Analysis and Modelling of the Windows mLAN Driver

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

BACHELOR OF SCIENCE

WITH

HONOURS IN COMPUTER SCIENCE

Rhodes University

2005

By

Shaun Miles

Supervised by

Professor Richard Foss

Abstract

The purpose of this thesis is to expose the inner-workings of the Windows music Local Area Network (mLAN) driver to aid in further research of the mLAN system. mLAN represents an innovative environment for networked high-speed transmission of real-time audio and MIDI streams. mLAN extends the IEEE 1394 architecture, which encapsulates the essential features of a networked real-time multimedia system. An exploration of the mLAN architecture and the Windows Driver Model gives us insight into the core concepts central to the creation of the mLAN driver. The Windows Driver Model is the required form a driver must follow. DriverStudio is a driver development tool that encapsulates the Windows Driver Model in a C++ library framework. A dynamic and static analysis of the mLAN driver is performed using the Windows Driver Model and the DriverStudio framework as reference. This understanding will be expressed by developing an object model of the mLAN driver, and the documentation of the I/O Control codes.

Acknowledgments

I would like to thank my supervisor, Prof. Richard Foss for his limitless help. Also thanks to Melekam Tsegaye who spent many hours helping me, and giving insights into the wonderful nature of drivers.

Table of Contents

List of Figures

1. Introduction

1.1 Problem Statement

The music Local Area Network (mLAN) is a proprietary technology that Yamaha began developing in 1993 [AE Notes, 2005]. IEEE 1394 (Firewire) was chosen as the networking technology for mLAN. The objective of mLAN is to replace the myriad of cables, plugs and "in", "out" and "through" sockets on synthesizers, mixers and other A/V equipment with a single connector. The mLAN system represents a solid advance in terms of networked real-time multimedia systems. It is characterised by two important features in real-time multimedia systems; high bandwidth and low latency.

Importance is attached to research within the Windows context, as Windows represents a more commercially viable marketing option. The driver is responsible for the PC interacting with the self-managed Firewire bus, which has mLAN extended nodes interfacing to audio studio equipment. The PC, through the driver, becomes a powerful tool as it can engage in connection management, and audio recording and playback, available as an application suit that comes with the system. Investigation into the full potential or the pursuing of other ideas within the mLAN system can be done through the use of applications.

1.2 Project Motivation

The aim of the project is to critically analyse the Windows mLAN driver with regards to an object oriented approach. In an effort to streamline the development of applications for research purposes, or the possible modification of the mLAN driver, an understanding of the driver is required. This understanding will be represented with use of an object oriented model.

1.3 Document Structure

Chapter two covers the mLAN system. This entails examining the underlying technology in use by the mLAN system. The IEEE 1394 bus architecture and the IEC 61883 specification for audio and data transmission are the technologies covered. The reasons supporting the features, as to why the technologies were chosen, are discussed.

There are not a lot of approaches to driver development, which is a wholly intricate and complicated process. One is constricted in developing a driver by the operating system and the type of driver it requires to interface with the hardware. To be more specific, each operating system has a model in which drivers must fit to interface with the hardware. In this instance, the focus of chapter three will be on the usage of the Windows Driver Model (WDM) in driver development.

Chapter four extends the previous chapter as the mLAN driver makes use of a proprietary Driver Development Tool from Compuware called DriverStudio. This makes use of the WDM, and in essence uses the Windows Driver Development Kit to build a driver. DriverStudio represents a framework of C++ libraries and classes that encapsulate driver development. It is relevant in examining the framework model of classes in order to understand the object model of the mLAN driver, and how they fit into the WDM.

Chapter five introduces the extrapolated model of the mLAN driver, explaining the usage of the classes and how they fit into the DriverStudio framework. A description of the methodology of analysis used in abstracting the object model and class interactions is presented. This chapter involves an overview of the class model structure, the classes involved and how they relate to each other. This takes a conceptual look at the underlying driver technology used by the mLAN driver, such the Windows Kernel Streaming architecture.

Chapter six discusses the work done to document the I/O Control (IOCTL) codes of the driver. The IOCTL codes represent the core functionality of the driver, as a majority of the processing occurs in response to IOCTL calls to the driver. It is through the IOCTL codes that applications can communicate with the driver.

Finally, chapter seven concludes the paper.

2. Music Local Area Network

Yamaha began developing a system to provide single cable connections between the arrays of standard audio studio equipment in an attempt to standardise digital audio data transmission. IEEE 1394 (Firewire) was chosen as the foundation on top of which to build the mLAN system, for it addressed the many aspects required of the proposed system. The mLAN system can operate independently of a central server or a host controller, not requiring the use or the power of a computer. Responsibility of management and control of the bus is automatic and falls to selected nodes on the bus. Using the computer as a tool to visually model the bus and to manage the connection of nodes, it extends the functionality of the mLAN system. This chapter briefly introduces the mLAN system by firstly covering the underlying technology. This deals with relevant IEEE 1394 and IEC 61883 concepts. Finally, the mLAN extensions to those technologies are presented.

2.1 IEEE 1394

2.1.1 The IEEE 1394 Architecture

IEEE 1394 is a specification that defines a serial bus architecture with a common set of core features that is an extension to the Control and Status Registers (CSR) architecture [Anderson, 1999]. The CSR is a standard definition to permit easier implementation of software, and allows interoperability between Firewire on different platforms. It is based on the ISO/IEC 13213 specification that standardises the offset locations within the initial register address space. The CSR specifies the arrangement of an addressable space range containing data structures that identify the components of the bus, be it a bus, a bridge or a node, the components state and other information.

A node is contained within the module and this represents a logical entity. A unit is a functional subcomponent of a node, for example, a 1394 node could have a Video and an Audio unit, each operated and controlled independently by its own software. Each node has its own Control and Status Registers (CSRs), as well as space for bus dependant registers, defined by IEEE 1394 [Anderson, 1999]. These registers allow for interoperability between Firewire implementations on different hardware platforms, for easier implementation of the software, and bridging between different buses types using the same CSR architecture. There is also a configuration ROM for identifying the vendor and node specific details. Up to 63 nodes can sit on one bus, except that a data item can be passed a maximum distance of 16 hops. Buses have unique ID's which allow for 1024 buses theoretically bridged together, although there are other physical limitations such as the maximum cabling distance [AE Notes, 2005].

Each bus has certain features that one node must handle (one that is capable), these being the bus management aspects of the bus. A single node, the determined root node, has the responsibility of commencing communication with the other nodes by sending cycle start packets, synchronising the nodes. Other responsibilities include the isochronous resource manager and the bus manager. Bus configuration is automatic and occurs upon the detection of a new node or the disappearance of a node, or a bus reset. The root is identified automatically, as well as the responsibilities allocated to capable nodes nearest the root [AE Notes, 2005]. The Firewire bus, as a result, manages all aspects of the bus itself.

Firewire supports two means of data transmission. One is asynchronous transfers which do not require a constant transfer rate, and targeted to specific unique node addresses. The other is isochronous transfers that require a transfer of data at constant intervals. The data is broadcast on a channel instead of to a unique node address, allowing one or more nodes to pick up on that channel. This transfer type requires regular bus access and thus has a higher bus bandwidth priority [AE Notes, 2005].

In figure 2.1, an overview of the IEEE 1394 protocol layers is shown. The protocol layers are a representation of the communication functionality of a Firewire node, simplifying and encapsulating the inherent concepts [AE Notes, 2005]. The Physical layer provides the physical interface to the serial bus, receiving and transmitting bits on the bus. The link layer is an interface for both isochronous and asynchronous data transactions. For asynchronous transactions, the link layer provides the interface between the transaction layer and the physical layer. The link layer for isochronous transactions interfaces the application level and the physical layer. The transaction layer is responsible for translating asynchronous transactions into the appropriate requests.

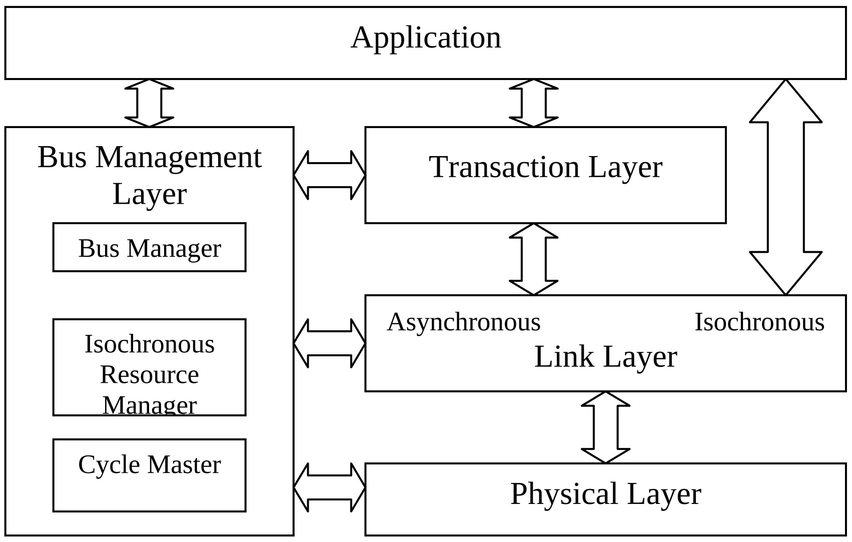


Figure 2.1: IEEE 1394 Protocol Layers [Anderson, 1999]

2.1.2 Virtual and Physical Devices

A computer must have a driver to know how to interface to any attached hardware device. IEEE 1394 is a bus, such that each device on the bus is seen as a node. A host controller (the card that allows the computer to connect to the bus) must appear as a node in order for the computer to interact with the bus [McKenzie, 2003]. In terms of the operating system interacting with the bus, it does so through the use of a bus driver and a host controller. A bus driver is responsible for interpreting requests to communicate with the Firewire bus from an application or a client driver. The host controller controls the physical I/O operations between the PCI bus and the IEEE 1394 bus.

A node on the IEEE 1394 bus needs a driver to tell the PC how to interact with the device. There can only be one driver for a host controller in a PC, limiting the number of concurrent drivers using the bus to one. A host controller is required for physical access to the bus, and if only one driver can use the bus at a time then the potential of IEEE 1394 technology is limited [McKenzie, 2003]. The Windows XP and later versions of the 1394 bus driver allow a user to enumerate devices through the 1394 bus using custom device IDs. These devices do not have to correlate to real hardware, and therefore allow the use of virtual devices. Virtual IEEE 1394 devices represent considerable work in rectifying the limitation of peer-to-peer 1394 communication. A virtual device represents a logical device that has the same functionality and privileges as physical devices. It has an address space and can engage in communication with nodes on the bus, and nodes on the bus can communicate with the virtual device. The diagram below presents the use of enumerated virtual Firewire devices between two PC's over a Firewire bus. The Plug and Play manager (Refer to Chapter 3) can be used to enumerate a virtual device such that it appears as a physical device. The ovals are to draw attention to the implementation of a new driver stack implemented to achieve the usage of virtual devices. The same driver stack is applicable for a PC-to-Firewire bus communication.

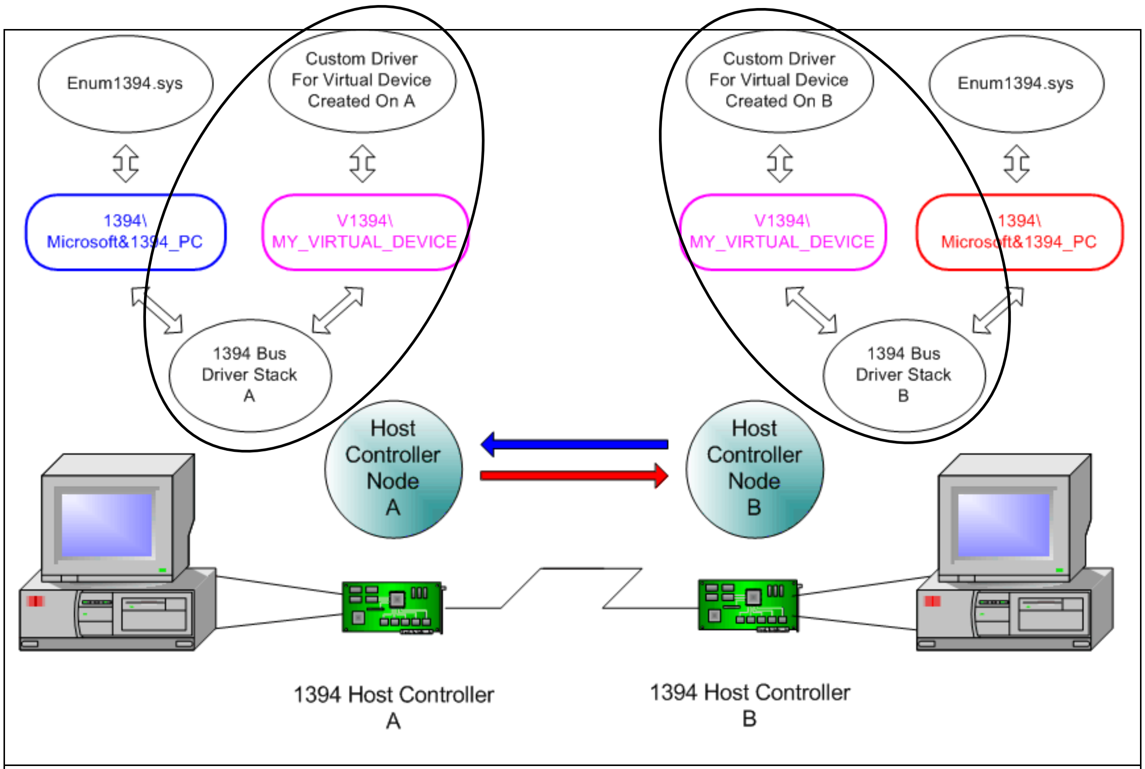


Figure 2.2: IEEE 1394 Peer-To-Peer Communication [McKenzie, 2003]

2.1.3 Summery of Firewire Features

Features responsible for the usage of Firewire as the mLAN system are as follows:

- High speed bus with scalable performance

  The throughput speed of the bus satisfies the high bandwidth of any real-time audio processing system, allowing a network of many devices capable of transmitting and receiving streams of audio data.

- Plug and play support

  The bus orchestrates the automatic configuration of newly added devices. Each time a device is added or removed the bus automatically re-enumerates itself, forming a new bus topology. This is done independently and without the intervention of a host system.

