header and CRC data. In asynchronous mode, transmitting 10MB of data using packet sizes of 56 bytes generates 40MB of data, the extra 30MB coming from packet header and CRC data, as

well as acknowledgment packets. As the packet size is increased, the data generated on the bus gets closer to the actual data transmitted from an application, but the two will never be equal since there will always be extra data in form of header data and acknowledgement packets.

6.2.2 Isochronous transmission tests

The tests measure how fast (in MB/sec) each client driver, using the IEEE1394 stack of the operating system under which the driver is installed, can successfully transmit data in isochronous mode. Data sizes of 1MB, 10MB and 40MB were transmitted using the Linux *raw1394* driver, the Linux *raw1394-2* driver and the Windows *raw1394* driver. The times taken for the transmissions to complete were then recorded. The expected theoretical data transfer rate for these tests is 40MB/sec. This is 80% of the 50MB/sec total data transfer rate defined by the IEEE1394 specification for the IEEE1394 bus, since up to 80% of bus bandwidth may be reserved for isochronous transmissions.

6.2.2.1 Isochronous transmission of a 1MB data buffer

Table 6.2.2.1 shows the results obtained and graph 6.2.2.1 shows a plot of packet size versus generated data transfer rate for the three drivers. The values in brackets indicate the data payload size, whereas the full packet size (the value not in brackets) includes header and CRC information.

Isochronous transmission tests for a 1MB buffer

Linux raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	1866664	3.969362854	0.470267917
76(64)	1466688	1.973743958	0.743099425
140(128)	1266748	0.995121928	1.272957579
268(256)	1166592	0.498498454	2.340211872
524(512)	1116592	0.247374227	4.513776611
1036(1024)	1092156	0.251124337	4.349064742

Windows raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	1866664	3.906121236	0.477881737
76(64)	1466760	1.951123027	0.751751673
140(128)	1266748	0.976499023	1.297234273
268(256)	1166592	0.488124512	2.389947588
524(512)	1116592	0.234999736	4.751460657
1036(1024)	1092156	0.121999878	8.952107313

Linux raw1394-2

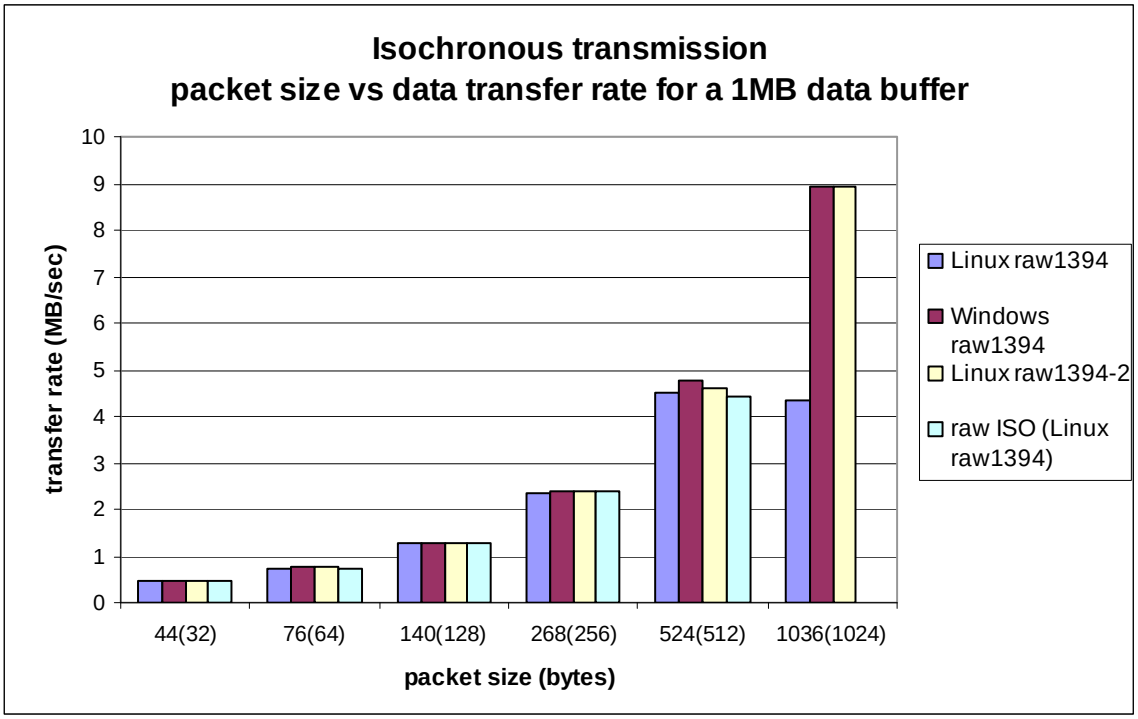
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	1866664	3.906123474	0.477881463
76(64)	1466664	1.952999247	0.75098032
140(128)	1266748	0.976499634	1.297233461
268(256)	1166592	0.488124817	2.389946094
524(512)	1116592	0.243999898	4.576198634
1036(1024)	1092156	0.121999959	8.952101369

Raw ISO (Linux raw1394)

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	1221544	2.581426961	0.473204944
76(64)	1352240	1.817536601	0.743996021
140(128)	1009764	0.786890849	1.283232613
268(256)	1403732	0.593761943	2.364132657
512(500)	1135752	0.255754964	4.440781841

Table 6.2.2.1 Isochronous transmission measurements for a 1MB buffer.

From graph 6.2.2.1, it can be seen that the data transfer rates for the three drivers are very close to each other for all packet sizes except for the Linux *raw1394* driver, whose transfer rate falls when a packet size of 1036 is used. Another observation that was made is that each time the packet size is doubled the transfer rate also doubles, ignoring the one case for the *raw1394* driver, i.e. the when the packet size was 1036.



Graph 6.2.2.1 Isochronous transmission of a 1MB data buffer using the three drivers.

6.2.2.2 Isochronous transmission of a 10MB data buffer

This test is the same as the one presented in section 6.2.2.1 except that the size of the data to be transmitted has been increased to 10MB. Problems were experienced in transmitting this amount of data with the Windows *raw1394* driver for smaller packet sizes. Each time a small packet size was used the system became unstable and had to be rebooted. The results are shown in Table 6.2.2.2 and Graph 6.2.2.2.

Isochronous transmission tests for a 10MB buffer

Linux raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	18666664	39.69800293	0.470216702
76(64)	14666664	19.84581497	0.739030573
140(128)	12666664	9.924594564	1.276290323
268(256)	11666816	4.954859843	2.35462079
524(512)	11156232	2.478992411	4.500309057
1036(1024)	10917084	2.481617432	4.399180897

Windows raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	unstable		
76(64)	unstable		
140(128)	unstable		
268(256)	unstable		
524(512)	11166524	2.441246969	4.574106652
1036(1024)	10917084	1.220623494	8.943858654

Linux raw1394-2

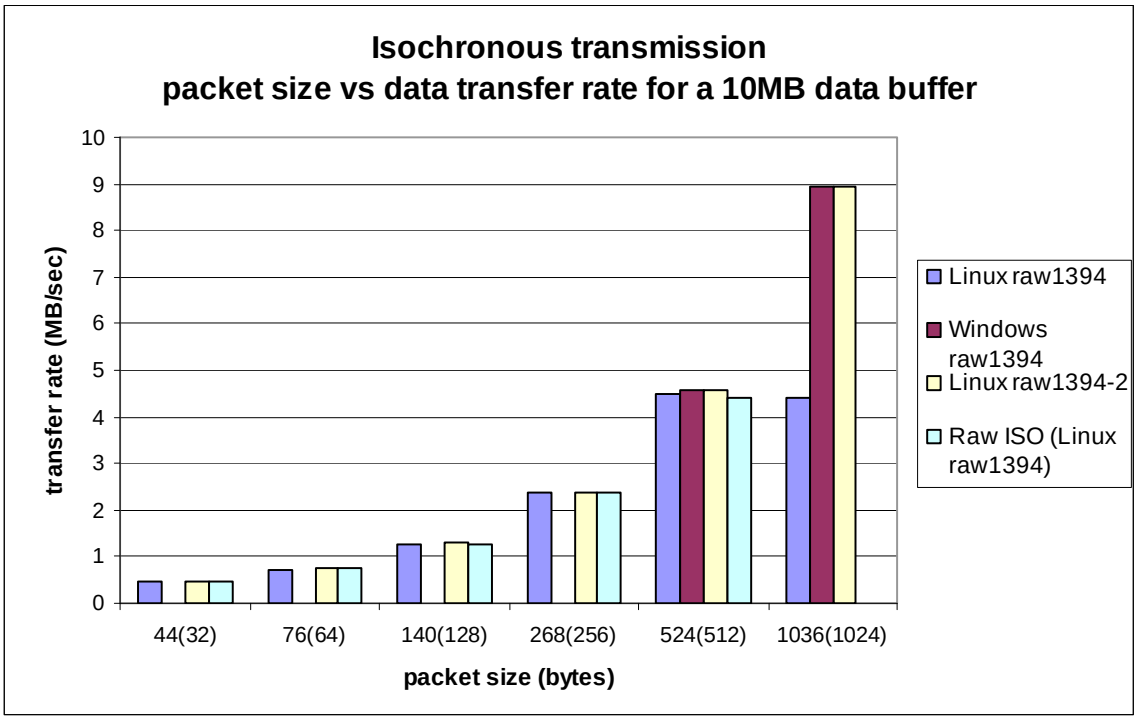
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	18666664	39.06236798	0.477868213
76(64)	14666664	19.53112111	0.750938152
140(128)	12666692	9.76549764	1.297086177
268(256)	11666816	4.882749959	2.389394521
524(512)	11166524	2.44125	4.574100973
1036(1024)	10917084	1.220625041	8.943847319

Raw ISO (Linux raw1394)

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	10254720	21.6141877	0.474443923
76(64)	10069828	13.51714868	0.744966874
140(128)	10627436	8.261167175	1.286432749
268(256)	11263316	4.761346293	2.365573791
512(500)	11367584	2.5684269	4.42589353

Table 6.2.2.2 Isochronous transmission of a 10MB data buffer using the three drivers.

Graph 6.2.2.2 shows that the performance of three drivers has stayed similar to the 1MB tests, *raw1394* still performing poorly when a packet size of 1024 is used.



Graph 6.2.2.2 Isochronous transmission of a 10MB data buffer using the three drivers.

6.2.2.3 Isochronous transmission of a 40MB data buffer

Table 6.2.2.3 and graph 6.2.2.3 show the results for this test, which is the same as the previous tests except for the larger data size (40MB). This time the Windows *raw1394* driver was unstable for all the packet sizes specified, and measurements for it could not be obtained.