- Eliminate host processor/memory bottleneck

  Processing any amount of multimedia data requires a robust system to handle the large and time-dependant flow of data. By eliminating the need for a central server the bus is free to automatically direct the transfer of data between the devices, thereby removing a potential bottleneck. This is supported by the use of peer-to-peer transactions

- Support for Isochronous data streams

  Firewire provides support for isochronous packet streaming, satisfying the need for a constant transfer rate in an audio system. This is a high-priority transfer mode that guarantees delivery of data at constant rate.

2.2 The mLAN extension

The mLAN system extends the IEEE 1394 CSR architecture by implementing several different registers and by using a series of application level chips [AE Notes, 2005]. The chips, ASICs (Application Specific Integrated Chips), provide an interface between the IEEE 1394 node and the studio equipment. These chips interface with different IEEE 1394 protocol layers to integrate different functionality. Their particular concern is translating incoming and outgoing stream and control information between the studio equipment and the link layer of the IEEE 1394 node.

This encapsulates the formatting and timing of audio data, based on the IEC 61883 specification.

In terms of the driver, Microsoft ships low level drivers for all the major device and bus types for the vendor-supplied drivers to make use of. These vendor drivers form client drivers that provide access to the vendor specific functionality of the device. The Windows mLAN driver is a client Firewire driver that makes use of the Windows supplied IEEE 1394 driver. The mLAN driver is able to manipulate the Unit Directory (in the CSR) of the host controller via the underlying IEEE 1394 driver to make it appear as a mLAN device.

2.2.1 The IEC 61883 Specification

IEC 61883 is a set of standards that are related to the transmission of audio, music and multimedia over IEEE 1394 [AE Notes, 2005]. The mLAN technology uses two of these standards. Firstly, mLAN uses IEC 61883-1 for the general packet format, dataflow and connection management, and general transmission rules for command controls. Specifically, these are encapsulated in the definitions of the Function Control Protocol (FCP), the Common Isochronous Packet (CIP) format and the Connection Management Procedures (CMP) [Haig, 2002].
Secondly, there is IEC 61883-6, a standard based on a set of documents presented to the IEEE 1394 Trade Association (1394 TA) by Yamaha. Yamaha introduced the concept and specifications for mLAN, which described a specification for the transmission of audio and music data [AE Notes, 2005]. Thus began a 1394 TA document describing the Audio and Music Data Transmission Protocol (A/M Protocol). This document was adopted as an IEC (International Electrotechnical Commission) standards specification.

The specification controls the transfer bit-rate, the encapsulation and extraction of audio and MIDI data into sequences, and the construction of the Common Isochronous Protocol (CIP) packets. These CIP packets (comprising a CIP header and several CIP data blocks) make up the payload data of the isochronous packets that in turn form the isochronous streams. The CIP formatting is based on the IEC 61883-1 specification for data flow and connection management [AE Notes, 2005]. The payload of the CIP packet is the AM824 (8 bit label and 24 bit Audio/Music data) block format given by the IEC 61883-6 specification. The CIP header contains data that provides a mechanism for synchronising the reassembly of the audio/music data.

2.2.2 Data Transmission

The isochronous packets are constructed by the IEEE 1394 node Link Layer. Packets are identified by a channel number that has a single corresponding stream. Each packet within a stream comprises several data blocks, which are made up of a number of quadlets (4-byte blocks) [AE Notes, 2005]. Sequences are designated by the series of quadlets corresponding to a position in each data block of that packet. Sequences and plugs are core to the functionality of the mLAN system, in that each sequence is issued from an output plug and gathered by an input plug [AE Notes, 2005]. A plug is an abstraction formed by the Enabler, representing the input and output capabilities of the studio equipment. The Enabler is software on the computer that interacts with the Transporters (device-specific enumeration of the plugs) and allows connection a management. It is the plugs that the studio equipment can send and receive data on the bus.

2.3 Chapter Summary

Firewire is a flexible bus architecture that is ideally suited to real-time systems, as time-dependant data can be transferred in a deterministic manner. Through extensions to the architecture, the potential of Firewire is realised. One such extension is the mLAN usage of integrated chips to achieve the required audio and control requirements for a real-time multimedia system. Analysing the mLAN driver will give insights into the usage of technologies presented in this chapter. In order to understand the driver, the Windows driver model architecture is presented in the next chapter.

3. The Windows Driver Model

3.1 Introduction

The Windows Driver Model (WDM) is a framework for designing and developing drivers for NT based Windows platforms. WDM drivers have well-defined responsibilities in facilitating I/O operations for applications and other drivers [Oney, 2003].

WDM drivers are kernel-mode drivers that share many of the design goals of the windows operating system. By coding a driver in C and avoiding using the standard C runtime libraries, the driver becomes portable among the Windows platforms. There is a kernel mode run-time library available for drivers to make use of; which includes string and list management routines. Also taking care not to use data types that are size dependant on the platform will ensure portability. Having the ability to dynamically configure the hardware requires the driver to be more flexible. This implies that the device and its driver support Plug and Play. The driver also has to be able to cope with being interrupted by other events, and suffer less processing time without locking up the system [MSDN, 2005].

The core concept of the WDM is the driver stack. According to Oney, the Windows operating system comes with base drivers that take care of generic I/O operations for standard hardware devices. In this architecture, a driver is supported by a chain of other drivers below it. This is moving away from monolithic drivers, which handle every action between an application making a request of a hardware device and the application receiving the desired response. The layered approach is such that at each level of the stack a driver in that stack has the opportunity to act on a request being sent down. If the driver cannot fulfil the request it will pass the request down the stack until the request can be satisfied. The results are then passed back up the stack and back to the application that made the request. This approach allows drivers to focus on a small area of functionality and specialisation [Oney, 2003].

Figure 3.1 illustrates a hypothetical driver stack, showing the different types of drivers a stack could have. Our focus is on the kernel-mode drivers, and there is a need to clarify their functions. The kernel-mode client driver handles requests from an application via the Win32 API. The class driver, usually supplied by Microsoft, provides the system-required but hardware-independent support for a class of device. The miniclass driver is usually supplied by the hardware vendor to integrate any unique functionality their device may have. The port driver (or in some cases, the host controller or host adapter driver) supplies required I/O operations on underlying physical devices or busses connected to the device. Once again, the miniport driver is tailored for vendor specific operations. The hardware bus driver is supplied by Microsoft and should not be replaced.

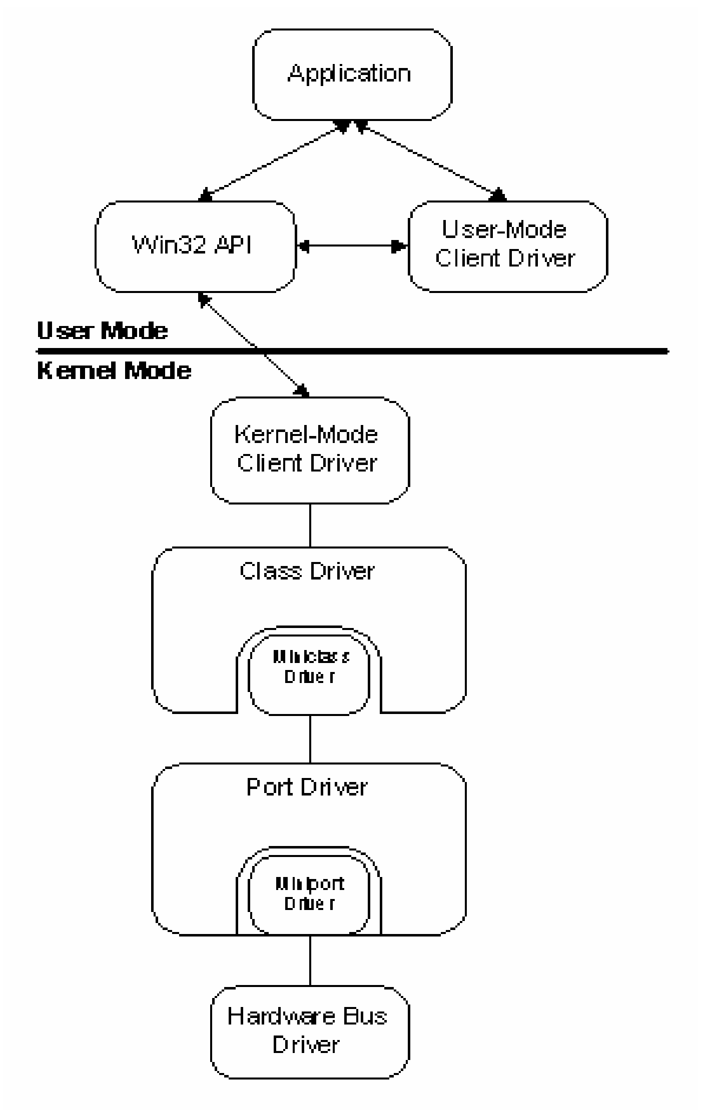


Figure 3.1: Layered Driver Architecture [MSDN, 2005]

An application trying to make use of a hardware device will do so through the Win32 API. It is the bridge between the user and kernel mode. In the kernel mode, a thread or a driver has access to system memory and hardware [MSDN, 2005]. In order for an application to access these system resources, their requests have to be mediated by the kernel components. There are two important concepts that have to be dealt with here, one being the driver communication channel and the other being the kernel mode operating system components. The kernel mode system communicates with the driver stack through I/O Request Packets (IRP's). These are reusable objects that have a predefined and unique code identifying them and their function. This will be covered in more depth later in this chapter.

3.2 Operating System Components

The operating system has certain components that monitor and allocate system resources as they are required. There are three core components that are relevant to drivers functioning within the kernel mode space. These are the PnP manager, the I/O manager and the Power manager. Another component relevant to the functioning of the kernel but specifically for kernel-mode drivers is the object manager. The Object Manager stores and manages all system objects, including device and driver objects. The I/O Manager makes use of the Object Manager to access and store the objects it uses [MSDN, 2005].

3.2.1 Plug and Play Manager

The PnP manager is firstly responsible for identifying the device that has been added to the system's environment. Each device is registered with a GUID, making it identifiable to the OS. As part of the Windows OS, a registry keeps track of vendor specific device hardware, identifiable through a unique identifier called a GUID (Global Unique ID). The GUID is a 128bit device identifier that is published for use in the IT industry, creating a unique association identifying the device with a GUID [Oney, 1999]. That means each device is registered with a GUID, making it identifiable to the OS, allowing the OS to load the correct driver. A driver is released with an INF file containing the GUID of the class of device the driver is responsible for, as well as other driver related information.

The PnP manager reacts to a newly detected device by searching for the correct driver to load if it has not already been loaded. If the PnP Manager cannot find the device it will ask the user to provide the system with locations for it to search. It responds to run time hardware events, for instance the addition or removal of a device from a bus [MSDN, 2005].

Resource requirements for devices are determined by the PnP Manager and are allocated appropriately. Dynamic reconfiguration of the hardware resource allocation can take place as required. Hardware resources include assignable, addressable bus paths that allow devices and system processors to communicate with each other. The PnP Manager handles power requests such as stopping or starting the device, indicating the intention to remove a device, and responds to other power related events. These requests are sent via the I/O Manager in IRP's using power function codes.

Finally, the PnP Manager is responsible for enumerating a newly added device, and any subsequent devices that the driver controls on its bus. This means that the device object stored in system memory is linked to the driver by calling the driver *AddDevice* routine and passing it the pointer to the device object. The PnP Manager keeps track of which devices are active through a list of the Device Objects. Device objects will be addressed further on in the chapter [MSDN, 2005].

3.2.2 Power Manager

The Power Manager is responsible for maintaining different power states for the entire system and the connected components. At certain times a device can sit idle allowing the power state to be lowered, so that the efficiency of the power usage can be increased. This is achieved through different power policies which can be acted on depending on the use of the device. The Power Manager, via the I/O Manager, can send power function codes contained in an IRP to the relevant driver. Drivers should implement the required power functionality in terms of handling the IRP's that are sent to it [MSDN, 2005].

3.2.3. I/O Manager

The I/O Manager forms part of the I/O Model for the Windows operating system. The I/O Manager's function is to facilitate I/O communication between an application and a device via a driver stack. This communication stream centres on the usage of IRP's to formulate a request to the driver stack that indirectly controls the device. These IRP's are routed to the driver stack where each driver has the chance to respond to the IRP until the request is satisfied. Higher level drivers break the IRP into smaller requests and route them to lower level drivers appropriately [Cant, 1999].

The I/O Manager also defines a set of required and optional routines that drivers should implement. Further, each driver should have a handler routine for each IRP of that set, which are basically entry points into the driver code and thus allows the driver to react to a request. In the table below, an example of a small set of IRP major function codes that have to be implemented, with their WIN32 function that will result in the relevant IRP being dispatched [Cant, 1999].

Win32 Function	IRP Major Code	Base Driver Routine
CreateFile	IRP_MJ_CREATE	Create
CloseHandle	IRP_MJ_CLOSE	Close
ReadFile	IRP_MJ_READ	Read
WriteFile	IRP_MJ_WRITE	Write
DeviceIoControl	IRP_MJ_DEVICE_CONTROL	DeviceControl

The I/O Manager knows implicitly what entry routines there are for each driver, and this is achieved by the I/O Manager keeping a list of Driver Objects. A driver object is equated with a Physical Device Object (PDO). A driver object is created by the I/O Manager when a driver is first loaded and installed. During the driver initialisation, the driver object is passed to the first driver entry routine called *DriverEntry*, which then associates its other entry routines with the driver object [MSDN, 2005].

The WIN32 API communicates with the I/O manager, which builds the appropriate request to be sent to the correct driver. The driver is represented by a driver object, while a device that belongs to the driver is represented as a device object. These objects are stored in system memory, and are created by the system as they are needed. On the receipt of an I/O request, the I/O manager knows the target device of the request (from the *DeviceIoControl* call of the application) and requests from the object manager a pointer to the driver object responsible for the device. An IRP is sent to the dispatch routine of the driver object, which then picks out which device it is targeted at and routes the IRP there.

3.3 I/O Request Packets

The core function of a driver is to respond to I/O requests from the system or application. The driver handles the request by processing the IRP; if the driver cannot do that then it passes the IRP to the driver below it in the stack. Each driver in the stack should be ready to receive any IRP and handle any error.

3.3.1 IRP Data Structure

IRPs are essentially data structures that contain certain fields, defining the type of action to be performed. Each IRP has an action defined by a function code. There are two types of function codes for an action. One is a major function code which indicates the main action, while the minor function code further specifies the action.

When an application sends a request for a certain operation from a driver controlled device, the I/O Manager accesses the object manager in order to resolve the request [MSDN, 2005]. The object manager is a core system component that manages all stored system created objects and data structures, keeping pointers to their locations. The request will want to make use of some service offered by a device and the object manager returns a pointer to the device object.

From here the I/O Manager builds and allocates memory for the IRP, and initialises certain fields which relate to the positions of drivers in the I/O stack. The IRP is passed to the top of the relevant driver stack. The driver then accesses the I/O stack to determine the current operation it has to complete.

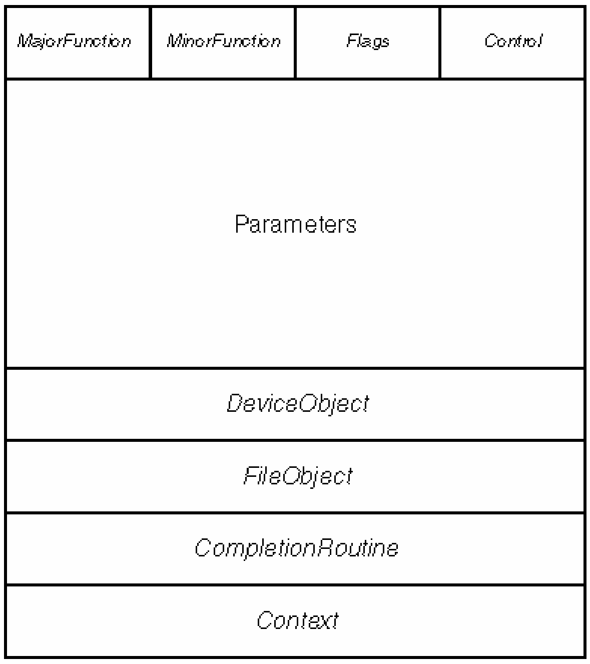


Figure 3.2: I/O Request Packet Structure [Oney, 2003]

3.3.2 I/O Stack

When an IRP is created, an array of IO_STACK_LOCATION structures are created and associated with that IRP [Oney, 2003]. Each stack location represents one of the drivers that will process the IRP, basically defining the driver stack. Each structure contains the function codes and parameters that describe the action of the IRP. The I/O stack location can also save contextual information about the operation of the current driver. This way the drivers can act on different levels, allowing the incremental completion of the task.

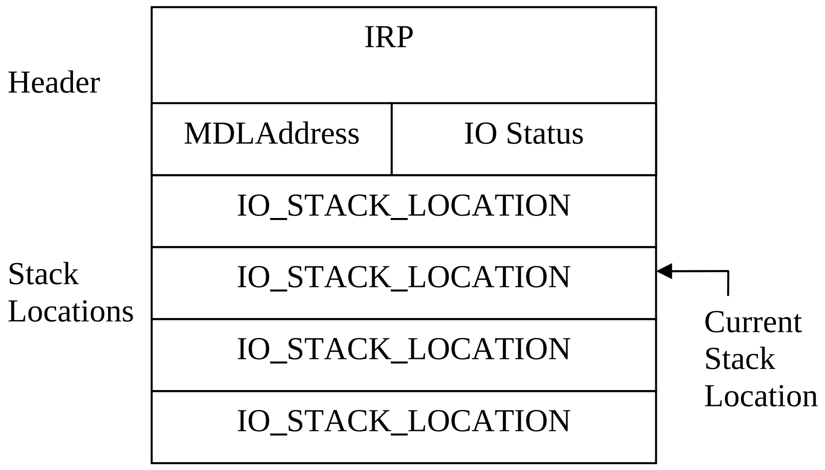


Figure 3.3: IRP Structure Relating to Stack Locations [Cant, 1999]

It falls to the current driver to access its associated I/O stack location structure, thereby accessing the IRP function code and its parameters. When the driver is done with the processing, the driver increments the I/O stack pointer to the next stack location and then passes the IRP to the next driver [MSDN, 2005].

3.3.3 IRP Queues

Since the system supports asynchronous requests within a multitasking and multithreaded context, drivers may not be able to finish processing the current IRP before another arrives. WDM drivers should then support driver queues, which the I/O Manager associates with each device object a driver creates [MSDN, 2005]. The lower-end drivers that support the I/O operations of a device supply a *StartIO* routine. IRPs that are queued by a driver using the I/O Manager's support routine, are then queued to the *StartIO* routine.

3.3.4 I/O Control Codes

Due to the nature of this project, an understanding of I/O Control Codes (IOCTLs) is important in the core features of any driver. I/O control codes are a communication channel between user-mode applications and drivers [MSDN, 2005]. Typically, IOCTLs are sent to drivers by applications using the *DeviceIoControl* call. When this happens, the I/O Manager creates an IRP concerning device or internal device control, including the IOCTL code. This results in the IRP being sent to the upper-most driver in the stack. This allows your driver to respond to an IOCTL sent by an application.

3.4 Driver and Device Objects

These objects form the core representation of the WDM architecture for the OS. They represent the driver and the devices under the driver's control, and store information relevant to the driver's operation. An in depth explanation of these objects is required to further understand the WDM architecture.

3.4.1 Driver Object

The I/O Manager creates a driver object for each driver that has been loaded and installed. They are defined using DRIVER_OBJECT structures. The driver object represents the driver itself and contains a list of pointers to all the device objects of the devices under the control of the driver [Oney, 2003].

The driver object sets the pointers to driver entry-point functions during driver initialisation. For a driver to receive IRPs, it needs to contain the corresponding entry point to the IRP handler routine. For instance, to receive an IRP major code of PNP there has to be a PNP dispatch routine published by the driver object so that the I/O manager knows the location of the function to dispatch the IRP to. Each entry point corresponds to an IRP major and minor function code. If a driver manages its own IRP queue then the driver object should contain an entry point to a *StartIO* routine. For non-persistent drivers, an entry point to the *Unload* routine can be called to free up system resources [MSDN, 2005].

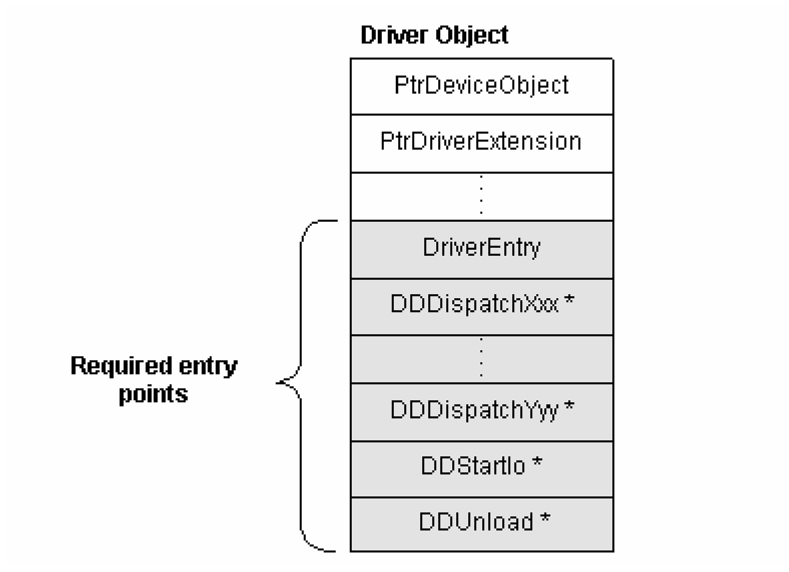


Figure 3.4: DRIVER_OBJECT Structure [MSDN, 2005]

Figure 3.4 depicts a typical driver object containing the accessible fields of the data structure. PtrDeviceObject is the pointer to the list of device objects for which the driver is responsible. The I/O Manager uses this list to connect the devices together. PtrDriverExtension contains the address of the *AddDevice* routine the PnP Manager will use to add and enumerate new devices under the control of the driver. The DDDispatchXXX entry points correspond to each of the major IRP codes the driver will handle.

3.4.2 Device Objects

The operating system represents devices using device objects. These are the target for all functional operations. The driver object holds pointers to the routines contained in the device object. There are usually multiple device objects for a device, one for each driver in the stack [Oney, 1999].

The device object uses a structure called DEVICE_OBJECT to represent itself, which is managed by the Object Manager. Each object can be named and as result has a handle to easily access it. Additionally, each device object has dedicated system storage space for maintaining device state and storing other driver related data structures. This space is represented in another data structure called the driver extension, particular to a driver and will incorporate all the devices that it is responsible for [MSDN, 2005]. The I/O Manager’s *IoCreateDevice* and *IoCreateDeviceSecure* routines allocate non-paged memory for the device object and extension. Every standard driver routine that receives an IRP can also access the driver extension via the pointer to the device object contained within the IRP.

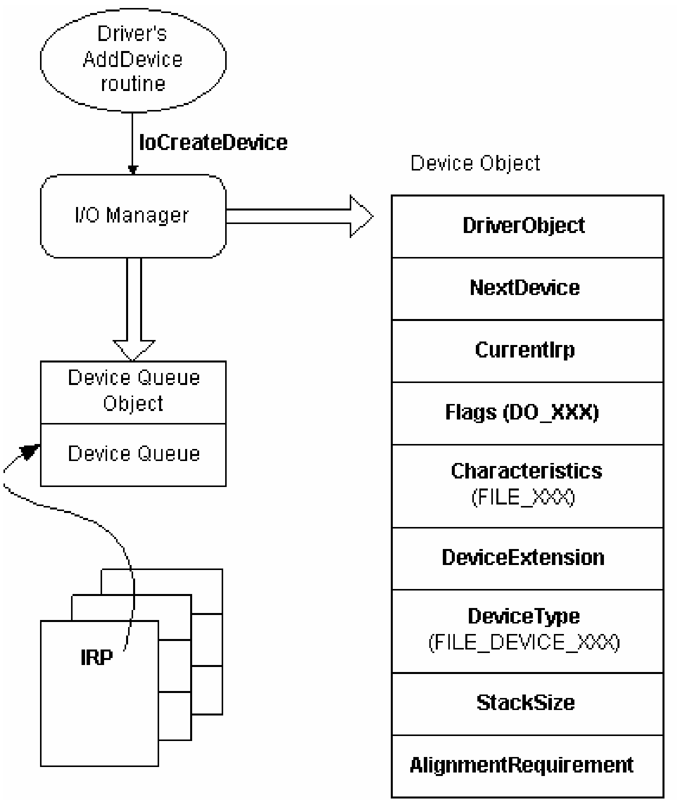


Figure 3.5: Device object creation [MSDN, 2005]

To create a device object, the driver calls the *AddDevice* routine. The actual object is created through a system call *IoCreateDevice*, which then initialises the object. The device object has certain properties defined through accessible fields, as shown above. These properties describe how the device object interacts with the system. There are different types of devices, with a corresponding type of driver, each interacting with the system in a different manner. Security and exclusivity aspects of the device and the driver are defined here [MSDN, 2005].

There are three kinds of WDM device objects:

- Physical Device Object (PDO)
- Functional Device Object (FDO)
- Filter Device Object (Filter DO)

The PDO represents and is created by a bus driver, with certain responsibilities and functions. These include enumerating and administrating the devices on the bus, responding to PnP and Power IRPs.
The FDO is the functional driver's device object, which is the main driver for the device. Its purpose is to provide an interface to the operation of the device.
A Filter DO is an optional driver that can modify the behaviour of the driver depending on what is required. For instance, if you wanted to monitor the functioning of the device, or you wanted the device to conform to expected specifications, this would be done using a filter driver [MSDN, 2005].

3.5 Entry Routines

As established above, IRPs are the communication streams from an application to a device in order access its resources [Oney, 1999]. Entry routines are the entry points into the driver code. They represent the different functionality of the driver by implementing standard routines. These standard routines are called under different circumstances, and are required to provide the functionality for which they are named. The amount of required routines increases as the amount of functionality and support the driver is designed to implement. Lower-end drivers, those that directly control access to the devices resources will have more required routines than the higher-end drivers.

3.5.1 DriverEntry

As discussed earlier, the *DriverEntry* routine is called by the I/O Manager the first time a driver is loaded and installed by the system [MSDN, 2005]. Any initialisation is taken care of here, setting up any required parameters. The main purpose of the *DriverEntry* routine is to fill in various pointers to functions in the driver object. These are pointers to the other subroutines of the driver. The I/O Manager makes use of these pointers when it builds an IRP. The subroutines represent the other entry points to the driver, including the dispatch routines, which receive and dispatch IRPs.

A pointer to the driver object is passed as a parameter of the *DriverEntry* routine when the driver is first loaded.

3.5.2 AddDevice

The *AddDevice* routine's responsibility is to create and initialise the driver's representation of the device object for each device enumerated by the PnP Manager [MSDN, 2005]. *AddDevice* routines are called during system initialization (when devices are first enumerated), and any time a new device is enumerated while the system is running. During device object creation, an association between the driver object and the new device object is created. Memory is allocated for the device object by the kernel system, and a pointer to the object is passed as a parameter of the *AddDevice* routine. The driver must provide storage, usually in the device extension of a device object, for pointers to certain objects obtained from the I/O manager or other system components.

3.5.3 Dispatch Routines

In order for a driver to process an IRP, it must have handler code for that IRP [MSDN, 2005]. Dispatch routines are supplied to handle one or more IRP major function codes. The driver's DRIVER_OBJECT structure has within it a dispatch table that holds the dispatch routines supplied by the *DriverEntry* routine. The I/O Manager uses the pointer to the relevant dispatch routine to dispatch the IRP for processing. The exact functionality a dispatch routine should provide depends on the I/O function code it handles, on the driver's position in the stack, and on the type of physical device it supports. This is where the driver decides how to process the IRP, or pass it on down the stack. This is an integral part of the driver's functionality in handling IRPs. There are certain dispatch routines a driver must support. These routines can be supported as far as the driver requires.