Isochronous transmission tests for a 40MB buffer

Linux raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	67102868	142.7135073	0.470192831
76(64)	58666664	79.45402098	0.738372499
140(128)	50634240	39.72032109	1.274769151
268(256)	46666664	19.86253495	2.349481781
524(512)	44666664	9.928829732	4.498683652
1036(1024)	43667224	9.907579712	4.407456237

Windows raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
unstable			

Linux raw1394-2

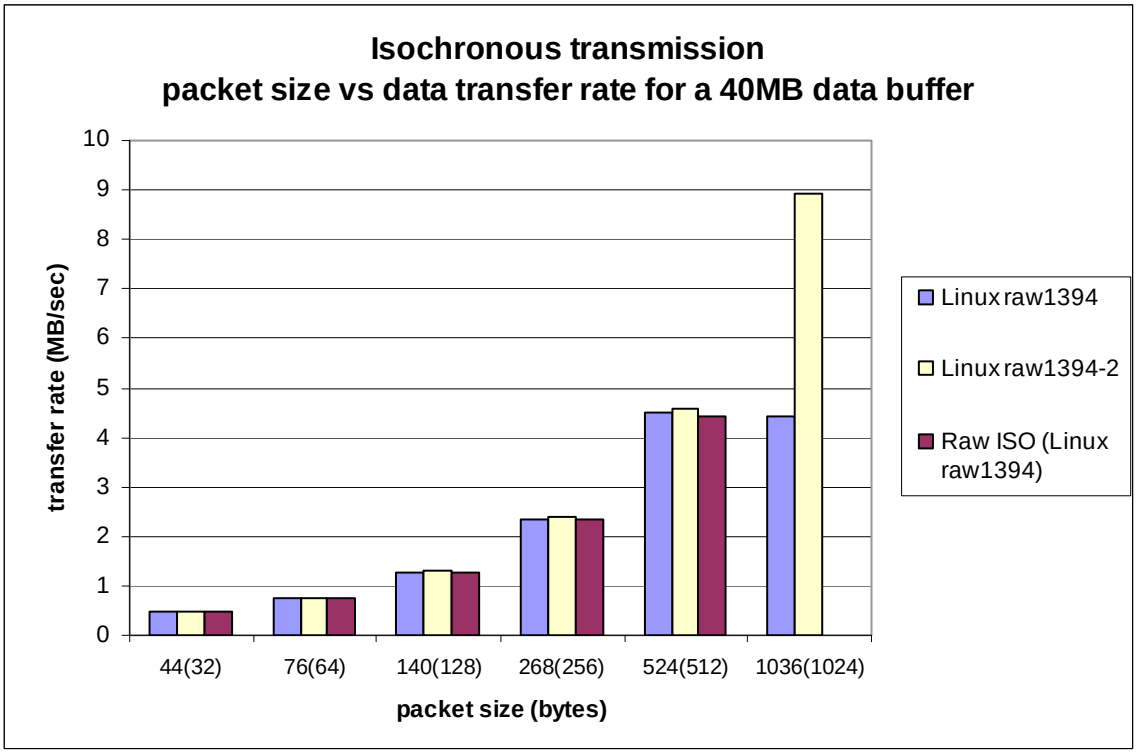
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	67103868	140.4236226	0.477867376
76(64)	58666664	78.12487063	0.750934543
140(128)	50666664	39.06236949	1.297070932
268(256)	46666664	19.53112156	2.389348909
524(512)	44666664	9.765497477	4.573926122
1036(1024)	43667224	4.882748678	8.943164369

Raw ISO (Linux raw1394)

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
44(32)	40135720	84.55058423	0.474694768
76(64)	40294816	54.02884098	0.745801969
140(128)	40382708	31.34475747	1.288340101
268(256)	39899260	16.86946523	2.365176338
512(500)	40261084	9.072432902	4.437738414

Table 6.2.2.3 Isochronous transmission of a 40MB data buffer using the three drivers.

From graph 6.2.2.3 it can be seen that the data transfer rates stay similar to those from the 1MB and 10MB tests even though the data buffer size is considerably larger, i.e. 40MB.



Graph 6.2.2.3 Isochronous transmission of a 40MB data buffer using the two drivers.

6.2.2.4 Buffered Isochronous transfers

This test differs from the previous three tests, in that this time the packet size is kept constant but the size of the buffers passed from the application to the kernel driver is varied. For example, in the previous three tests the application would pass down, to the kernel client driver, a 10MB data buffer not as a single chunk, but broken up (in user space) into a particular frame size e.g. 32 bytes. The total number of packets that can be constructed from a 10MB buffer is 312500 (10000000/32). The application would then have to make 312500 transmission requests, one for each packet, from user space when transmitting 10MB of data.

When the client driver supports buffering, the size of the buffer passed from an application to the kernel driver is greater than a single packet size. The buffer contains not one, but multiple packets. The contents of the buffer are broken up into individual packets in kernel space by the client driver, and transmitted one packet at a time. By using buffering, the number of transmission requests an application has to make to a

client driver is greatly reduced. For example, if the application wants to transmit 10MB of data in buffered mode, it specifies the buffer size it wants to use. If it specifies a buffer size of 1MB, then the data to be transmitted will be broken into 10 transmission requests each 1MB in size. With buffering enabled, only 10 transmission requests are made by the application, in contrast to the 312500 calls made in unbuffered mode.

When using buffering, the Windows *raw1394* driver, which was unstable for packet sizes ranging from 32 to 1024, functioned without problems when a packet size of 2048 was specified. The Linux *raw1394* driver does not support buffering, so measurements for it are not present in the results, but the new Linux *rawiso* API supports buffering, measurements for which are present. The Linux *raw1394-2* driver also supports buffering. The Windows *raw1394* driver is different from the Linux *raw1394-2* driver in that it only works as expected when smaller buffer sizes are specified. Buffer sizes in the measurements for the Windows driver range from 4k to 32k. A buffer size of 64k was tried, but caused the system to be unstable. The Linux *raw1394-2* driver on the other hand, can handle large buffer sizes. In the three tests conducted here, it was used to transmit 40MB of data using buffer sizes of 1MB, 10MB and 40MB. Buffering using the new *rawiso* API was tested using buffer sizes of 1MB and 10MB. The *rawiso* API did not support using a large buffer size of 40MB.

Since the Windows *raw1394* driver measurements were taken using a packet size of 2048 which the Linux host controller driver does not support (the maximum allowed isochronous packet size in the Linux driver is 1024, not including header and CRC data) similar measurements with the Linux driver were not made. Instead measurements to demonstrate the Linux *raw1394-2* driver's large buffer handling capabilities were taken.

Unbuffered data transfer measurements are included for comparison for each driver. The Windows unbuffered measurement for a 10MB data buffer is used here, because there were no results available for the 40MB data buffer tests (see table 6.2.2.3).
The results for the tests are shown in table 6.2.2.4, graphs 6.2.2.4a, 6.2.2.4b and 6.2.2.4c.

Buffered Isochronous transmission tests

Windows raw1394	**framesize(2048)**		
Buffer size (bytes)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
4096	43159484	2.440871784	17.68199554
8192	43159484	2.440871765	17.68199568
16384	43259484	2.440871704	17.72296509
32768	43141800	2.439871806	17.68199456

Unbuffered Windows raw1394

Packet Size (bytes)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
2048	10789868	0.610124247	17.68470611

Linux raw1394-2	**framesize(1024)**		
Buffer size (MB)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
1	43667112	4.884843271	8.939306663
10	43666704	4.882873149	8.942829901
40	43709356	4.887486247	8.943115907

Unbuffered Linux raw1394-2

Packet Size (bytes)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
1024	43667224	4.882748678	8.943164369

Unbuffered Linux raw1394

Packet Size (bytes)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
1024	43667224	9.907579712	4.407456237

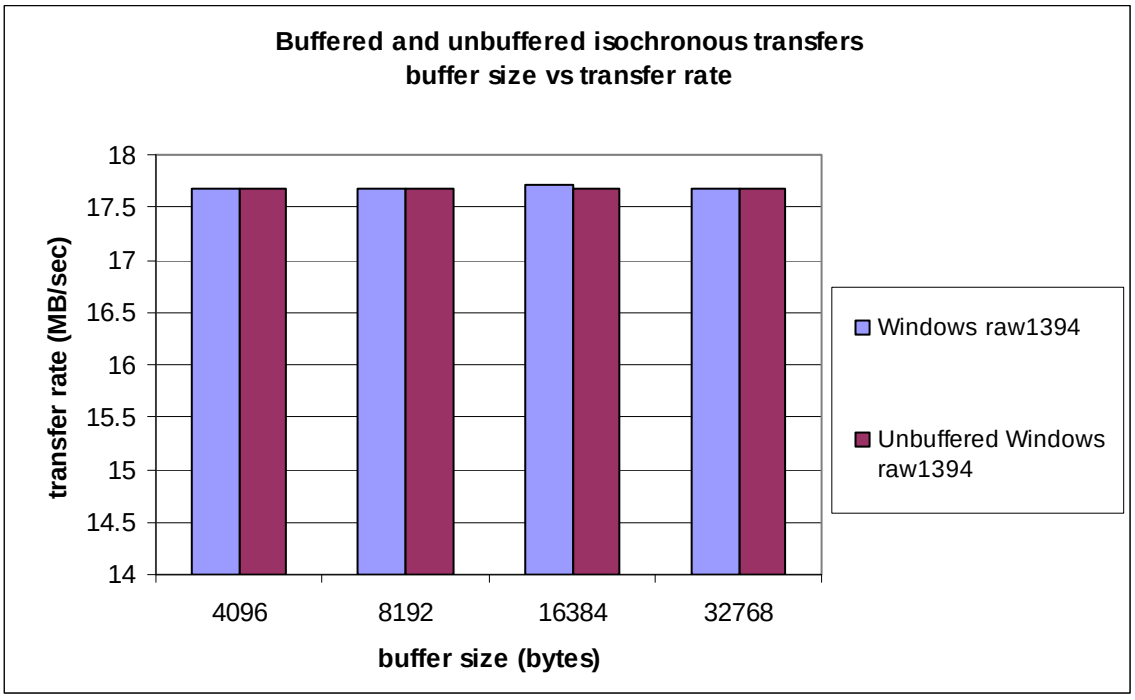
Raw ISO(Linux raw1394)	**framesize(500)**		
Buffer size (MB)	**Data size (bytes)**	**Time (secs)**	**Transfer rate (MB/sec)**
1	1372416	0.306755941	4.47396714
10	10997572	2.459422587	4.471607303
10	40386004	9.031924622	4.471472658

Unbuffered Raw ISO (Linux raw1394-2)