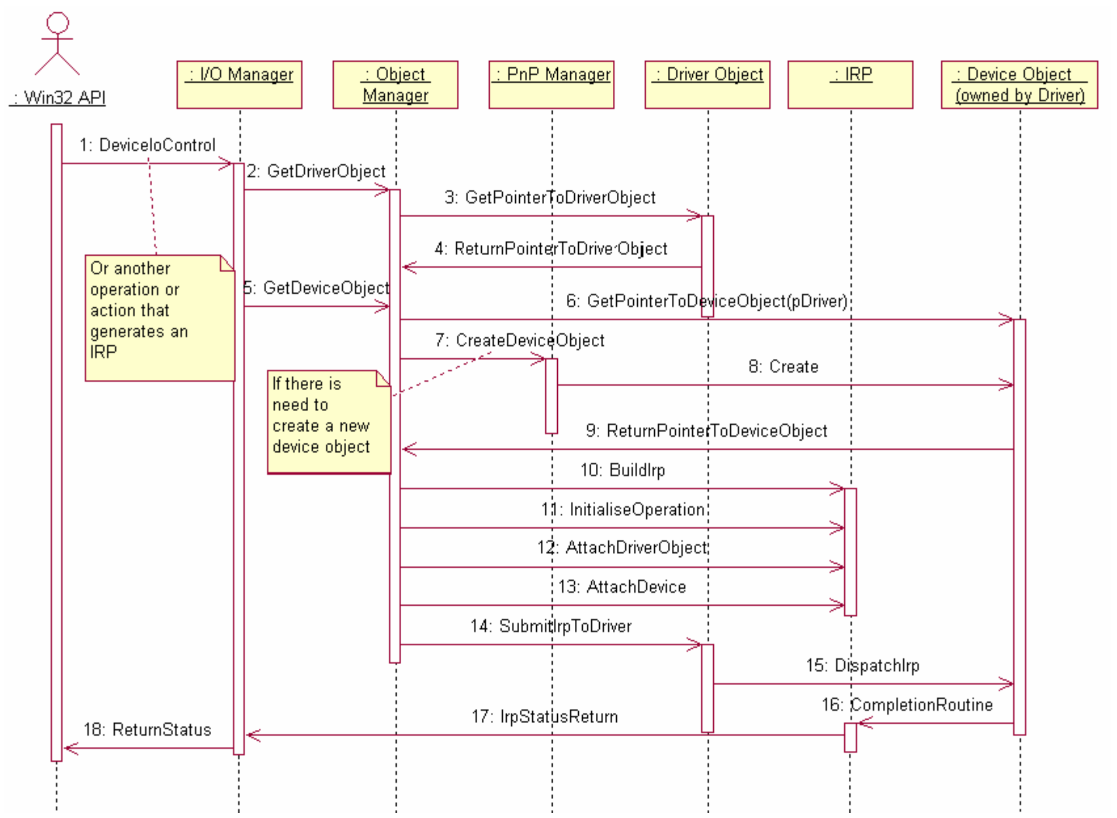


Figure 3.6: Relationship between I/O manager and IRP processing

The above figure is a sequence diagram expressing the relationship between the I/O manager and the driver’s IRP processing mechanism. Firstly, the I/O manager will respond to an application’s request for an I/O operation, or some request that has to be handled by a client driver. As an aside, the PnP manager can generate its own IRPs although it has to go through the I/O manager. An I/O operation is always directed at a device object, represented as a handle in user-mode. The I/O manager finds the driver object responsible for the device to get the device object. The object manager is in charge of managing system objects, and will return a pointer to an object should a request be made. The I/O manager then builds an IRP, specifying the relevant fields and in particular, using the driver and device object pointers in specifying the target for the I/O operation. The location of the dispatch routine is extracted from the driver object’s table of dispatch routines, and is used to call the routine to pass it the IRP. The driver will route the IRP to the target device’s IRP handler routine for that IRP major function code to be processed.

3.5.4 StartIO

The *StartIO* routine is used exclusively with an IRP queue, and is called in response to an IRP being ready in the queue for serialised IRP processing. In higher-end drivers this may inhibit performance, and may face certain inoperability factors [MSDN, 2005]. The processing of IRPs may be slower due to the *StartIo* routine becoming a bottleneck. It is recommended that higher-end drivers use internal queues.

For lower-end drivers, the support of the *StartIo* routine becomes the foremost focus, especially when considering that *StartIo* is responsible for starting any I/O operation on a physical device [MSDN, 2005]. Control over the processing of IRPs can be instituted by creating queues for their different function codes. Serialising the incoming IRPs can increase the throughput processing of I/O requests.

3.6 Chapter Summary

Following the Windows Driver Model is explicitly required for any kernel-mode driver, ensuring a reliable architecture for the representation of drivers and decoupled modularised driver stack for I/O operations. This separation and specialisation of drivers is inherently more flexible and sturdy than the legacy monolithic driver approach. In terms of development, drivers can be written to take advantage of established drivers supplied by Microsoft for generic I/O operations on standard device hardware. There is still a proportion of non-core driver code that the WDM requires of the driver to implement for administrative purposes. This is code that most drivers are required provide, specifically the handling of IRPs the driver may not explicitly need. The next chapter deals with a driver development tool that smoothes the driver production process by providing a framework of classes to neatly encapsulate required generic driver functionality.

4. Compuware DriverStudio

The development of the Windows mLAN driver was done with the help of a proprietary Driver Development Environment called DriverStudio. Any kernel-mode driver has to conform to the WDM and support Plug and Play functionality. So there is basic functionality that any and every driver should implement. In order to streamline and optimise the development of a driver, Compuware developed DriverStudio to work in conjunction with Microsoft Visual Studio. DriverStudio is a C++ class library that forms a framework for driver development [DriverStudio, 2004]. It is an object-oriented approach to writing drivers, with the DriverStudio classes denoted by a leading ‘K’. This also allows for code generation of classes and features of the required aspects of a driver. This involves the use of special classes to encapsulate the concepts inherent in driver writing, forming a hierarchy of classes. However, a developer can deviate from the structured framework supplied by DriverStudio if the driver’s functionality requires special support.

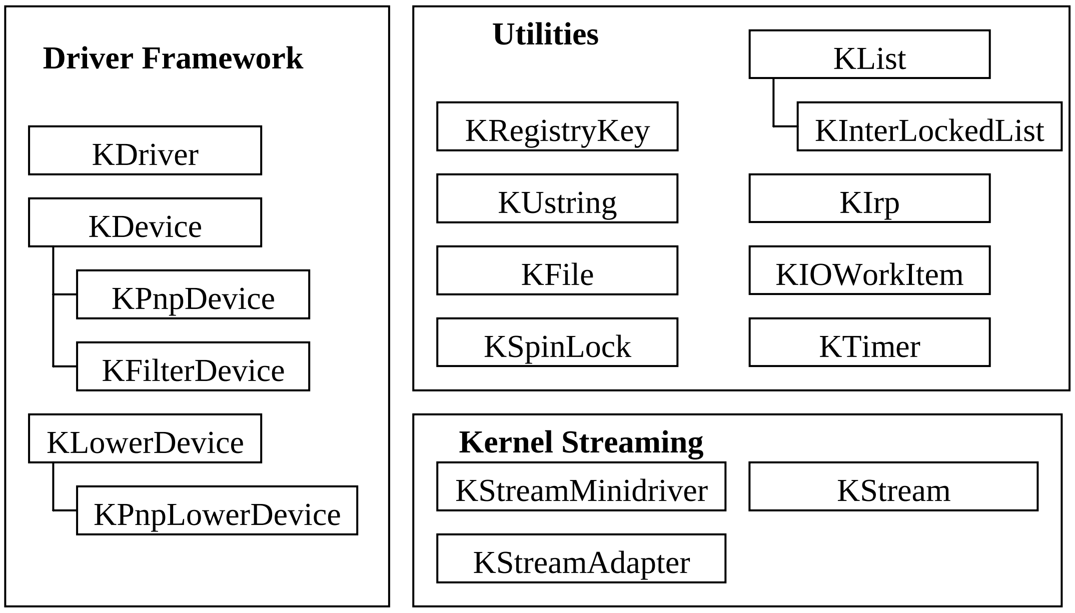


Figure 4.1: DriverStudio Classes [DriverStudio, 2004]

The above diagram depicts a subset of the DriverStudio classes used by the mLAN driver. The framework is a collection of objects and classes that form a foundation and a skeleton for the quick and efficient development of a driver. The classes and objects are built specifically to closely model the WDM architecture. The classes that driver writers use to derive their own classes for their drivers contain functionality and resources to allow for the integration of the driver into the framework. The objects provide functionality to allow for the ease of working with data structures within the driver context. The purpose of the framework is to provide a model for the communication between the different components, which encapsulates the flow of data required to facilitate the functioning of the driver. There are certain classes that require subclasses to be derived from them, in order for the driver to provide specific functionality.

4.1 KDriver

The driver object is implemented by the class KDriver, which contains the entry points into a driver. It is the class responsible for initializing the driver, and for directing I/O requests to the device objects to which they are targeted. It is an abstract class that requires that a class be derived from it and certain member functions be overridden. Each driver built using the framework distinguishes one class as the driver class, such that only one instance of KDriver can exist for a driver. KDriver has a macro member function called *DECLARE_DRIVER_CLASS* which must appear exactly once, outside of any function. This sets up the framework for the class derived from KDriver. The macro dynamically allocates an instance of the specified class from the non-paged pool. By default the class library framework routes and handles the IRPs for processing. This can be changed by overriding the *DispatchFilter* member function, and informing the framework that it should use this method.

4.1.1 DriverEntry

All classes derived from KDriver must implement this member function. KDriver's *DriverEntry* differs slightly from that specified by MSDN. In the MSDN documentation there are two parameters, a pointer to the DRIVER_OBJECT structure created by the I/O Manager and a pointer to a string containing the path to the driver's registry key. KDriver's *DriverEntry* just takes the registry path parameter, loads and validates the path. Using the registry path, member variables can be updated with values stored in that key. Any initialization of the driver object will take place here. For example, explicit use of the *DispatchFilter* routine can be set up here.

4.1.2 DispatchFilter

If the *DispatchFilter* routine has been enabled in the framework, then all incoming IRPs to the driver will be routed to this routine first. This member function allows the preprocessing of IRPs, as well as the entry point for IRPs. This allows the driver to monitor all the I/O requests from a single point. Every IRP has to be processed, returning the status of the completed IRP to the I/O Manager.

4.1.3 AddDevice

The PnP Manager, by way of the I/O Manager, prompts the framework to call *AddDevice* when a new device that the driver is responsible for is detected. It receives a pointer to the device object created by the system to represent the newly detected device. This is generally the PDO created by the system to keep track of the enumerated devices. The *AddDevice* member function usually creates the driver's functional representation of the physical device called the FDO. The FDO is created using the PDO (the physical representation of the device), and is usually represented by an instance of a subclass of KPnpDevice. The FDO has the responsibility of being the recipient of the IRPs sent by the system or the driver's dispatch routine. An association between the PDO and FDO is formed by creating an instance of a KPnpLowerDevice object. This will be covered later on in this chapter.

4.2 KDevice and subclass KPnpDevice

As mentioned in the previous chapter, a device object represents the virtual or physical device that is the target of I/O operations. KDevice is the base class from which new classes are defined, and instances created. The member functions of KDevice model system services that deal with device objects. KPnpDevice is a subclass of KDevice that has extra support for PnP drivers. It is a class that requires certain member functions to be overridden. A driver writer can set different policies regarding PnP and Power management. These policies allow the handling of PnP and Power related IRPs in specific ways, responding to a certain set of minor function codes.

It is the responsibility of the device object to provide handler member functions in response to IRPs directed at it. The IRPs that get dispatched to the device object have to be handled in a way that satisfies the request. An IRP could contain a request for the driver to handle some I/O operation, a PnP or a Power state change. Member functions within the KDevice and KPnpDevice class streamline the handling of the IRPs. The classes also have member functions available for the system (via the framework) to inform the device object of the resources under its control. Other member functions are for the release of these resources. Within the constructor of the subclass of KPnpDevice, it is critical to attach the KPnpLowerDevice object to the device object to register a clear path of communication. When this is done, the framework alerts the I/O manager to the addition of the lower device object to the driver stack.

It is through the device object that the framework defines the upper edge of the driver. The upper edge of the driver is the part of the driver that provides an interface between the system and the lower edge, with communication through the use of IRPs. The lower edge driver is usually an interface directly to the hardware or to another driver. The lower edge device objects are distinguished as being un-owned by the driver class, which provides a different functionality set and allows the decoupling of the driver stack. The lower edge driver is the framework’s representation of the drivers lower down in the driver stack. These objects are described in the next section.

4.3 KLowerDevice and subclass KPnpLowerDevice

The Windows Driver Model is a layered architecture. Low level drivers interface to the hardware and provide access to its resources. Intermediate level drivers provide additional translation and support for requests. High level drivers handle the initial I/O requests and begin the processing of the IRPs [DriverStudio, 2004]. Drivers that receive an I/O request have the option of handling the IRP and processing it, or passing it on down the stack, either in the form of the original request, or as a set of new requests. In this way drivers can communicate with other drivers and can make use of services available from other drivers rather than redundantly implementing everything again. Requests are always addressed to a device object within the driver, although the request may have to go through several drivers before reaching the target. The lower device object is that system device object that receives and processes the hardware level I/O requests. The main purpose of the lower edge driver class is to decouple the upper and lower edge of a device, allowing the layering of drivers. The result is that drivers have a limited set of functions for which they are responsible.

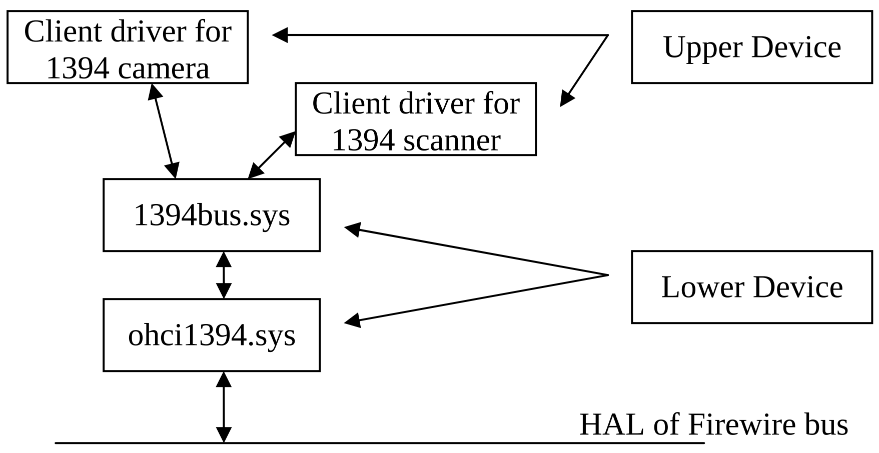


Figure 4.2: Hypothetical Driver Stack

The above diagram illustrates the usage of a lower device object within a stack, further showing the benefit of the layered architecture.