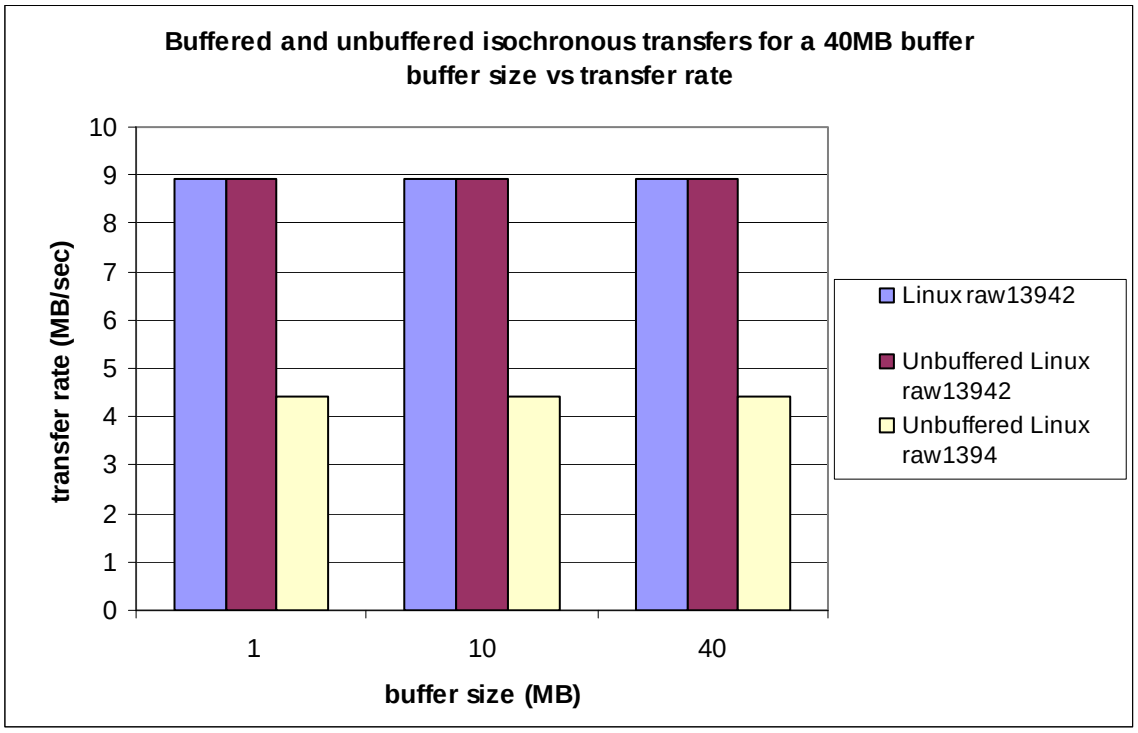
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
500	1135752	0.255754964	4.440781841
500	11367584	2.5684269	4.42589353
500	40261084	9.072432902	4.437738414

Table 6.2.2.4 Data transfer rates for buffered isochronous transfers of a 40MB data buffer. 1MB, 10MB and 40MB data buffer sizes for raw ISO.

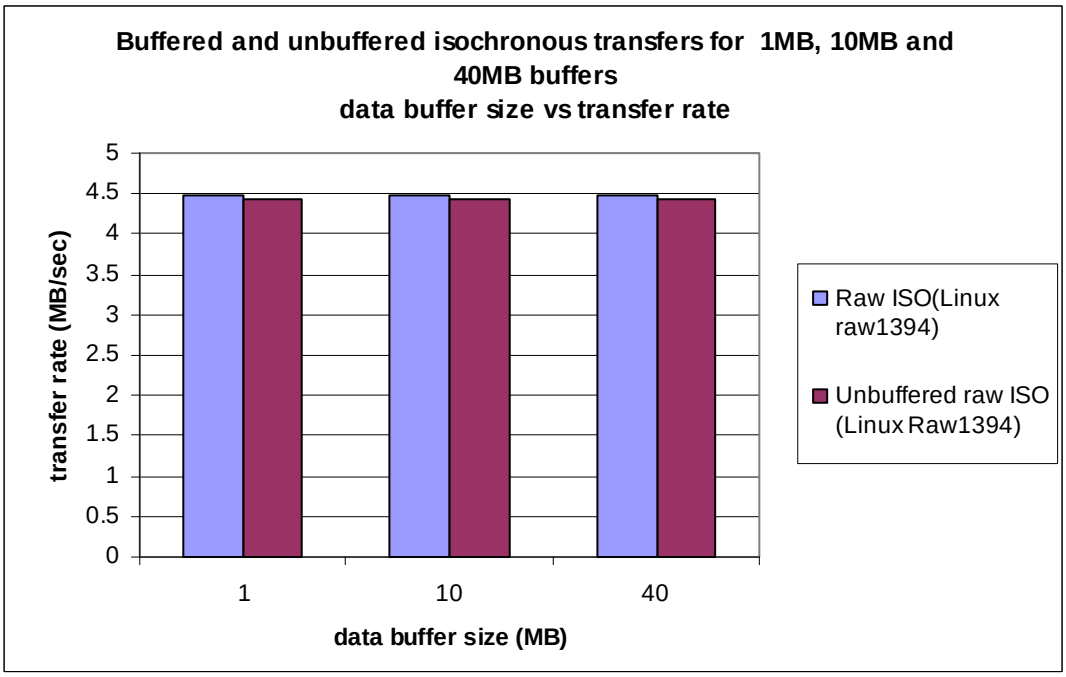
From graphs 6.2.2.4a, 6.2.2.4b and 6.2.2.4c it can be seen that the data transfer rates for buffered isochronous transfers differ very little from the unbuffered transfers. It must be noted that the Windows driver transferred data almost twice as fast as the Linux driver. This is expected because the packet size used in the Linux driver is 1024, whereas the packet size used in the Windows driver is 2048. The Linux *raw1394* driver's transfer rate is much slower than *raw1394-2*'s.



Graph 6.2.2.4a Plot of buffer size vs. transfer rate for a 40MB buffer using buffered isochronous transfers for the Windows *raw1394* driver



Graph 6.2.2.4b Plot of buffer size vs. transfer rate for a 40MB buffer using buffered isochronous transfers for the Linux *raw1394-2* driver



Graph 6.2.2.4c Plot of buffer size vs. transfer rate for 1MB, 10MB and 40MB data buffers using buffered isochronous transfers for the Linux *raw1394* driver

6.2.3 Asynchronous transmission tests

This section presents the results obtained from tests conducted in the same manner as the isochronous tests presented thus far. The tests measure how fast (in MB/sec) each client driver, using the IEEE1394 stack of the operating system under which the driver is installed, can successfully transmit data in asynchronous mode. Data buffers with sizes of 1MB and 10MB were transmitted using the Linux *raw1394* driver, the Linux *raw1394-2* driver, and the Windows *raw1394* driver. The expected theoretical data transfer rate for this test is 10MB/sec. This is 20% of the 50MB/sec total data transfer rate defined by the IEEE1394 specification for the IEEE1394 bus, since 20% of bus bandwidth is reserved for asynchronous transmissions. In the absence of isochronous traffic the asynchronous transfer rate can be greater than 10MB/sec [Anderson, 1999].

6.2.3.1 Asynchronous transmission of a 1MB data buffer

A 1MB data buffer was transmitted in asynchronous mode using packet sizes in the range 32 to 512. Transfers were made from one machine to another using a number of combinations of drivers. Table 6.2.3.1 and graph 6.2.3.1 show the various combinations of drivers used and the test results obtained for each combination.

Asynchronous transmission tests for a 1MB buffer

Linux raw1394 to Linux raw1394

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	5.153560628	0.776162403
88(64)	2533332	2.750780538	0.920950241
152(128)	1654960	1.288412496	1.284495459
280(256)	1433240	0.688358073	2.082114028
536(512)	1249920	0.365823303	3.416731492

Linux raw1394 to Windows raw1394

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	1.813305318	2.205916434
88(64)	2533332	0.955927124	2.65013089
152(128)	1800112	0.495435059	3.633396481

280(256)	1433240	0.251368225	5.701754866
536(512)	1249920	0.136363749	9.166072429

Linux raw1394-2 to Windows raw1394

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	1.81669106	2.201805298
88(64)	2533492	0.94546639	2.67962143
152(128)	1800112	0.499789632	3.601739381
280(256)	1433240	0.256748352	5.582275364
536(512)	1249920	0.15254482	8.193788553

Windows raw1394 to Windows raw1394

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	2.299773132	1.739301997
88(64)	2533332	1.186589213	2.134969686
152(128)	1800112	0.628371806	2.864724329
280(256)	1433240	0.384418556	3.72833199
536(512)	1249920	0.196399658	6.364165868

Windows raw1394 to Linux raw1394

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	2.13992393	1.869225323
88(64)	2533332	1.137934367	2.226254935
152(128)	1800112	0.60378597	2.981374344
280(256)	1433240	0.33087087	4.331720106
536(512)	1249920	0.184180115	6.786400367

Windows raw1394 to Linux raw1394-2

data size 1MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	2.028761169	1.97164657
88(64)	2533332	1.059637675	2.390753047
152(128)	1800112	0.591828817	3.041609243
280(256)	1433240	0.329343526	4.351808634

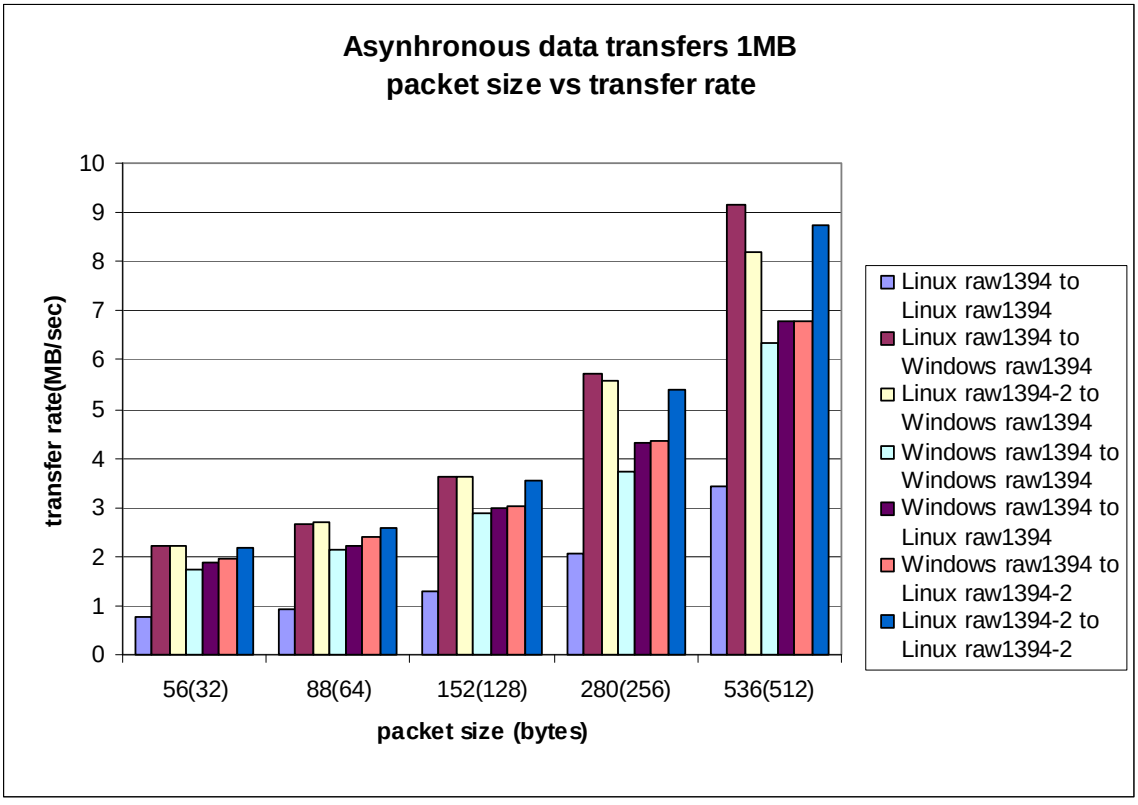
536(512)	1249920	0.183993245	6.793292873

Linux raw1394-2 to Linux raw1394-2

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	4000000	1.827820618	2.188398555
88(64)	2533492	0.977996867	2.590490916
152(128)	1800112	0.507454631	3.547335841
280(256)	1433240	0.26595872	5.388956602
536(512)	1249920	0.142984294	8.741659416

Table 6.2.3.1 Asynchronous data transfer rates for transmission of a 1MB buffer

From graph 6.2.3.1 it can be seen that the Linux *raw1394* to Windows *raw1394* transmission times are consistently higher than the rest. The Linux *raw1394-2* driver's performance is much better than the Linux *raw1394* driver's for Linux to Linux transmissions.