KPnpLowerDevice is a subclass derived from KLowerDevice that encapsulates extra PnP functionality. If a driver is WDM compliant (supports PnP and Power Management) then the lower device object will be of type KPnpLowerDevice. The PDO is modelled by the class KPnpLowerDevice [DriverStudio, 2004]. There are two ways to create and initialise a lower device instance; one way is to pass the PDO and the FDO pointers as parameters to the constructor, and the other way is to use a constructor without any parameters and call the *Initialize* member function. The *Initialize* routine takes the FDO pointer and the PDO, where the PDO is the device object passed by the system when it called the *AddDevice* routine. When the driver creates an instance of a subclass of KPnpDevice, the PDO is a parameter of the constructor that is used to initialise the instance of KPnpLowerDevice. The FDO pointer is that of the KDevice subclass instance. During the lower device creation and initialisation the PDO is attached to the FDO, returning the pointer to the device object that was attached. It is then the responsibility of the KPnpDevice subclass constructor to call a member function *SetLowerDevice*.

4.4 Non-Driver Framework classes

These groups of classes represent the container and utility classes that provide structured support for developing kernel-mode drivers. To program kernel-mode or system components, the correct usage of objects and data types is required. These classes model system data types and objects, and required generic driver functionality and expose a high level abstraction for use by the driver. These classes are developed specifically for use by drivers of the DriverStudio framework, incorporating tedious but necessary functionality. The KInterLockedList class, for instance, provides an implementation of doubly linked lists that can be serialised and uses a built in spin lock to enable safe access to the list. Another example of useful functionality is the KIoWorkItem which encapsulates a system request to execute a callback, allowing the queuing of a work item that will get executed by the system’s worker thread.

4.4.1 KIrp

The KIrp class abstracts the IRP sent by I/O manager and simplifies its use. An IRP is a complex data structure that leads to code for interacting with an IRP to be as complex. The difference in use is illustrated by the code snippet below, showing the easily readable usage of the KIrp object versus the relatively complex C equivalent.

```
// Straight C version
PIRP pIrp;
IO_STACK_LOCATION pStack;
pStack = IoGetCurrentIrpStackLocation(pIrp);
switch (pStack->Parameters.DeviceIoControl.IoControlCode)
{
   . . .
}

// Using class library
KIrp I(pIrp);
switch (I.IoctlCode())
{
   . . .
}
```

Figure 4.3: IRP versus KIrp usage [DriverStudio, 2004]

The KIrp class provides easy to use accessors for involved operations to access the various fields of the IRP. Simple operations are provided for generic operations on the IRP, closely modelling the behaviour of the IRP.

4.4.2 KDriverManageQueue

It is the responsibility of the driver to implement a queue for I/O requests, should the driver require serialised IRP processing. The KDriverManageQueue framework class provides a utility class for the serialised queuing of incoming IRPs. Conceptually, all incoming IRPs are routed to the queue to wait for when the driver is ready to process the next IRP. This saves the driver writers the extra effort of implementing a working queue.

4.5 DriverStudio’s Kernel Streaming Services Framework

Kernel Streaming (KS) services support kernel-mode processing of data streams for audio and for other types of continuous media [MSDN, 2005]. The current Windows provided multimedia streaming driver in use that encapsulates KS services is the AVStream minidriver. In order to utilise this service, you have to develop a minidriver that runs as an AVStream minidriver. Below is a diagram showing the KS architecture.

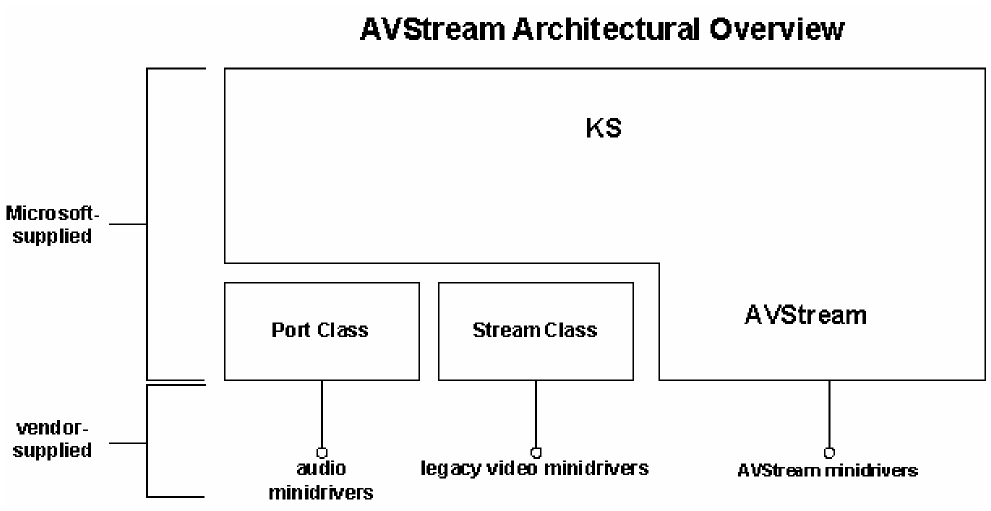


Figure 4.4: AVStream Architectural Overview [MSDN, 2005]

The port class driver provides an implementation of a KS filter driver, encapsulating the model of kernel-mode nodes processing a stream between a start and end point. The Stream class driver is provided for backward compatibility with KS 1.0 version video drivers.

DriverStudio provides a framework of classes for encapsulating the functionality of the KS Architecture, shown below in the object model of the framework. These classes are KStreamMinidriver, KStreamAdapter and KStream.

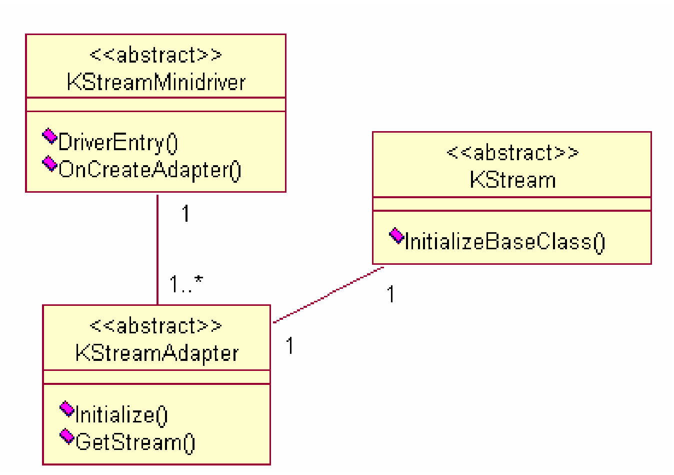


Figure 4.5: DriverStudio Kernel Streaming Framework

As a benefit of using the DriverStudio framework, driver writers do not have to worry about the specifics of the underlying KS architecture. It is required that there is a derivation of the three classes that make up the architecture. The Minidriver class is responsible for the control of the stream adapter class. In essence the Minidriver can be viewed as a filter driver, because it acts on behalf of the mLAN driver to process data streams. An adapter is in turn responsible for control of a stream resource. The adapter class controls an individual physical hardware device or layered software components above a hardware driver. The minidriver and adapter objects provide control over the device driver and hardware while the stream object provides control and data handling for an individual media stream. Specifically, the stream class is where stream control, stream properties, and stream data buffers are managed [DriverStudio, 2004].

4.6 Compiling a driver

Although DriverStudio is a driver development tool, it still requires the Windows DDK build utility to compile drivers. The DDK is required to be used by all kernel-mode drivers as it builds the driver source code with kernel-specific headers, libraries, compilers and linkers. The build utility can be run from the Visual Studio IDE or a command prompt. The framework generates the `SOURCES` and `MAKEFILE` files that are used by the build utility when a new driver project is created, leaving customisation up to the developers. These files contain the location of the various source files, the libraries to be linked and the intended output used in the compilation. Building a driver is a critical event, as it has to be optimised to the platform (for example, x86 32-bit or IA64 64-bit builds) it is intended for. Thus the usage of the correct build options (of the correct DDK version) for the target platform is necessary for the driver to compile and function correctly.

4.7 Versioning Issues

It is important to note that the DriverStudio framework is a development tool that encapsulates the WDM architecture. It makes system calls and uses system objects, though the framework exposes a high level of abstraction. The classes of the framework are based on the WDM and are developed to use the recommended techniques. Microsoft allows developers to create drivers for their operating systems by providing the Windows Driver Development Kit (DDK), and it is the DDK that is responsible for the building of the driver. The DDK is versioned by release and operating system, and documents the WDM, providing a design guide for developing drivers. Through each subsequent update to the operating system (for example, the Windows XP Service Pack 2) and the DDK, a change in the WDM or Microsoft core drivers may occur. This change has to be reflected in the DriverStudio framework, and it may be that different versions of DriverWorks (the specific module for driver development) will be incompatible. Consequently, code written for a driver using an earlier version of DriverStudio and the Windows DDK may not compile with later versions unless modification of the code occurs.

4.8 Chapter Summary

This chapter has explained the usefulness of using a framework like DriverStudio in driver development. It is also fundamental to understanding the analysis of the mLAN driver. The relevant concepts have been covered to allow the next chapter to begin explaining the analysis of the driver. It is evident from the previous two chapters that the relationship between the WDM and the DriverStudio framework, as the encapsulation of a complex model for drivers is simplified using class libraries.

5. Object-oriented model of the mLAN driver

This chapter introduces the model of the mLAN driver, explaining the usage of the classes and how they fit into the DriverStudio framework. Firstly a brief description of the methodology of analysis used in abstracting the object model and class interactions is presented. Following that is an overview of the class model structure, the classes involved and how they relate to each other. This takes a conceptual look at the underlying driver technology used by the mLAN driver.

5.1 Methodology of analysis

The purpose of the previous DriverStudio chapter is to provide an understanding of the framework used to create the mLAN driver. DriverStudio contains different modules, providing development tools, extensive testing utilities and a powerful debugger called SoftIce. SoftIce requires that building the driver is done using the DDK “checked build” option, which includes debug information into the build. SoftIce is able to convert that debug information and the source code into a symbol table that allows the debugger to step through each line of code, examining the value of every variable and the memory allocation.

The object model can be extrapolated from the source code by examining the classes used for the driver and devices, and relating their usage to the DriverStudio framework. This does not present a challenge, but the main focus of the project, is in essence to model the interaction between the classes, and in particular the servicing of I/O requests. SoftIce becomes a valuable tool in performing analysis of the driver source code, allowing the extraction of sequence diagrams through quick and accurate analysis. The usage of data structures and objects will become clearer through tracking IOCTL requests from an application to the driver code. The alternative is manually reading the code to document the driver.

5.2 Overview of the Class Structure

The object-oriented model of the mLAN driver is represented in the class diagram below. This is the mLAN driver, and in its compiled form is called “mlanbus.sys”. From a high level, the driver can be viewed as a kernel component, since it resides in system addressable memory space and operates in the kernel-mode. It can only be accessed by user-mode applications via the WIN32 API, which exposes driver functionality through IOCTL codes.

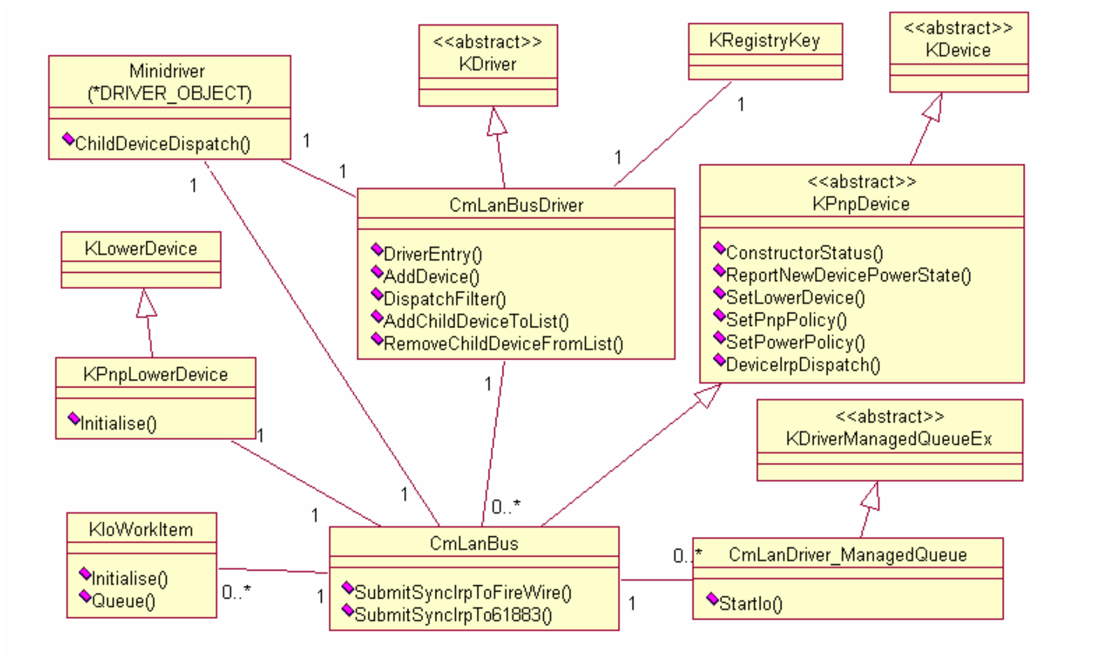


Figure 5.1: Object Model of mLAN Driver

The class that is conceptually seen by the system as the driver is the CmLanBusDriver class, which inherits from the KDriver class of the framework. This is the class that contains the entry points into the driver code. These will be covered in a later section with the aid of sequence diagrams. Every time the system detects a new device under the control of the driver, it calls the *AddDevice* routine of the driver code. The driver then instantiates a new device object of type CmLanBus, passing it the pointer to the system-created device object obtained from the *AddDevice* routine. Upon the creation of the device class object, it needs to initialise its lower device object of type

KPnpLowerDevice. This represents the driver technology on which the mLAN driver sits. The actual mechanics of this is handled by the framework and the I/O manager.