Graph 6.2.3.1 A plot of packet size vs. data transfer rate for asynchronous transfers of a 1MB buffer

6.2.3.2 Asynchronous transmission of a 10MB data buffer

Tables 6.2.3.2 and graph 6.2.3.2 present the results obtained when the data to be transmitted was increased to 10MB.

Asynchronous transmission tests for a 10MB buffer

Linux raw1394 to Linux raw1394

Data size 10MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	52.15602859	0.766929559
88(64)	25333332	27.72395323	0.91377055
152(128)	18000000	14.25100309	1.263068984
280(256)	14333516	6.957538167	2.060141915
536(512)	12499840	3.688368998	3.388988468

Linux raw1394 to Windows raw1394

data size 10MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	18.05831797	2.215045724
88(64)	25333332	9.323589559	2.717122181
152(128)	18000000	4.897288269	3.675503465
280(256)	14333148	2.499454447	5.734510592
536(512)	12499840	1.338277059	9.340248281

Linux raw1394-2 to Windows raw1394

data size 10MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	18.17033175	2.201390737
88(64)	25334304	9.359518433	2.706795673
152(128)	18000000	4.922309021	3.656820391
280(256)	14333516	2.512281759	5.705377571
536(512)	12499840	1.373709391	9.099333587

Windows raw1394 to Windows raw1394

data size 10MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	22.79233755	1.754975764

88(64)	25333332	11.79664634	2.147502881
152(128)	18000000	6.175462952	2.914761232
280(256)	14333516	3.409633809	4.203828564
536(512)	12499840	1.93437087	6.461966624

Windows raw1394 to Linux raw1394

data size 10MB

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	21.31536519	1.876580562
88(64)	25333332	11.22448997	2.256969543
152(128)	18000000	5.906849752	3.047309608
280(256)	14333516	3.256109192	4.402037879
536(512)	12499840	1.811194763	6.901433383

Windows raw1394 to Linux raw1394-2

data size 10MB

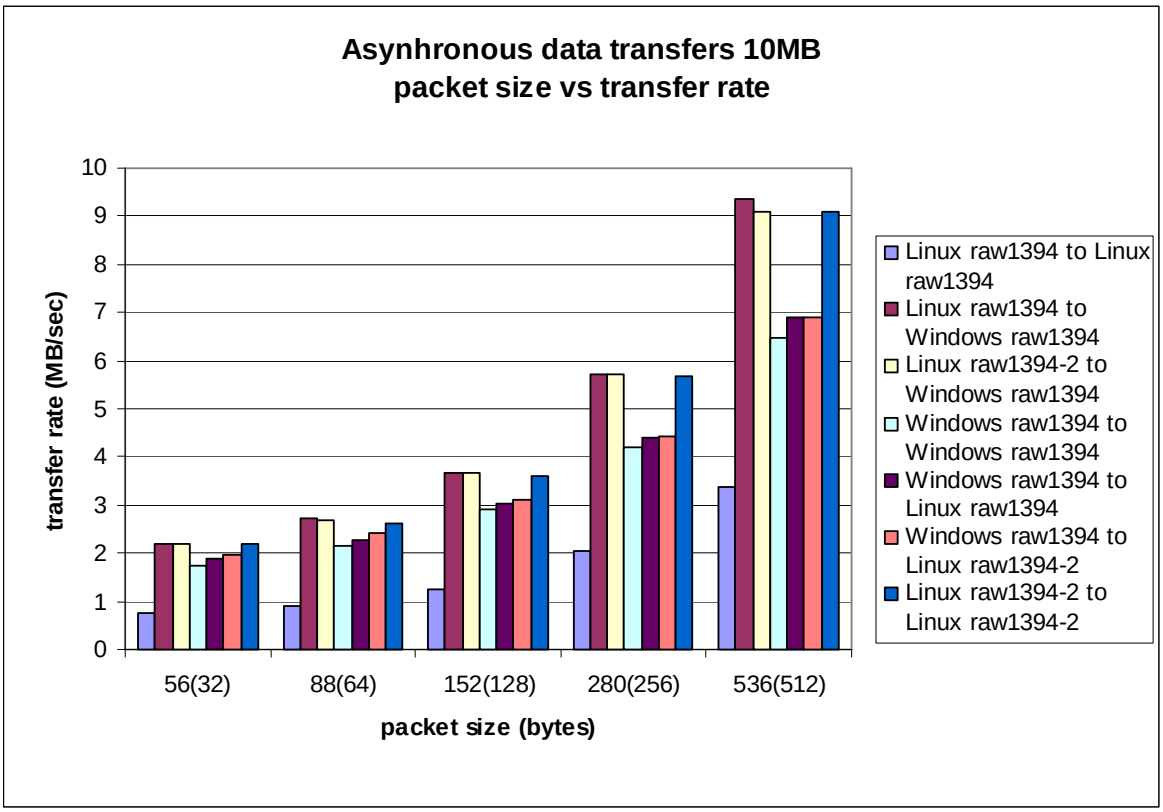
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	20.30745789	1.969719707
88(64)	25333332	10.46005343	2.421912295
152(128)	18000000	5.822876383	3.091255733
280(256)	14333516	3.24742454	4.413810336
536(512)	12499840	1.811110616	6.901754034

Linux raw1394-2 to Linux raw1394-2

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
56(32)	40000000	18.21714016	2.195734328
88(64)	25333332	9.633947083	2.629590113
152(128)	18000000	5.013583008	3.59024673
280(256)	14333516	2.521374878	5.684801624
536(512)	12499840	1.375271342	9.088999107

Table 6.2.3.2 Asynchronous data transfer rates for transmission of a 10MB buffer

From graph 6.2.3.2 it can be seen that the results for the 10MB tests are very similar to the 1MB data buffer test results.



Graph 6.2.3.2 A plot of packet size vs. data transfer rate for asynchronous transfers of a 10MB buffer

6.2.3.3 Buffered asynchronous transfers of 1MB and 10MB data buffers

Asynchronous transfers, in buffered mode, were conducted using a packet size of 1024 bytes with the Linux *raw1394-2* driver. The Linux *raw1394* and Windows *raw1394* drivers do not support buffering for asynchronous transmissions. Unbuffered mode measurements for the *raw1394* and *raw1394-2* drivers have been included for comparison. Since transmissions were made only from Linux, Windows unbuffered measurements are not included.

Buffered asynchronous transmission tests

Linux raw1394-2 to Windows raw1394 buffered

Frame size (1048)

Buffer size (MB)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
1	1158372	0.090947184	12.73675499
1	11583744	0.850747559	13.61595914
10	11583368	0.840971456	13.77379448

Linux raw1394-2 to Linux raw1394-2 buffered

Frame size (1048)

Buffer size (MB)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
1	1158372	0.093351095	12.40876714
1	11583744	0.849816427	13.63087795
10	11583368	0.847300517	13.67090869

Unbuffered Linux raw1394-2 to Windows raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
1024	11584964	0.841261454	13.77094356

Unbuffered Linux raw1394-2 to Linux raw1394-2

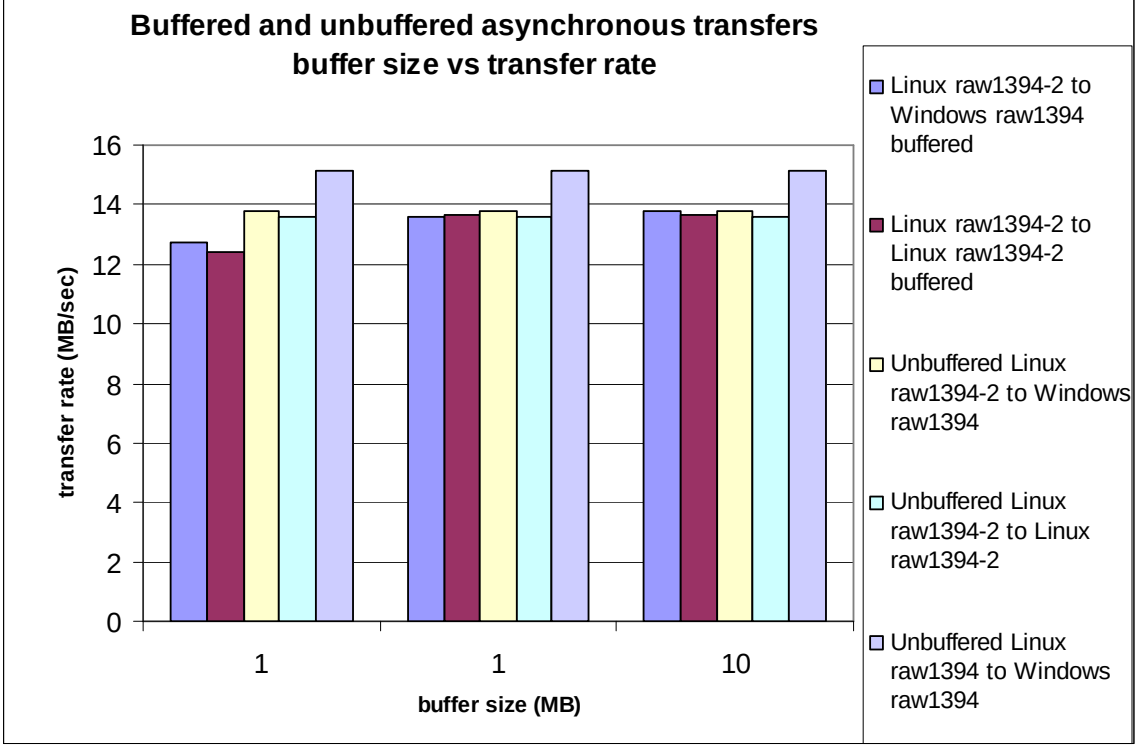
Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
1024	11584964	0.85070695	13.61804321

Unbuffered Linux raw1394 to Windows raw1394

Packet Size (bytes)	Data size (bytes)	Time (secs)	Transfer rate (MB/sec)
1024	11584964	0.763982686	15.16390909

Table 6.2.3.3 Measurements for buffered asynchronous data transfers

In graph 6.2.3.3, the first set of bars shows the transfer rates for 1MB of data using a 1MB buffer. The second set of bars shows the transfer rates for 10MB of data using a 1MB buffer and the third set of bars shows the transfer rates for 10MB of data using a 10MB buffer. From the graph it can be seen that Linux *raw1394-2* driver's buffering has not improved its performance significantly. The unbuffered and buffered transfers made with this driver have nearly the same transfer rates for this set of measurements. Buffering helps reduce the number of transitions made from user space to kernel space and back again during a data transfer session. From the results it seems that the kernel space to user space transition time does not significantly slow down the data transmission rate.