The Minidriver object is a data structure representing another driver for making use of Kernel Streaming services, modelling the DriverStudio KS architecture. CmLanDriver_ManagedQueue is the class that makes up a driver-managed IRP queue. The rest of the classes (KIoWorkItem and KRegistryKey) are included as indication of how the mLAN driver makes use of the framework's utility classes.

5.2.1 Entry points

There are three entry points into the driver code, these being the *DriverEntry, AddDevice* and *DispatchFilter* routines. The *DriverEntry* routine is called when the driver is first loaded, giving the driver a chance to initialise the driver state. This is depicted below in the sequence diagram. Driver state variables are retrieved from the system registry, stored from the last time the driver was loaded. The framework's dispatch filter mechanism is enabled.

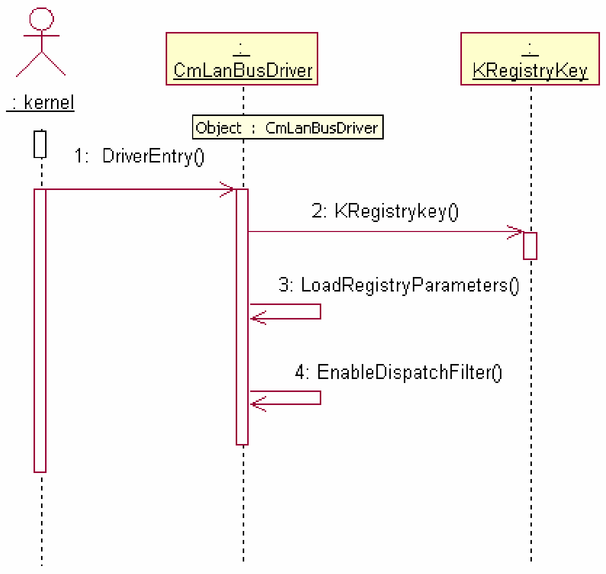


Figure 5.2: The DriverEntry Routine

The next entry point is the *AddDevice* routine, which is called in response to a new device being detected. This creates and initialises a device class object, which in turn initialises its attached lower-device object. During initialisation of the device class object, the relative Power and PnP policies are set. These policies govern the way the device reacts to the various Power and PnP IRPs sent to it. For instance, this will allow or disallow a device being put into sleep mode. Finally, should there be no problems, the driver, via the framework, alerts the PnP manager of the power state of the new device. Below is the sequence diagram for the routine.

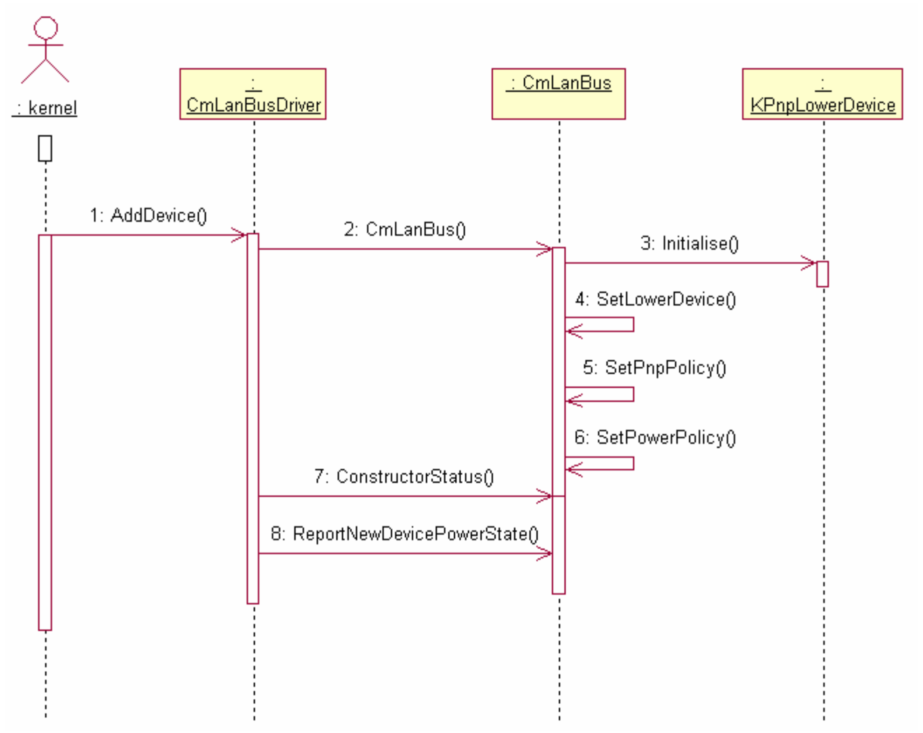


Figure 5.3: The AddDevice Routine

For the last entry point, as seen in the sequence diagram below, the DriverStudio framework neatly encapsulates the dispatching mechanism required by other WDM drivers by routing IRPs to a single routine. The *DispatchFilter* routine simply checks whether the IRP is targeted to a device under control of the driver, or a child minidriver object responsible for a stream. It is up to the target device to handle the IRP correctly. The minidriver object receives an IRP from the Streaming driver it represents in order to communicate with the mLAN driver. This is covered in the next chapter. An IRP for a device of the driver is dispatched to the device class to be handled there. The framework is aware of which IRP handler routine should be called by examining the IRP major function code.

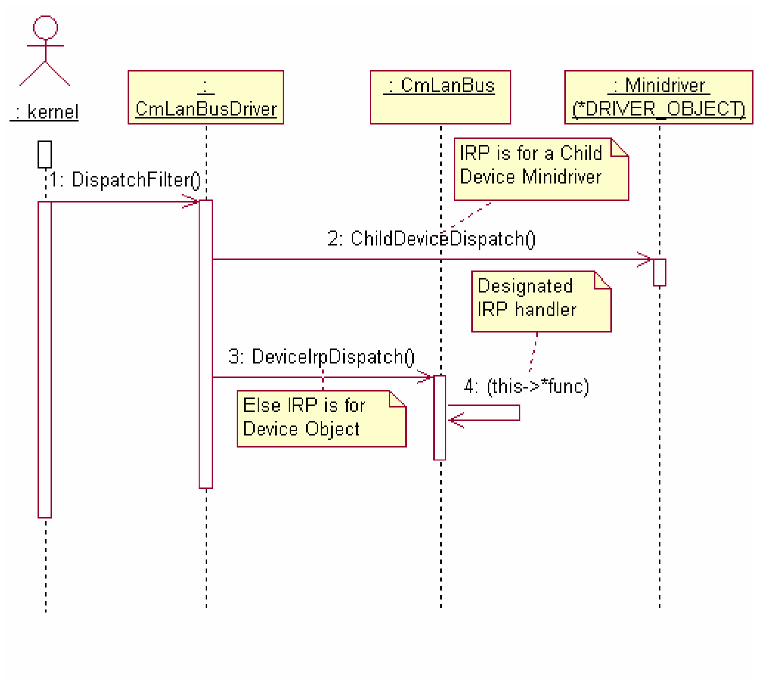


Figure 5.4: The DispatchFilter Routine

5.2.2 Kernel Streaming Minidriver

As mentioned above, the mLAN driver makes use of a Kernel Streaming Minidriver to control the isochronous streaming between the PC and a node on the Firewire bus. It is a driver created with DriverStudio based on the Windows Kernel Streaming architecture. Below is a class diagram depicting the mLAN specific implementation of the framework's KS architecture. The member functions and variables, for the most part, are left out because they have no direct bearing on this discussion.

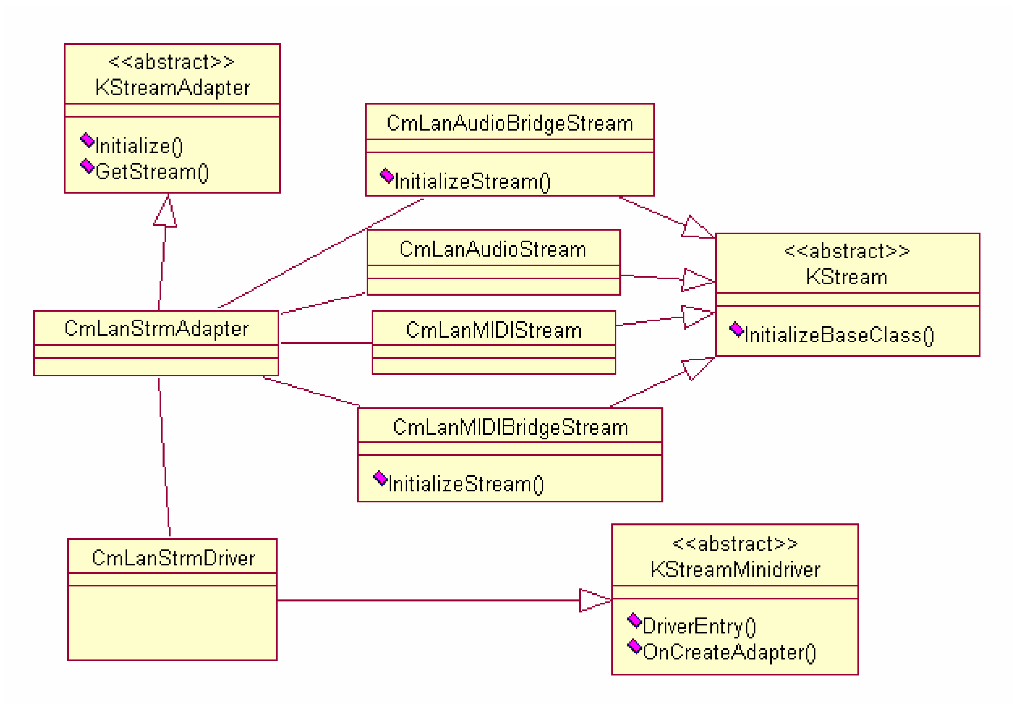


Figure 5.5: The mLAN Kernel Streaming Minidriver Class Diagram

The CmLanStrmDriver class is the inherited KStreamMinidriver class used to provide the entry points of the driver. The framework responds to the I/O manager's call to create a new device by calling *OnCreateAdapter*, which creates the CmLanStrmAdapter object and initialises it. The CmLanStrmAdapter object contains objects representing all possible types of streams that it can handle, and will instantiate the required stream class. The stream child classes are all inherited from the same parent class, requiring upon creation to call *InitializeBaseClass* to set up the stream parameters. These stream classes represent the MIDI and audio data streams that exist in mLAN's isochronous streaming. Unfortunately Microsoft's IEEE 1394 driver does not support the bridging of multiple busses due to limitations in the representation of the bus number of the driver. As a result, the bridge streams are not fully implemented, merely providing a skeleton for future work when support for multiple busses is implemented.

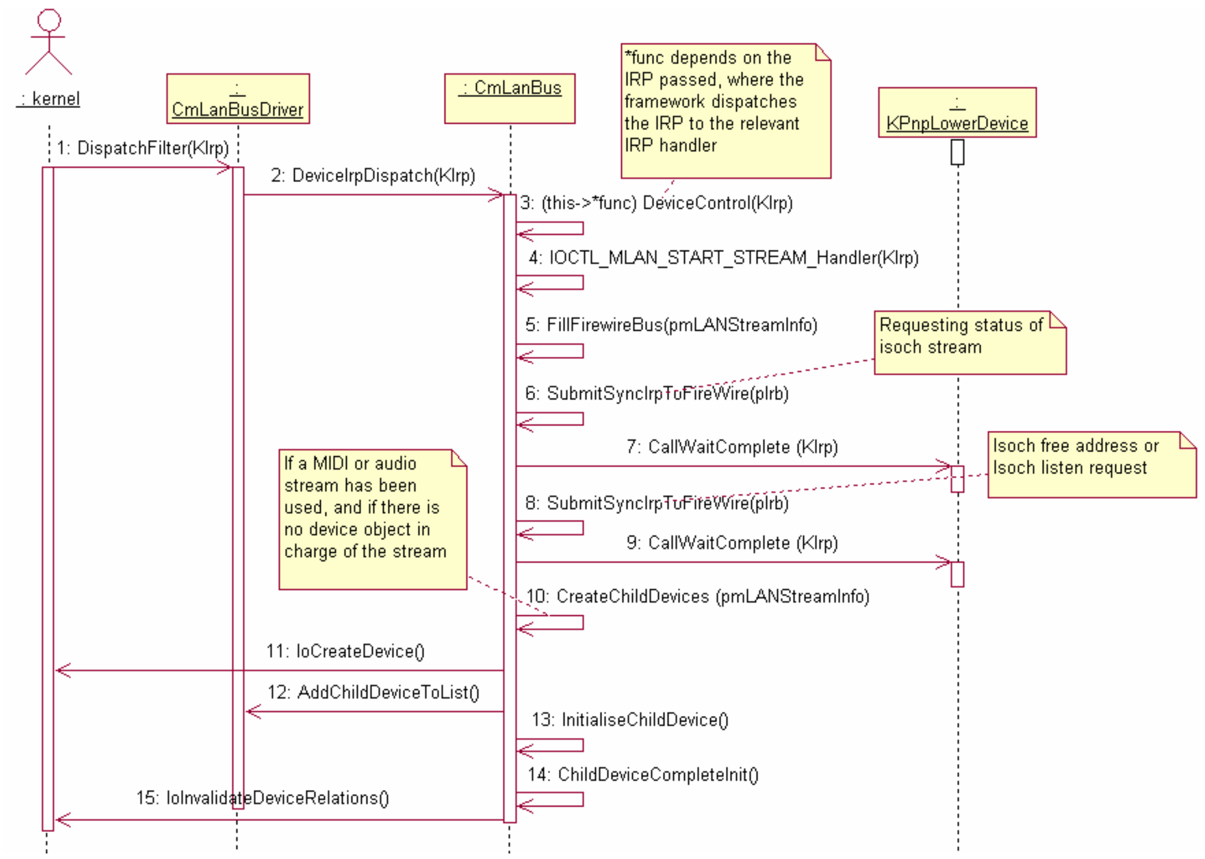

: kernel
: CmLanBusDriver
: CmLanBus
: KPnpLowerDevice
*func depends on the IRP passed, where the framework dispatches the IRP to the relevant IRP handler
1: DispatchFilter(KIrp)
2: DeviceIrpDispatch(KIrp)
3: (this->*func) DeviceControl(KIrp)
4: IOCTL_MLAN_START_STREAM_Handler(KIrp)
5: FillFirewireBus(pmLANStreamInfo)
Requesting status of isoch stream
6: SubmitSyncIrpToFireWire(pIrb)
7: CallWaitComplete (KIrp)
Isoch free address or Isoch listen request
If a MIDI or audio stream has been used, and if there is no device object in charge of the stream
8: SubmitSyncIrpToFireWire(pIrb)
9: CallWaitComplete (KIrp)
10: CreateChildDevices (pmLANStreamInfo)
11: IoCreateDevice()
12: AddChildDeviceToList()
13: InitialiseChildDevice()
14: ChildDeviceCompleteInit()
15: IoInvalidateDeviceRelations()

Figure 5.6: Sequence Diagram for Creating Minidriver

The diagram above models the interaction between the mLAN driver and the KS minidriver in response to an application based request to begin streaming. The points before number four in the sequence diagram is generic IOCTL processing, while points five to nine represent the request for bus resources. In those requests, parameters for the stream are initialised. Points ten to fifteen represent the steps taken to create the minidriver object. Essentially, the minidriver is called to be created by the system call of *IoCreateDevice*, which is required to control an isochronous stream. This is created by the device object and stored in driver object for dispatching purposes. The minidriver object is removed when a PnP IRP updating the state of the device to be “removed” is received or the stream it is responsible for ceases streaming.