Graph 6.2.3.3 A plot of buffer size vs. transfer rate for buffered asynchronous transfers.

6.3 IEEE1394 Driver Performance Summary

From the results presented in the preceding sections it can be seen that the data transfer rate is affected by the packet size used during data transmission. Each time the packet size was doubled the transfer rate doubled as well.

6.3.1 Isochronous Performance

In isochronous mode, a maximum transfer rate of 8.9 MB per second was observed when using a packet size of 1024 bytes for both the Windows *raw1394* and Linux *raw1394-2* drivers. The Linux *raw1394* driver was much slower. A maximum transfer rate of 4.5 MB per second was observed for it using a packet size of 512 bytes. When buffering was used the maximum transfer rate for the Linux *raw1394-2* driver stayed the same, as in unbuffered transfers, at 8.9 MB per second using a packet size of 1024. The maximum transfer rate for the Windows driver doubled to 17.6 MB per second when using a packet size of 2048. When using buffering, the maximum transfer rate did not change much and increased to 17.7 MB per second. A packet size of greater than 1024 could not be used

with the Linux driver stack, because it does not support packets with a data payload greater than 1024 bytes. As can be seen from the test results, the performance of the new *rawiso* implementation did not differ very much from the old *raw1394* isochronous implementation. Although the highest data transfer rate obtained (17.7 MB/sec) is sufficiently high enough for most video and audio applications, it is not close to the expected theoretical transfer rate of 40MB/sec.

6.3.2 Asynchronous Performance

In asynchronous mode, a maximum transfer rate of 9.3 MB per second was observed while transferring data from the Linux *raw1394* driver to the Windows *raw1394* driver using a frame size of 512 bytes. When performing buffered transfers, a maximum transfer rate of 13.7 MB per second was observed while transmitting data from the Linux *raw1394-2* driver to the Windows *raw1394* driver using a packet size of 1024 bytes. This was a lower value than the 15 MB per second transfer rate obtained when performing data transmission from the Linux *raw1394* driver to the Windows *raw1394* driver in unbuffered mode using a packet size of 1024 bytes.

6.3.3 Overall performance

For isochronous transfers, the Linux *raw1394-2* driver consistently performed better than the Windows and Linux *raw1394* drivers when stability as well as data transfer rates are considered. For asynchronous transfers, data transfer rates were consistently high for data transmissions performed from the both the Linux *raw1394* and raw1394-2 drivers to the Windows raw1394 driver. Transmissions made from the Windows driver were slower than those from the Linux drivers except for Linux *raw1394* to Linux *raw1394* transfer rates which were consistently low.

The Linux *raw1394* and *raw1394-2* client drivers both use the same Linux *ieee1394_core* and host controller drivers, i.e. the same IEEE1394 stack, but their performance is different because they are implemented differently. The results presented thus far help determine whether the data transfer rate offered by the publicly available *raw1394* driver is sufficient for an IEEE1394 application of interest. If not, whether it is possible to construct a new client driver that performs better than *raw1394*, which *raw1394-2* does.

The results can also be used to help determine the operating system on which the IEEE1394 application of interest is to be developed based on the data transfer rate the operating system's IEEE1394 stack offers.

6.4 Summary

This chapter presented the tests conducted to determine the performance of the IEEE1394 driver stacks from the two operating systems and the results that were obtained. Data transmission times for selected data sizes, as well as graphical comparisons of the data transmission rates obtained using the existing and newly constructed client drivers from the two operating systems were presented.

7 Conclusion

Windows and Linux are two of the most popular operating systems in use today. Windows has the biggest market share, and Linux is gaining in popularity. Every new device that gets released to the public by a hardware manufacturer will almost certainly come equipped with a device driver that will make it operate on the Windows operating system. The two operating systems' driver architectures are different in many ways but have some similarities. This chapter presents concluding remarks for the various issues raised in the chapters presented thus far.

7.1 Device Driver Architectures

Comparison of the driver architectures used by the two operating systems has shown that the Windows operating system has a more developed architecture than Linux. The Windows driver architecture uses a formally defined driver model, which driver developers are encouraged to follow. Although driver writers can ignore the Windows driver model and construct their own monolithic drivers, it was found that most driver writers did not take this route. No formally defined driver model exists for the Linux operating system. Linux driver writers produce drivers based on their own personal designs. Unless two groups of driver developers cooperate and produce drivers that work together, drivers from different developers cannot operate together under the Linux operating system. Under Windows, drivers from two or more sets of developers can be made to work together, provided the developers have followed the Windows Driver Model (WDM) to construct their drivers. The Windows driver architecture supports PnP (Plug and Play) and Power management, by dispatching messages at appropriate times to device drivers which have been implemented to handle these messages. No such facility is offered by the current Linux driver architecture.

7.1.1 IEEE1394 driver stacks

The Linux IEEE1394 driver stack was found to be very similar to the Windows IEEE1394 driver stack with both sets of driver stacks using familiar layers to break up IEEE1394 functionality. In each stack there is a host controller layer that is IEEE1394

hardware specific, a functional driver layer that handles IEEE1394 I/O operations and a client driver layer with which applications interact.

7.2 Designing device drivers

When designing device drivers the facilities offered by an operating system should be evaluated. The Windows and Linux operating systems are both modern operating systems. They make available implementations for data structures such as stacks, queues and spin locks, as well as HAL (Hardware Abstraction Layer) routines for performing hardware independent operations. This enables device drivers to operate on different architectures such as IA64 (Intel's 64 bit platform) and SPARC. Driver functionality on both operating systems can be broken up into modules, which can be stacked together and that communicate using a standardised data structure. Under Windows this standardised data structure is the IRP (I/O Request Packet) and under Linux it can be any driver-defined structure since no standardised structure exists on that operating system.

7.2.1 Designing IEEE1394 Client Drivers

IEEE1394 client drivers should enable applications to perform all IEEE1394 operations and not a subset of them, unless the client driver is a highly specialised driver, for example, a driver for an IEEE1394 audio player that only performs audio transmission on an isochronous channel and nothing else. The Windows *1394bus* driver for example, changes the destination bus ID of asynchronous packets to be that of the local bus because its designers did not anticipate that their drivers might be used in a multi-bus environment. The Linux driver on the other hand, does not intercept and modify application specified information. A driver that enables applications to perform all possible IEEE1394 operations will make it possible to construct "applications" that control different IEEE1394 devices, instead of having to construct a separate driver for each IEEE1394 device. Application development is a much simpler process than driver development.

7.3 Implementing Device Drivers

Device drivers on both operating systems are made up of a set of routines that each operating system expects all drivers to implement. They include routines for standard I/O

such as reading from and writing to a device, and for sending device I/O control commands to a device. Every driver for each operating system implements a routine that will be executed when the driver is loaded for the first time, and a routine that gets executed when a driver is unloaded. It is possible to construct a driver for each operating system that uses identical naming for the various driver routines, although the usual approach is to use conventional names for each operating system. The device driver naming scheme on Windows (using device interfaces) is a lot more flexible than the current device driver naming scheme used by Linux. Driver naming clashes are more likely to occur in Linux as compared to Windows, which uses a GUID (Globally Unique Identifier) for each device.

7.3.1 Implementing IEEE1394 client drivers

Both the Windows and Linux operating systems provide implementations for IEEE1394 host controller and IEEE1394 bus drivers. It is not necessary for drivers from manufacturers of new devices to re-implement all IEEE1394 functionality. Instead client drivers should be constructed that utilise the already existing host controller and bus drivers, except where the required IEEE1394 functionality is not present in the bus and host controller drivers e.g. IEEE1394 bridging support.

7.3.1.1 IEEE1394 bridging support

It was found that both the Windows and Linux IEEE1394 stacks do not currently support IEEE1394.1 bridging. This was expected since the IEEE1394.1 standard is currently in draft phase. The Linux IEEE1394 stack could easily be modified to support IEEE1394.1 bridging, whereas the Windows IEEE1394 stack could not, since the source code for the Windows IEEE1394 drivers are proprietary to Microsoft.

7.3.2 Driver Development Environments

The Windows operating system provides a DDK (Device Driver Developer's Kit), which contains relevant documentation and development tools that help decrease the time required in learning to create new drivers. The Linux operating system does not provide a DDK, therefore initially some time will have to be spent by device driver developers to gather other sources to aid in the driver development process. Once time has been spent

in getting familiar with the two driver development environments, developers will find it easier to create Linux drivers than Windows drivers, because all of the Linux kernel source code is available to them. This enables driver developers to trace problems in their drivers by having a closer look at the kernel code that their drivers rely on. Under Windows only binary debug builds of the operating system's components that contain debug symbols such as function names and variable names are available, which is not as useful as having the operating system's source code.

7.4 IEEE1394 Driver Performance

The tests conducted using the *raw1394* driver implementations on both operating systems showed the performance of the host controller and bus drivers as well as the client drivers of each operating system. From the tests, it was found that the performance of the Linux IEEE1394 stack is on par with the Windows IEEE1394 stack. The Linux IEEE1394 stack was found to be the more reliable IEEE1394 stack, as it always behaved as expected. The Windows IEEE1394 stack produced the highest data transfer rates in the tests, because it could support a larger packet size than the Linux IEEE1394 stack.

7.5 Device Driver Licensing

Linux uses an open source scheme, whereby developers of device drivers usually release the source code for their drivers, but are not required to do so. In most cases, device drivers are created and maintained by groups of private individuals spread around the world. Most of the device drivers produced for Windows are released in binary form under a private licensing scheme, suited to the device's vendor. Hardware vendors may release device drivers for the Windows operating system under an open source license, but this is not the usual practice. Licensing schemes such as the GNU Public License (GPL) require source code for any software that uses that License to be released to clients. It was found that Linux's licensing scheme, which uses the GNU license, has an advantage over the Windows proprietary scheme, as it allows end users of device drivers to modify driver source code and tailor it to their needs.

7.6 Final Remarks

Drivers should be designed so that use of them requires very little interaction from end users, and all of a driver's functionality is made available to applications. The former is one of the strong points of Windows, which fully supports PnP. IEEE1394 devices, for example, are inherently PnP aware. The *raw1394* drivers presented in this work can be used to demonstrate this. A single *raw1394* driver, which gets automatically loaded when an IEEE1394 device is present in the system, can be used to control any IEEE1394 protocol compliant device.

From the examination of the driver architectures in use by the Linux and Windows operating systems, it was found that the Windows operating system has the more mature driver architecture as compared to Linux. The Windows driver architecture is standardised, well documented, has a device driver development kit, and several example drivers from which new drivers can be created. The performance of the Linux IEEE1394 drivers was found to be as efficient as the Windows IEEE1394 drivers. The Linux IEEE1394 driver stack was found to be more reliable than the Windows IEEE1394 stack in the tests, but the highest data transfer rates were recorded while using the Windows IEEE1394 stack, due to its ability to support a larger packet size than the Linux IEEE1394 stack.

Windows is a commercial, proprietary operating system and Linux is a free, open source operating system. If a device driver manufacturer wants to reach a large customer base, it has to supply a Windows device driver for its device. Although use of the Linux operating system is growing, it is not anticipated that Linux will be as popular as the Windows operating system in the near future, except in environments such as professional studios and academic institutions.

8 References

1394ta — *The 1394 trade association*, http://www.1394ta.org, 2002.

ACPI — *Advanced Configuration & Power Interface*, http://www.acpi.info/index.html, 2002.

Anadtech — *Anadtech*,http://www.anandtech.com, 2002.

Anderson D — *Firewire System Architecture*, Second Edition, Mindshare, 1999.

APM — *Advanced Power Management v.1.2*, http://www.microsoft.com/hwdev/archive/BUSBIOS/amp_12.asp, 2002.

Asche R — *MSDN:Writing Windows NT Kernel-Mode Drivers in C++*, http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnardevice/html/msdn_ntcpp.asp, *1995*.

Beck, Bohme, Dziadzka, Kunitz, Magnus, Verworner — *Linux Kernel Internals*, Addison Wesley, 1998.

Blanchard E — *Introduction to IRQs, DMAs and Base Addresses*, http://www.linuxgazette.com/issue38/blanchard.html, 1999.

Cant C — *Writing Windows WDM Device Drivers*, CMP Books, 1999.

Compuware — *NuMega DriverStudio Version 2.5*, http://www.compuware.com, 2001.

dapdesign *FireSpy 400 bus analyser*, http://www.dapdesign.com,2002.

Davis W *Operating Systems*, 2nd Edition, Addison Wesley, 1983.

Deitel HM *Operating Systems*, 2nd Edition, Addison Wesley, 1990.

Dhawan S *Networking Device Drivers*, John Wiley & Sons, 1995.

Fliegl D *Programming Guide for Linux USB Device Drivers*, http://usb.cs.tum.edu/usbdoc/, 2000.

Flynn IM, McHoes AM *Understanding Operating Systems*, Brooks/Cole, 1991.

Gilbert D *The Linux SCSI Generic (sg) HOWTO*, http://www.linux.org/docs/ldp/howto/SCSI-Generic-HOWTO/index.html, 2002.

Gooch R *The Linux-kernel mailing list FAQ*, http://www.tux.org/lkml/#s1-4, 2002.

GPL *GNU General Public License*, *http*://www.gnu.org/copyleft/gpl.html, 2002.

IEEE1212 *P1212 draft standard for a Control and Status Registers (CSR) Architecture for microcomputer buses*, IEEE, 1999.

IEEE1394a *P1394a draft standard for a High Performance Serial Bus*, IEEE, 1998.

IEEE1394.1 *P1394.1 Draft Standard for High Performance Serial Bus Bridges*, IEEE, 2002.

Jay P — *Ori S, Pomerantz O, The Linux Kernel Module Programming Guide*, http://www.tldp.org/LDP/lkmpg/, 2001.

KDB — *The Linux Built in Kernel Debugger*, http://oss.sgi.com/projects/kdb, 2002.

Kulkarni A — *Writing a Linux device driver*, http://www.freeos.com/articles/2677/2/13/, 2000.

Lawyer D S — *Plug and Play HOWTO*, http://www.tldp.org/HOWTO/Plug-and-Play-HOWTO-1.html, 2001.

Linux 1394 — *IEEE 1394 for Linux*, http://linux1394.sourceforge.net, 2002.

Linus FAQ — *The Rampantly Unofficial Linus Torvalds FAQ*, http://www.tuxedo.org/~esr/faqs/linus/index.html,2002.

Linuxhq — The Linux Headquarters, http://www.linuxhq.com, 2002.

Lohmeyer J — *SCSI Architecture Road Map*, http://www.t10.org/scsi-3.htm, 2002.

Lozano J — *The Windows 2000 Device Driver Book: A Guide for Programmers*, 2nd edition, Prentice Hall PTR, 2002.

Massie P — *Operating systems theory and practice*, Macmillan, 1986.

Microsoft 1394 — *IEEE 1394 Technology*, http://www.microsoft.com/hwdev/bus/1394/default.asp, 2002.

Microsoft DDK — *DDK- Kernel Mode Driver Architecture*, Microsoft, 2002.

Microsoft Hwdev — *Microsoft Hardware Dev*, http://www.microsoft.com/whdc/hwdev, 2002.

Microsoft KB Q319197 — *Windows does not support 1394 bridging*, http://support.microsoft.com/default.aspx?scid=kb;en-us;Q319197, 2002.

Microsoft WDM — *Introduction to the Windows Driver Model*, http://www.microsoft.com/hwdev/driver/wdm, 2002.

NuMega — *Using Driver Works*, Version 2.5, Compuware, 2001.

Oney W — *Programming the Microsoft Windows Driver Model*, Microsoft, 1999.

Open Group — *Universal Unique Identifier*, http://www.opengroup.org/onlinepubs/9629399/apdxa.htm, 1997.

Pajari G — *Writing unix device drivers, 1st edition*, Addison Wesley Professional; 1991.

Rubini A, Corbet J — *Linux Device Drivers*, 2nd Edition, Oreilly, 2001.

Russinovich M — *SysInternals*, http://www.sysinternals.com, 2001.

Sandra — *System Analyser, Diagnostic and Reporting Assistant* , http://www.sisoftware.net/, 2002.

Schmid P — *The Mobile Storage Giant: A FireWire Hard Drive From Western Digital*, http://www.tomshardware.com/storage/20020426/index.html, 2002.

Statmarket *Statmarket.com*, http://statmarket.com,2002.

Sun Sun Microsystems Inc, Writing Device Drivers, iUniverse.com, 2002.

Tom's Hardware *Tom's Hardware Guide*, http://www.tomshardware.com, 2002.

Tsegaye MA *ieee1394diag*, http://ieee1394diag.sourceforge.net, 2002.

9 Appendix

This appendix presents the design and application interface of the three APIs developed during this investigation. The first API is *M1394*, a class that makes accessing the existing Linux *libraw1394* library more user friendly. The second API is *Raw1394*, a class that presents an API for the Windows *raw1394* driver created during this investigation. The third API is *raw1394-2*, a library that presents an interface to the Linux *raw1394-2* driver created during this investigation.

A1 M1394 API reference

M1394 is a wrapper class for the Linux *libraw1394* library. One of the attributes of this class is a *handle* used by the C *libraw1394* library. Methods of the *M1394* class are wrappers to all of the *libraw1394* library routines. Figure A1 shows how *M1394* interacts with *libraw1394*.

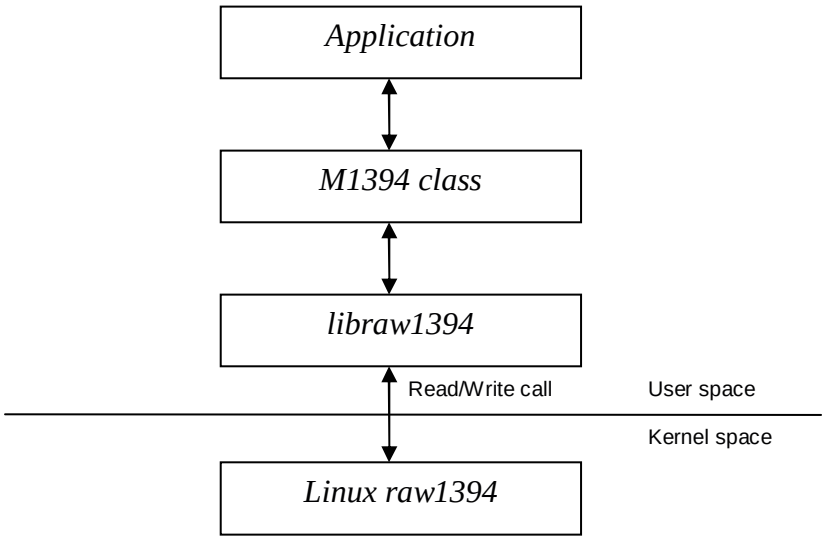


Figure A1 *M1394* class interaction with *libraw1394* and the kernel *raw1394* driver

When an *M1394* class is instantiated, it obtains a handle to the *raw1394* driver through *libraw1394* and selects the first IEEE 1394 card present on the host for use. Applications

can then make calls to its methods to make use of *libraw1394* functionality.

A1.1 M1394 class attributes

raw1394handle_t handle;	Handle to the raw1394 driver used by the libraw1394 library
struct raw1394_portinfo pinf[16];	Libraw1394 defined structure for obtaining IEEE 1394 card information from the Linux raw1394 driver.
int numcards;	The number of IEEE 1394 cards present in the system
struct pollfd pfd;	A structure used when polling the Linux raw1394 driver. Polling of the driver needs to be performed to if messages are to be passed to and from kernel space.
char * current_error_msg;	If some error occurs the class set this to a human understandable string.
int no_error;	Used to track the occurrence of error while making calls to Linux's libraw1394 library
int activeport;	The currently selected IEEE 1394 card. There can be more than one card available on the system.
void * userdata;	Users of the M1394 class can associate data with an instance of the class for later use. This member stores a reference to any user supplied data.
M1394_cmd_handler fcphandler; M1394_tag_handler taghandler; M1394_busreset_handler busreset_handler; M1394_iso_xmit_handler isoxmithandler[MAX_ISO_CHANNELS]; M1394_iso_recv_handler isorecvhandler[MAX_ISO_CHANNELS];	Application supplied callback routines which will be called by M1394 when the libraw1394 library passes kernel messages to M1394. The routines handle fcp, command completion(tag), bus reset, iso transmission/reception messages

A1.2 M1394 class methods

Each of the methods presented in this section make calls to their corresponding *libraw1394* function.

A1.2.1 Asynchronous operations

Each of the routines, shown in figure A1.2.1, allow performing the corresponding IEEE 1394 operation read, write and lock. They require a destination node id (16bit, first 10 bits bus id, next 6 node id, can be composed as (busid<<10| nodeid)), a 48 bit destination CSR address, the length of the data to be written to the CSR address and the buffer containing the data to be used for the operation. For lock operations additional arguments are necessary. *extcode* is the extended transaction code, which determines the lock operation, *arg* is the argument part of a lock operation and *result* is where the return value of the lock operation will be saved.