5.2.3 IOCTL Dispatching

IOCTL codes are specified when using the IRP_MJ_DEVICE_CONTROL major function code of the IRP. As a result, the driver can pick out the IOCTL by having a handler routine for the IRP that switches the IOCTL code and calls the appropriate IOCTL handler routine.

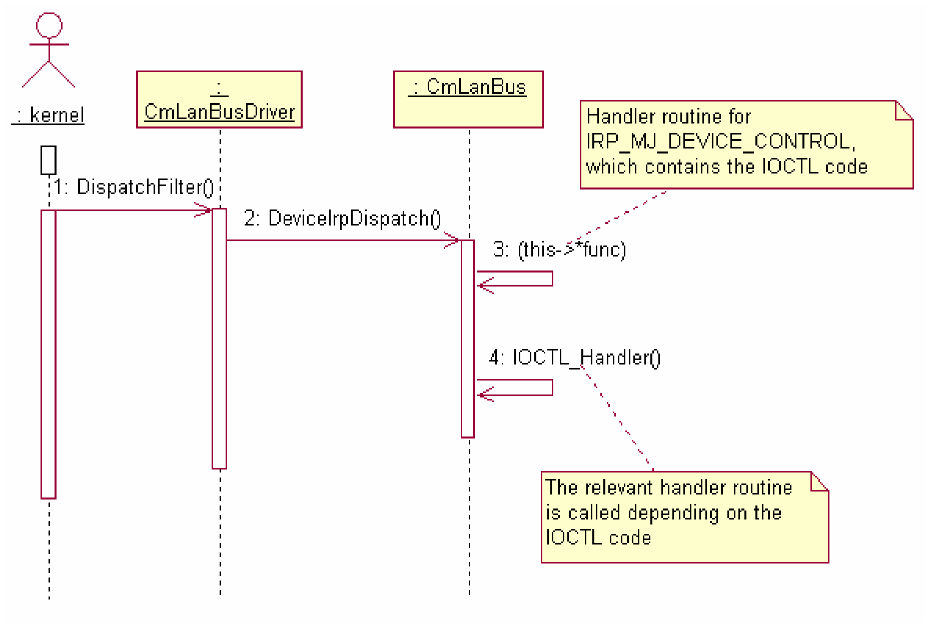

: kernel
: CmLanBusDriver
: CmLanBus
Handler routine for IRP_MJ_DEVICE_CONTROL, which contains the IOCTL code
1: DispatchFilter()
2: DeviceIrpDispatch()
3: (this->*func)
4: IOCTL_Handler()
The relevant handler routine is called depending on the IOCTL code

Figure 5.7: Sequence Diagram of IOCTL Dispatching

This is a generic IOCTL dispatching mechanism which is relevant in understanding the communication aspects of a driver. There is no need to explicitly return information, as the input and output buffers are contained in the IRP, which the I/O manager returns to the application.

5.3 Chapter Summary

After discussing the structure of the driver, relating it to the WDM and the DriverStudio framework, it is now appropriate to move onto the chapter concerning the analysis and the documenting of the IOCTL codes.

6. IOCTL Analysis and API design

This chapter represents the work done to document the IOCTL codes and the respective handler routines of the mLAN driver. The purpose of this is to provide information concerning the IOCTLs in order to propagate understanding of how the IOCTLs work and their intended usage. Firstly, an overview of the usage and form of IOCTLs in regard to the mLAN driver is given. Due to the number of IOCTL codes, it is prudent to group them according to their underlying function. Each group is identified and explained. Following that is a brief section describing the issues surrounding the analysis of the driver.

6.1 mLAN driver IOCTL codes

The mLAN driver implements IOCTL function codes in the custom range from 2048 to 4095. The values from 0 to 2048 are reserved for use by Microsoft [MSDN, 2005]. This allows independent hardware vendors and other types of companies that write drivers to provide specialised IOCTL for their drivers, and removes any overlapping that might exist for devices of a similar type. The enumeration of the IOCTL codes for the mLAN driver can be found in “mlbusdrvioctl.h”. Below is an example of a macro for IOCTLs, defining the name of the control code.

```
#define    IOCTL_MLAN_ALLOCATE_STREAM    CTL_CODE(FILE_DEVICE_UNKNOWN,
0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define  IOCTL_MLAN_ASYNC_LOCK  CTL_CODE(FILE_DEVICE_UNKNOWN,  0x906,
METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Figure 6.1: Code excerpt from “mlbusdrvioctl.h”

The first parameter specifies the type of device the IOCTL is targeted at. Microsoft has a list of device types predefined by the operating system, of which each device type has its own associated set of function codes. The usage of `FILE_DEVICE_UNKOWN` in this case is because a mLAN device must make use of custom IOCTL function codes and no device similar to the mLAN device exists. The second parameter is the function code of the IOCTL, which gives the action of the IOCTL. This is important, as it is used in differentiating between different IOCTL codes for the same device type. The third parameter specifies how buffers are passed for I/O and file system controls. Input and output buffers are used to store and access the data contained in the IRP. To recap, IRPs have fields for the input and output buffers used in the processing of the IOCTL and the data that is returned to the application. This is the IRP that the IO manager creates to communicate with the driver. The fourth parameter describes the access level that the IOCTL handler routine has over the IRP.

The mLAN driver provides and implements 63 different IOCTL codes and their respective handler routines. Due to space limitations in this write up, the IOCTL codes are too lengthy to be dealt with individually and will be grouped by their core functionality. For example, all 61883 related IOCTLS will be grouped together. This is not a fair representation of the IOCTL codes as each IOCTL handles a specific action. For the documentation of each specific IOCTL, please refer to the Appendix.

There are five groups into which each IOCTL code can be categorised. For each of these groups a brief discussion describing the intentions of the IOCTL codes will be given, followed by detailed documentation of one IOCTL code from that group.

6.1.1 IEEE 1394 based mLAN IOCTL codes

These IOCTLs are implementations of IEEE 1394 class calls based on the IEEE 1394 bus driver. They are characterised by their function and communication with the IEEE 1394 bus driver. Below is a table containing the IOCTL codes in this group with a brief description.

IOCTL code	Description
`IOCTL_MLAN_ADDRESS_RANGE_NOTIFY`	Returns the node address of the source of an I/O operation on the specified address range
`IOCTL_MLAN_SET_ADDRESS`	Sets the address of the allocated address range
`IOCTL_MLAN_GET_ADDRESS`	Retrieves the address of the node of the allocated address range
`IOCTL_MLAN_FREE_ADDRESS_RANGE`	Frees a previously allocated address range
`IOCTL_MLAN_ASYNC_LOCK`	Performs a locked asynchronous operation on the indicated

	destination address
`IOCTL_MLAN_ASYNC_READ`	Performs an asynchronous read operation from the source address
`IOCTL_MLAN_ASYNC_WRITE`	Performs an asynchronous write operation on the indicated destination address
`IOCTL_MLAN_BUS_RESET_NOTIFY`	Waits for a bus reset to occur
`IOCTL_MLAN_BUS_RESET`	Initiates a bus reset
`IOCTL_MLAN_GET_ADDR_FROM_DEVICE_OBJECT`	Not implemented
`IOCTL_MLAN_GET_LOCAL_HOST_INFO`	Returns local host information
`IOCTL_MLAN_GET_SPEED_TOPOLOGY_MAPS`	Obsolete
`IOCTL_MLAN_SEND_PHY_CONFIG_PACKET`	Sends PHY config packet request to the 1394 bus driver
`IOCTL_MLAN_GET_LOCAL_NODE_ADDRESS`	Returns the local node address
`IOCTL_MLAN_GET_CHANNELS_AVAILABLE`	Returns the bandwidth and channels currently available
`IOCTL_MLAN_ALLOCATE_CHANNEL`	Allocates the specified channel
`IOCTL_MLAN_RELEASE_CHANNEL`	Releases the allocated channel
`IOCTL_MLAN_GET_BANDWIDTH_AVAILABLE`	Returns the bandwidth and channels currently available
`IOCTL_MLAN_ALLOCATE_BANDWIDTH`	Allocates the specified bandwidth
`IOCTL_MLAN_RELEASE_BANDWIDTH`	Releases the allocated bandwidth

The mLAN driver must do additional processing in the IOCTL handler routine, rather than submit the request straight to the underlying IEEE 1394 bus driver. There are some IOCTL calls that do not communicate with the Firewire driver, but instead act on or use IEEE 1394 related data structures. It is through this additional processing that the inherent functionality of the mLAN driver becomes apparent, as well as the layered nature of the WDM architecture. In order to communicate with the Firewire driver, an IRB has to be constructed with a function code specified, the IRB packaged into an IRP, and submitted to the Firewire driver. The actual mechanism uses a pointer to an IRB that is passed to the IRP, allowing (upon completion of the IRP) the response data to be extracted from the IRB. The function code dictates the data structure to be used, as it is forms a union with the IRB. Generally the packets sent to the bus tend to be transmitted asynchronously, as they are control packets.

The usage of this group of IOCTLS can be further categorised into sub groups that either allocate or de-allocate IEEE 1394 resources, modify or return driver state information, initiate an asynchronous request, or setup a notification of some event. In order to effectively use the Firewire bus, certain configuration and driver state information has to be known by the application. This will be used in subsequent requests of Firewire resources. For instance, the application has to know the bus speed in order to request bandwidth resources. A resource, in this case, is something controlled by the Firewire driver. For the mLAN driver to make use of the resource, an appropriate request has to be sent to the Firewire driver. If there are enough free resources to allocate, a handle to the requested resource is returned. If the resource is no longer used, it should be de-allocated and be returned to the pool of free resources. An address range, channel number, and bandwidth are resources that are allocated to the mLAN driver by the IEEE 1394 bus driver. These need to be allocated before any asynchronous requests can be made. There are times when an application needs to be notified of an event that takes place independent of any determinable factors. I/O operations on an allocated address range or a bus reset are two instances of events that an application would want to be aware of. A request for notification will return a pointer to an event object, which can be polled to check if the event has transpired. It is advisable to setup a waiting thread to avoid locking up the application.

An in-depth explanation of an IOCTL code handler routine from the IEEE 1394 group will serve as an example, to give a template for those contained in the appendix. The code, IOCTL_MLAN_ALLOCATE_ADDRESS_RANGE, is chosen as an example. Given below is the extract of the IOCTL explanation from the appendix. First is the specification of the input buffer, followed by the output buffer. It depends on the IOCTL as to what data structures or data primitives are used as input and output. In this case it is a pointer to an `ALLOCATE_ADDRESS_RANGE` struct.
Following that is a textual description of the IOCTL, giving a brief overview of the intended action to be performed. Below the description is a diagram representing a hybrid between a Ward and Mellor Structured Data Diagram [Ward, et al, 1985], and Larry Constantine's Data Flow Diagram. It is hybridised because neither method neatly encapsulates the IOCTL handler, but rather taking elements from both methods and combining them yields a clearer picture. The entire diagram can be viewed as process instead of each full circle, as in the Ward and Mellor approach. The dotted circle still represents a trigger for the process, responding to the IRP dispatch routine. The parallel lines represent a data store, common to both methods of modelling. Each full circle represents a sub-process that acts on a data store. The dotted lines represent the flow of the trigger, while the solid lines represent the flow of data and control.

After the structured data diagram is the abovementioned data structure used as input and output in the IOCTL call. Note that the data structures are documented in a separate section of the appendix.

Input Buffer:

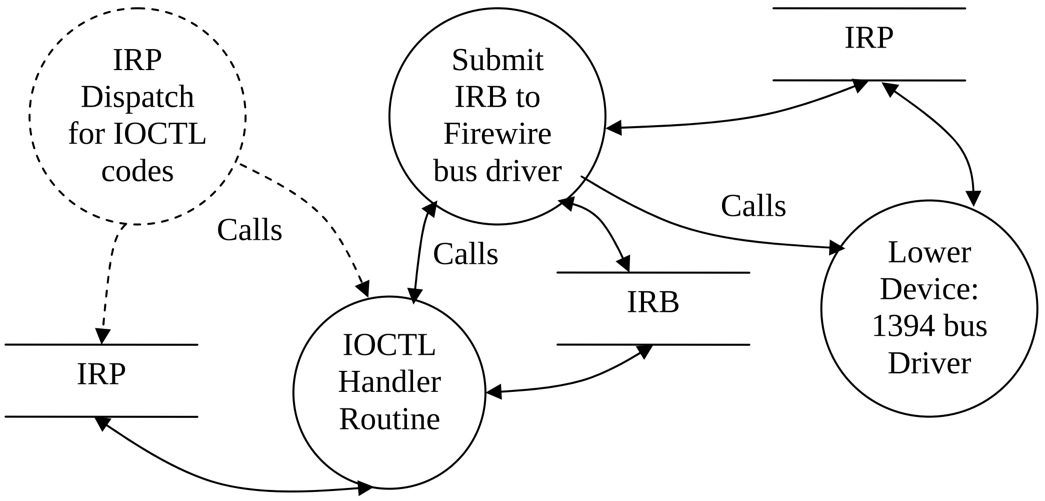
A Pointer to an `ALLOCATE_ADDRESS_RANGE` struct.

Output Buffer:

A pointer to the same `ALLOCATE_ADDRESS_RANGE` struct.

Textual Description:

The struct specifies the address range that will be allocated in Host Controller address space by the IEEE 1394 bus driver. An address range has to be allocated before the host controller can respond to any communication from that node. The mLAN driver sends an IRB containing the address range to the 1394 driver. The IRB function code is that of `REQUEST_ALLOCATE_ADDRESS_RANGE`. A handle to the address range is returned.


IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