The sync ops are synchronous. The async ops are asynchronous, which means that a function call such as *async_read* will return before any data is read. The fourth argument

to an async_xxx operation is a user supplied pointer to data, which will be passed to the user on completion of the operation performed by the particular async_xxx call. Each routine returns 0 for success and -1 on failure.

```
/* sync ops*/
int sync_read(nodeid_t node, nodeaddr_t addr,
         size_t length, quadlet_t *buffer);
int sync_write(nodeid_t node, nodeaddr_t addr,
          size_t length, quadlet_t *data);
int sync_lock(nodeid_t node, nodeaddr_t addr,
         unsigned int extcode, quadlet_t data, quadlet_t arg,
         quadlet_t *result);
int sync_lock64(nodeid_t node, nodeaddr_t addr,
               unsigned int extcode, octlet_t data, octlet_t arg,
               octlet_t *result);


/* async ops */
int async_read(nodeid_t node, nodeaddr_t addr,
               size_t length, quadlet_t *buffer, unsigned long tag);
int async_write(nodeid_t node, nodeaddr_t addr,
                size_t length, quadlet_t *data, unsigned long tag);
int async_lock(nodeid_t node, nodeaddr_t addr,
               unsigned int extcode, quadlet_t data, quadlet_t arg,
               quadlet_t *result, unsigned long tag);
int async_lock64(nodeid_t node, nodeaddr_t addr,
               unsigned int extcode, octlet_t data, octlet_t arg,
               octlet_t *result, unsigned long tag);
```

Figure A1.2.1 Asynchronous routines supported by the *M1394* class

A1.2.2 Isochronous operations

The routines shown in figure A1.2.2 are used for performing isochronous operations with the M1394 class. *start_iso_rcv* begins isochronous reception on the specified channel and *stop_iso_rcv* halts it. *iso_xmit_init* initializes isochronous transmission and *iso_recv_init* initializes isochronous reception. *buf_packets* is the number of packets to be buffered, *max_packet_size* is the maximum size of isochronous packets that will be transmitted or received. *speed* is the bus speed to be used during the operation. *irq_interval* is the maximum latency when sending or receiving packets. Setting this value to -1 will let the driver calculate a default value.

iso_xmit_start commences isochronous transmission and *iso_recv_start* commences isochronous reception. *start_on_cycle* is the isochronous cycle number on which transmission or reception should start. *prebuffer_packets* specifies the number of packets to be queued before isochronous transmission occurs. *iso_shutdown* frees isochronous resources that are currently in use. *set_iso_xmit_handler* installs an application

implemented callback routine that M1394 will call to inform applications to fill an isochronous packet buffer before transmission. *set_iso_recv_handler* installs an application callback routine that will be called to receive incoming isochronous packets on a channel. Each routine returns 0 on success and -1 on failure.

```
int start_iso_rcv(unsigned int channel);
int stop_iso_rcv(unsigned int channel);

int iso_xmit_init(unsigned int buf_packets,
                  unsigned int max_packet_size,
                  int channel,
                  enum raw1394_iso_speed speed,
                  int irq_interval);

int iso_recv_init(unsigned int buf_packets,
                  unsigned int max_packet_size,
                  int channel,
                  int irq_interval);

int iso_xmit_start(int start_on_cycle, int prebuffer_packets);
int iso_recv_start(int start_on_cycle);
void iso_shutdown();

void set_iso_xmit_handler(M1394_iso_xmit_handler ixh);
void set_iso_recv_handler(int channel,M1394_iso_recv_handler irh);
```

Figure A1.2.2 Isochronous routines provided by the M1394 class

A1.2.3 Function Control Protocol (FCP) operations

The routines shown in figure A1.2.3 are used for performing FCP operations. The Linux libraw1394 library implements these operations only for the FCP register, rather than operations for any register. *start_fcp_listen* commences listening for incoming FCP data written to the local node's FCP register and *stop_fcp_listen* halts it. *set_fcp_handler* installs an application defined callback routine that will be called by *M1394* to handle incoming FCP data. *fcp_send_command* sends data to a specified node's FCP command register and *fcp_send_response* sends data to a specified node's FCP response register. Both of these routines require a destination node ID, bus Id, a buffer containing data and the size of the buffer.

```
int start_fcp_listen();
int stop_fcp_listen();
void set_fcp_handler(M1394_cmd_handler fcph);
int fcp_send_command(int nodeid, int busid, void * data, size_t size);
int fcp_send_response(int nodeid, int busid, void * data, size_t size);
```

Figure A1.2.3 FCP routines provided by the *M1394* class

A1.2.4 Miscellaneous operations

The following routines perform various operations that are not classified as either asynchronous or isochronous operations.

- Error checking: *isok* returns 1 if there is no error and 0 if there is an error as a result of a previous *M1394* operation. *get_current_error_msg* returns a human understandable error description.

```
int isOk();
const char * get_current_error_msg();
```

- Information: *get_generation* returns the current bus generation number. This value is updated every time a bus reset occurs. *get_port_count* returns the number of IEEE 1394 cards available on a host. *get_local_id returns* the local node ID. The first 10 bits specify the bus ID the next 6 the node ID. *get_node_count* returns the total number of nodes currently on the bus.

```
int get_generation();
int get_port_count();
void get_port_info(int portno, PortInfo & pinf)
nodeid_t get_local_id();
int get_node_count();
```

- Operations: *process_msg polls* the *raw1394* driver through *libraw1394*. This is required since polling is necessary for passing new requests to and from kernel space. *pollForLeftOverMessages* does the same thing as process_msg but blocks until all messages have been processed. *process_msg* only processes a single message. *set_active_port* selects the IEEE 1394 card to perform operations on. *set_tag_handler* installs an application defined callback routine that will be called when asynchronous routine calls complete. *generate_bus_reset* generates a soft bus reset. *set_userdata* associates a reference to a user defined data structure with the M1394 class. *get_userdata* returns the associated data or NULL if there is no associated data.

```
int process_msg();
void pollForLeftOverMessages();
void set_active_port(int portno);
void set_tag_handler(M1394_tag_handler tagh);
int generate_bus_reset();
void set_busreset_handler(M1394_busreset_handler bh)
void set_userdata(void *data);
void * get_userdata();
```

A2 The Windows Raw1394 API reference

The Windows *Raw1394* API exposes routines to applications that allows them to communicate with the Windows *raw1394* driver. The *Raw1394* API communicates with the *raw1394* driver using the Win32 *DeviceIoControl* function. This is because the windows *raw1394* driver was designed to be more flexible than the Linux *raw1394* driver. It allows the passing of multiple parameters along with the data and getting immediate feedback on the results of the operations i.e. when using *DeviceIoControl* the request will have input and output parameters. In contrast, the read operation can only read data from kernel space and does not have input parameters associated with it. Any buffer passed down to kernel space is wiped clean by the operating system before data from kernel space is copied to it. Figure A2 shows how the Windows *Raw1394 class* interacts with the Windows *raw1394* driver.

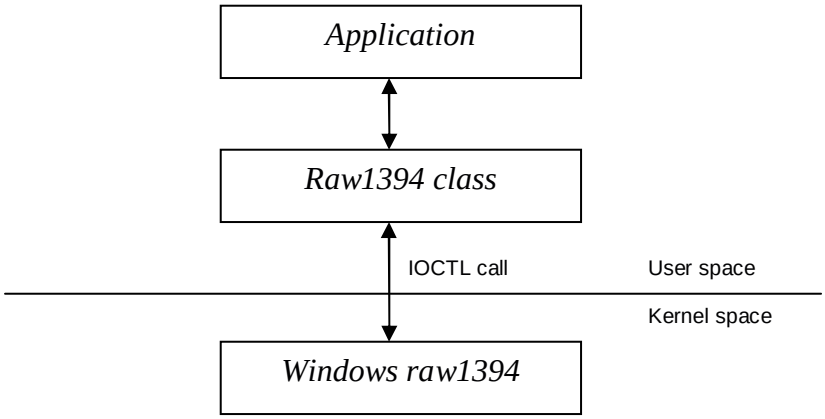


Figure A2 *Raw1394* class interaction with the windows *raw1394* driver

When necessary, the *Raw1394* API launches a second thread, which is used to listen for events from kernel space. Instead of polling, as done by the Linux *raw1394* driver and Linux *libraw1394* library, a handle to a kernel space event object is obtained by *Raw1394* from kernel space, which it uses to listen for events from the kernel space driver *raw1394*. In this way, the *Raw1394* API only goes down to kernel space to retrieve data as it becomes available. Applications do not have to perform any polling. A new thread is launched when operations such as asynchronous listens and isochronous listens are performed.

A2.1 Windows Raw1394 class attributes

HANDLE handle;	Handle representing an open handle
int error;	Indicates error status since the last operation performed with the API.
Thread* isocthread; Thread* busresetthread;	Thread used for listening for events: isochronous and bus resets
Vector * asyncThreads;	A list of threads used for listening for incoming asynchronous data written to any registered CSR address ranges. Unlike Linux's *raw1394,* our API handles asynchronous listen operations for any specified CSR address range not just the FCP address.
ISO_CALLBACK iso_callback; BUSRESET_CALLBACK busreset_callback;	Callback routines used to handle incoming events: isochronous data and bus resets.
Vector * asyncCallbacks;	A list of callback functions for each of the registered CSR address ranges. When data destined for a specific local CSR address comes in, this list is searched for a an application defined callback routine that will handle the incoming data.
PVOID userdata;	User defined data can be associated with the Raw1394 class. A reference to this data is stored here.

A2.2 Windows Raw1394 class methods

A2.2.1 Asynchronous operations

Figure A2.2.1 shows the asynchronous operations supported by the Raw1394 API. *async_read*, *async_write* and *async_lock* perform the corresponding IEEE 1394 asynchronous read, write and lock operations. Each routine accepts a node ID, bus ID, and destination CSR address as well as the data to be sent to the specified register. Lock operations require additional arguments. *lockOp* is the lock operation type. *arguments* is the argument value of a lock operation. *returned_values* is the result of the lock operation. *sizeOfarguments* and *sizeOfreturned* values specify the sizes of parameters specified earlier. These sizes are 4 bytes for a 32 bit lock operation and 8 bytes for a 64 bit lock operation. Each routine returns the size of the data actually written when the operation completes.

async_start_listen is used to notify the driver to start listening for asynchronous write transaction data written to a particular CSR address. The arguments to it are a CSR address, the size of the address range and a data buffer where received data will be stored. *async_stop_listen* tells the driver to stop listening for incoming data written to the

specified CSR address. *set_async_callback* is used to install an application defined callback routine that will be called by *Raw1394* whenever new data arrives.

```
size_t async_write(nodeid_t nodeid, nodeid_t busid,
        nodeaddr_t address,byte_t * data_buffer, size_t length);
size_t async_read(nodeid_t nodeid, nodeid_t busid,
        nodeaddr_t address,byte_t * data_buffer, size_t length);
size_t async_lock(nodeid_t nodeid, nodeid_t busid,
        nodeaddr_t address,byte_t * arguments, byte_t * datavalues,
        byte_t * returnedvalues,size_t sizeOfarguments,
        size_t sizeOfdatavalues, doublet_t lockOp);
void async_start_listen(octet_t address,size_t address_size,
        byte_t * buffer);
void async_stop_listen(octet_t address);
void set_async_callback(nodeaddr_t address, ASYNC_CALLBACK func);
```

Figure A2.2.1 Asynchronous operations implemented by the Windows *Raw1394* API

A2.2.2 Isochronous operations

Figure 2.2.2 shows the isochronous operations provided by the Windows Raw1394 API. *iso_start_listen* initializes isochronous listening for the specified channel. The speed of the link as well as the *tag* and *sy* fields of the expected isochronous packets need to be specified. *bytesPerFrame* specifies the size of the expected isochronous frame. *buffer* specifies a list of buffers where incoming isochronous data will be stored. *sizes* specifies the size of each buffer and finally, *buffer_count* specifies the number of supplied buffers. *iso_start_send* has the same parameters as *iso_start_listen*. It is used for transmitting data contained in the supplied buffer on the specified isochronous channel. *iso_stop_listen* halts isochronous reception and *iso_stop_send* halts isochronous transmission. *async_stream* is used to transmit asynchronous stream packets on the bus. The data buffer contains the packet data and the size of the transmitted frame, and is specified by *buffer_size*. *set_iso_callback* installs an application defined callback routine that will be called by the *Raw1394* API whenever new isochronous data arrives.

```
void iso_start_listen(doublet_t channel,
   byte_t speed,  byte_t tag,   byte_t sy, size_t bytesPerFrame,
        byte_t ** buffer, size_t *buffer_sizes,  doublet_t buffer_count);
void iso_start_send(doublet_t channel,
   byte_t speed, byte_t tag, byte_t sy, size_t bytesPerFrame,
        byte_t ** buffer, size_t *buffer_sizes,doublet_t buffer_count);
void iso_stop_listen(doublet_t channel);
```

```
void iso_stop_send(doublet_t channel);
size_t async_stream(doublet_t channel, byte_t speed,
        byte_t tag, byte_t sy,
        byte_t * data_buffer, size_t buffer_size);
void set_iso_callback(ISO_CALLBACK func);
```

Figure A2.2.2 Isochronous operations implemented by the Windows *Raw1394* API

A2.2.3 Miscellaneous operations

The following routines perform various operations that are not classified as either asynchronous or isochronous operations.

- Error checking: *isOk* indicates whether an error occurred during the last operation performed with the API. It returns 0 on failure and 1 on success.

```
int isOk();
```

- Information: *get_generation* returns the current bus generation number. *get_cycle_time* returns current cycle time data. *get_available_resources* is used to check if the requested resources are available for use before an isochronous transmission can be made. *get_local_nodeid* returns the local node/bus ID.

```
quadlet_t get_generation();
void get_cycle_time(doublet_t & clockticks,
        doublet_t & isocCyclesThisSecond, byte_t & secondCount);
void get_available_resources(byte_t speed,
        octet_t & channelsAvailable,  quadlet_t & bytesPerFrameAvailable);
void get_local_nodeid(nodeid_t & nodeid,nodeid_t & busid);
```

- Operations: *generate_busreset* generates a soft bus reset. *set_busreset_callback* installs an application defined callback routine that will be called by the *Raw1394* API when a bus reset occurs. s*etUserData* associates user defined data with an instance of a *Raw1394* class. *getUserData* returns this data or NULL if there is no previously associated data.

```
void generate_busreset();
void set_busreset_callback(BUSRESET_CALLBACK func);
void setUserData(PVOID data);
PVOID getUserData();
```

A3 Linux raw1394-2 API reference

The Linux *raw1394-2* API uses the same notation as the Linux *libraw1394* library. Every routine it implements is prefixed by *raw13942_*. Figure A3 shows the interaction of the Linux *raw1394-2* library with the Linux *raw1394-2* driver. Like the Windows *Raw1394* API, the *raw1394-2* API performs interactions with the *raw1394-2* driver using the Linux *ioctl* system call. The advantages of this are explained in section A2.

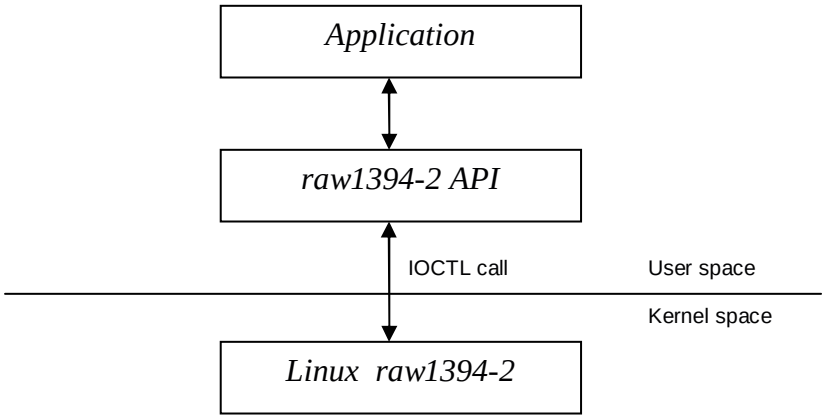


Figure A3 Linux *raw1394-2* API interaction with the Linux *raw1394-2* driver

A3.1 Linux libraw1394-2 implemented operations

A3.1.1 Asynchronous operations

Asynchronous operations supported by the Linux raw1394-2 API are shown in figure A3.1.1. *async_write*, *async_read* and *async_lock* perform the respective IEEE 1394 operations. Each routine requires the node ID, bus ID of the target node as well as the CSR register address where the data of the specified length will be written to. A handle to an open raw1394-2 device is also required. The *async_write* operation has an additional argument called *frame_size*, which means that multiple frames of that size can be placed in the data buffer and transmitted at once. Each of the routines returns the size of data actually operated on by that routine.

async_start_listen is used to start listening for data written from remote nodes to the specified CSR address range. *address* is the start address and *address+size* is the address range. *async_stop_listen* halts listening for remote data written to the specified CSR

address. *set_async_callback* is used by applications to install a callback routine that will be called by the *raw1394-2* library, whenever new asynchronous data is written to the local node.

```
size_t raw13942_async_read(raw13942_handle_t handle,
       nodeid_t nodeid, nodeid_t busid, nodeaddr_t address,
       byte_t * data, size_t length);
size_t raw13942_async_write(raw13942_handle_t handle,
       nodeid_t nodeid, nodeid_t busid,nodeaddr_t address,
       byte_t * data, size_t length, size_t frame_size);
size_t raw13942_async_lock(raw13942_handle_t handle,
       nodeid_t nodeid, nodeid_t busid, nodeaddr_t address,
      quadlet_t argument, quadlet_t * data, doublet_t lockOp);
void raw13942_async_start_listen(raw13942_handle_t handle,
       octet_t address,size_t address_size);
void raw13942_async_stop_listen(raw13942_handle_t handle, octet_t address);
void raw13942_set_async_callback(raw13942_handle_t handle, async_callback_t func);
```

Figure A3.1.1 isochronous operations supported by the *raw1394-2* API

A3.1.2 Isochronous operations

Isochronous operations supported by the raw1394-2 library are shown on figure A3.1.2. Each routine requires a handle to the raw1394-2 device. *iso_start_listen* commences isochronous reception and *iso_stop_listen* halts isochronous reception on the specified channel. *iso_start_listen* requires the desired transmission speed and the isochronous packet header fields, *sy* and *tag*. Isochronous transmission is performed using *iso_talk*, which has the same parameters as *iso_start_listen*. It returns the size of the data it transmitted. In addition it requires the data buffer to be transmitted and its length. The data buffer can contain multiple isochronous packets, each packet having the size specified by *frame_size*. *async_stream* is used to transmit isochronous stream packets and returns the size of the data it transmitted. *set_iso_callback* is used by applications to install a routine that will be called to notify them when new isochronous data becomes available.

```
void raw13942_iso_start_listen(raw13942_handle_t handle,
       doublet_t channel,byte_t speed, byte_t tag,byte_t sy);
void raw13942_iso_stop_listen(raw13942_handle_t handle, doublet_t channel);
size_t raw13942_iso_talk(raw13942_handle_t handle,
     doublet_t channel,doublet_t speed, doublet_t tag, doublet_t sy,
     byte_t * data, size_t length, size_t frame_size);
```

```
size_t raw13942_async_stream(raw13942_handle_t handle,
       doublet_t channel, byte_t speed,
      byte_t tag, byte_t sy, byte_t * data, size_t length);
void raw13942_set_iso_callback(raw13942_handle_t handle, iso_callback_t func);
```

Figure A3.1.2 Isochronous operations supported by the *raw1394-2* library

A3.1.3 Miscellaneous operations

- Error checking: *isok* is used to check whether an error occurred since the last operation performed with the *raw1394-2* library. The result is 0 for failure and 1 for success.

```
int raw13942_isok(raw13942_handle_t handle);
```

- Information: *get_generation* returns the current bus generation number. *get_local_id* returns the local node ID and bus ID. *get_port_count* returns the number of IEEE 1394 cards available on the host. *get_node_count* returns the number of nodes that are currently on the bus. *get_bus_info* returns IEEE 1394 bus information. The information is returned in a `BUS_INFO_1394` structure.

```
int raw13942_get_generation(raw13942_handle_t handle);
void raw13942_get_local_id(raw13942_handle_t handle, nodeid_t * nodeid,
       nodeid_t * busid);
int raw13942_get_port_count(raw13942_handle_t handle);
int raw13942_get_node_count(raw13942_handle_t handle);
void raw13942_get_bus_info(raw13942_handle_t handle, PBUS_INFO_1394 pbi);
typedef struct _BUS_INFO_1394
{      nodeid_t nodeid;  // local node ID
       int generation;   // currnt bus generation
       int node_count;   // total number of nodes on the bus
       nodeid_t irm;     // the current irm
       nodeid_t busmgr;  // the current bus manager
       octet_t channels_inuse;  // 64 bit bitmap of isochronous channels
}BUS_INFO_1394,*PBUS_INFO_1394;
```

- Event processing: Unlike the Windows *Raw1394* library, the Linux *raw1394-2* library is not multi-threaded. This functionality can be implemented at the application level when necessary using the routines presented in this section. The following routines wait for the particular event to occur and route data from that event to an application supplied callback routine. *async_wait_for_data* waits for data written to the specified CSR address. Similarly *iso_wait_for_data* waits for data arriving on the specified channel. These routines should be called repeatedly in a thread for even processing to occur.

```
void raw13942_wait_for_busreset(raw13942_handle_t handle);

void raw13942_async_wait_for_data(raw13942_handle_t handle,
                octet_t address,byte_t * data,size_t length)
void raw13942_iso_wait_for_data(raw13942_handle_t handle,
                doublet_t channel,
                byte_t * data,
                size_t length);
```

- Operations: *set_user_data* is used to associate application defined data with a currently open device handle. *get_user_data* retrieves this associated user defined data. *generate_bus_reset* generates a soft bus reset. *set_busreset_callback* installs an application defined callback that will be called when a bus reset occurs. s*et_port* selects the IEEE 1394 card that subsequent operations should be performed on. *create_handle* is used to obtain a handle to the *raw1394-2* device. This handle is required by all the operations presented thus far. d*estroy_handle* is used to relinquish this handle.

```
void raw13942_set_user_data(raw13942_handle_t handle, void * user_data);
void * raw13942_get_user_data(raw13942_handle_t handle);
void raw13942_generate_busreset(raw13942_handle_t handle, int setroot);
void raw13942_set_busreset_callback(raw13942_handle_t handle,
       busreset_callback_t func);
void raw13942_set_port(raw13942_handle_t handle, int portid);
       raw13942_handle_t raw13942_create_handle();
void raw13942_destroy_handle(raw13942_handle_t handle);
```