```
typedef struct _ALLOCATE_ADDRESS_RANGE {
    IN ULONG            fulAllocateFlags;
    IN ULONG            fulFlags;
    IN ULONG            nLength;
    IN ULONG            MaxSegmentSize;
    IN ULONG            fulAccessType;
    IN ULONG            fulNotificationOptions;
    OUT ADDRESS_OFFSET  Required1394Offset;
    OUT HANDLE          hAddressRange;
    IN UCHAR            Data[1];
} ALLOCATE_ADDRESS_RANGE, *PALLOCATE_ADDRESS_RANGE;
```

In this particular IOCTL call, the UCHAR Data[1] field of the data structure specifies the beginning of the address range to be used. The ULONG nLength field specifies how long the address range will be. In the IOCTL handler routine a buffer in system memory is created to hold the address range. This is used in the IRB union struct. The rest of the input fields are there to specify parameters to be used in the IRB. These relate to memory management, completion notification and broadcast features of the asynchronous request. The Firewire driver returns a handle to the allocated address range and an ADDRESS_OFFSET struct that contains the IEEE 1394 node address. The information is available from the struct that was used as input to the IOCTL.

6.1.2 Stream based mLAN IOCTL codes

The only difference between this group of IOCTLs and the previous is that these IOCTLs deal with the isochronous nature of the mLAN system. As before, some of these IOCTL handler routines do not communicate with the IEEE 1394 driver, they are grouped here by the manner in which they interact with stream related structs used by the mLAN driver. The main focus of this group is the management of stream data structures used to represent a stream, and the sequences that make up a stream. A stream is viewed as another IEEE 1394 resource. Like any other resource, it has to be allocated before it can be used, and de-allocated when it is not in use. An allocated stream exists in two states, started and stopped. There are IOCTLs that handle these actions, as well as return stream specific information. The table below contains the brief description of the IOCTLs contained in this group.

IOCTL code	Description
`IOCTL_MLAN_START_STREAM`	Starts the allocated stream
`IOCTL_MLAN_STOP_STREAM`	Stops the allocated stream
`IOCTL_MLAN_GET_STREAM_INFO`	Returns information of the stream
`IOCTL_MLAN_FREE_STREAM`	Frees the allocated stream
`IOCTL_MLAN_GET_DRIVER_VERSION`	Returns the driver version
`IOCTL_MLAN_CONNECT_SEQUENCES_TO_DEVICES`	Creates the connection map of sequences
`IOCTL_MLAN_SET_SYT_SOURCE`	Sets the bus master field

This group’s IOCTL showcase is the IOCTL_MLAN_ALLOCATE_STREAM handler routine. Following the same format as above, the IOCTL makes use of the

`MLAN_ISOCH_PARAM` struct. The mLAN driver communicates with the IEEE 1394 bus driver several times to request Firewire resources. These are requests for allocating a channel, bandwidth, and a resource handle for transactions. The resource handle is required for requesting and attaching buffers to the stream. In terms of the input required for the IOCTL, two mLAN structs form fields of the input struct used to pass information for use in the requests.

Input Buffer:

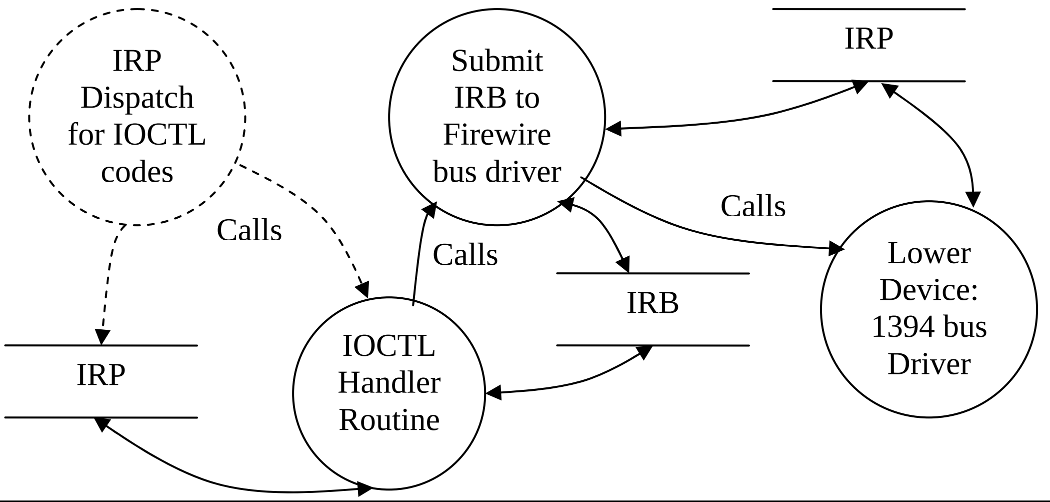
The input buffer uses a `MLAN_ISOCH_PARAM` struct, which specifies the type of the stream to be used.

Output Buffer:

The same struct is returned, specifying the allocated stream information.

Textual Description:

The request responds to the need to allocate an isochronous stream resource specified by the input struct. It goes through several steps to allocate a stream resource. It creates the specified stream type, and gets a channel from the underlying 1394 bus driver. The IRB function codes submitted to the Firewire driver are as follows: `REQUEST_ISOCH_ALLOCATE_BANDWIDTH`, `REQUEST_ISOCH_ALLOCATE_CHANNEL`, `REQUEST_ISOCH_ALLOCATE_RESOURCES`, `REQUEST_ISOCH_ATTACH_BUFFERS`. Inside the struct, a pointer is provided to a `MLAN_STREAM_STATUS` struct, which contains the information about the allocated stream.


IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

```
typedef struct _MLAN_ISOCH_PARAM {
      OUT HANDLE                   hStream;
      OUT MLAN_STREAM_STATUS       StreamStatus;
      OUT ULONG                    ErrorCode;
      OUT ULONG                 aBitmapClient[MLAN_MAX_NUM_SEQUENCES
                                / ULONG_BITSIZE];
      IN MLAN_STREAM_CONFIG      mLANStreamConfig;
      IN MLAN_DATA_FIELD_CONFIG  mLANDataFieldConfig;
} MLAN_ISOCH_PARAM, *PMLAN_ISOCH_PARAM;
```

The `MLAN_STREAM_CONFIG` struct describes the required stream configuration, in particular the bus speed, stream type, channel, and buffer related information. The `MLAN_DATA_FIELD_CONFIG` struct describes the intended configuration of the stream data, and will be returned with extra information concerning the individual sequences. A third struct, `MLAN_STREAM_STATUS`, returns information about the allocated stream and individual sequence status. The primary field in the `MLAN_ISOCH_PARAM` struct is the handle to the allocated stream. The stream handle is required for any subsequent stream related operation.

6.1.3 61883 based mLAN IOCTL codes

The IEC 61883 based group of IOCTL codes are characterised by the underlying use of the 61883 protocol driver and the 61883 specific structs that they act on. Similar to the use of the IEEE 1394 bus driver, the IOCTL handler routines utilises the functionality of the 61883 protocol driver and have to build an AV_61883_Request, specifying the function code and setting the correct data for input. The request is submitted to a process responsible for building a suitable IRP and sending it to the 61883 protocol driver. It is possible to further divide this group into three sub groups featuring:

- Notification requests
- Function Control Protocol (FCP) packet requests
- Plug related requests

The notification sub-group sets up callback routines to wait for the specified event to occur.

The Function Control Protocol sub-group allows the “send” and “get” of request and response FCP packets to and from the bus. An FCP packet is an AV/C command or response encapsulated in a FCP frame. The FCP packet is then received from or transmitted to a device on the bus by the IEEE 1394 drive via the IEC 61883 protocol driver.

The last sub-group has to do with plug-related requests such as creating and deleting a plug, connecting and disconnecting plugs, and setting and getting plug state information. The IOCTLs are briefly described below.

IOCTL codes	Description
`IOCTL_MLAN_61883_GET_PLUG_HANDLE`	Returns a handle to the specified plug
`IOCTL_MLAN_61883_CREATE_PLUG`	Returns a handle to a newly created plug
`IOCTL_MLAN_61883_PLUG_NOTIFY`	Returns an event handle that indicates when an operation is performed on the plug
`IOCTL_MLAN_61883_DELETE_PLUG`	Deletes a plug
`IOCTL_MLAN_61883_DISCONNECT_PLUG`	Disconnects the specified connection
`IOCTL_MLAN_61883_GET_FCP_REQUEST`	Returns the next FCP request packet from the queue
`IOCTL_MLAN_61883_GET_FCP_RESPONSE`	Returns the next FCP response packet from the queue
`IOCTL_MLAN_61883_GET_PLUG_STATE`	Returns the state of the specified plug
`IOCTL_MLAN_61883_SEND_FCP_REQUEST`	Sends FCP request to the device
`IOCTL_MLAN_61883_SET_FCP_NOTIFY`	Registers a client driver notification of FCP requests or responses
`IOCTL_MLAN_61883_SEND_FCP_RESPONSE`	Sends FCP response to the device
`IOCTL_MLAN_61883_SET_PLUG`	Changes transmission settings for a plug

An example from this group is the IOCTL_MLAN_61883_CONNECT_PLUG handler routine. The input buffer receives a pointer to a `CMP_CONNECT` struct, with the same struct being returned. Of the input structs used for the various IOCTL calls, those that make use of the 61883 protocol driver mirror existing 61883 data structures. The `CMP_CONNECT` struct is no exception and provides handles to the input and output plugs that are intended to be connected. Two other 61883 structs are used to specify the connection type and data format to be used in the streams. These are the `CMP_CONNECT_TYPE` and the `CIP_DATA_FORMAT` structs. The output of the IOCTL call is a handle to a plug connection.

Input Buffer:

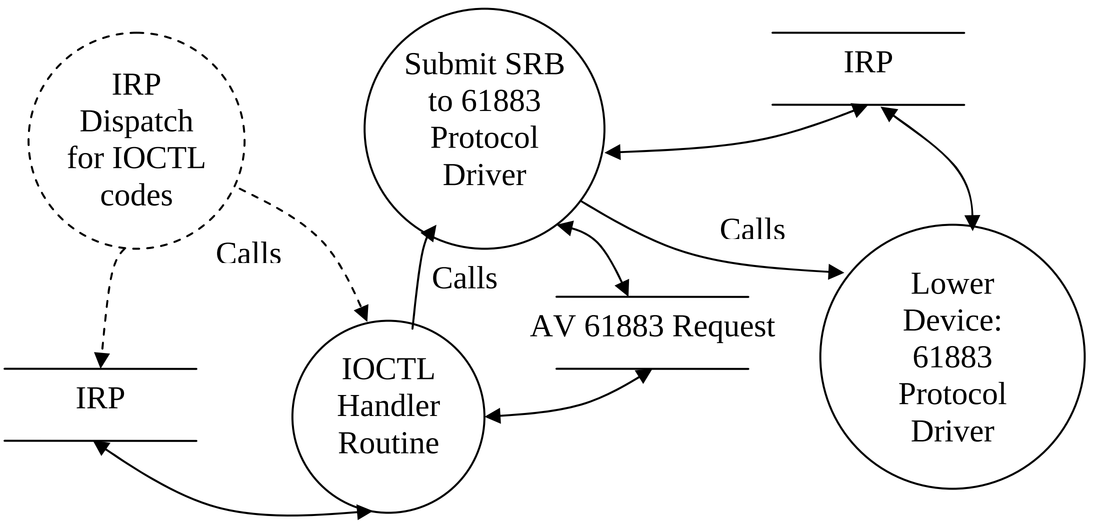
The input buffer takes a `CMP_CONNECT` struct.

Output Buffer:

The same struct is returned.

Textual Description:

The request is forwarded to the underlying 61883 protocol driver, using the handles of plugs contained in the input struct to specify which plugs to connect. A function code of `Av61883_Connect` is used. A handle to the connection is returned.


IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
IRP
IOCTL Handler Routine
Lower Device: 61883 Protocol Driver

```
typedef struct _CMP_CONNECT {

    IN HANDLE                hOutputPlug;
    IN HANDLE                hInputPlug;
    IN CMP_CONNECT_TYPE      Type;
    IN CIP_DATA_FORMAT       Format;
    OUT HANDLE               hConnect;
} CMP_CONNECT, *PCMP_CONNECT;
```

6.1.4 ASIO based mLAN IOCTL codes

The next group of IOCTLs, shown below, is the ASIO based calls, which is a mLAN implementation of ASIO functionality.

IOCTL codes	Description
`IOCTL_MLAN_ASIO_INIT`	Returns the current state information of the ASIO driver and ASIO streams, and initialises the driver
`IOCTL_MLAN_ASIO_FREE`	Frees up ASIO allocated buffers
`IOCTL_MLAN_ASIO_ALLOC`	Allocates ASIO buffers and resources
`IOCTL_MLAN_ASIO_STOP`	Stops the ASIO driver
`IOCTL_MLAN_ASIO_CLEANUP`	Removes ASIO related objects
`IOCTL_MLAN_ASIO_INFO`	Returns the current state information of the ASIO driver and ASIO streams
`IOCTL_MLAN_ASIO_RESET`	Resets the ASIO driver

ASIO (Audio Streaming Input Output) is an API that provides the basis for an audio streaming driver to make use of multiple channels, overcoming limitations of Microsoft supplied audio streaming drivers. The classes can be overridden such that the needed functionality can be extracted. The mLAN driver uses the ASIO classes to create a client ASIO driver for use by applications requiring audio recording and playback capabilities. The following diagram shows the finite state machine diagram of the driver, illustrating the operation states and the operation that changes the state.

There are four states: *Loaded, Initialised, Prepared* and *Running*. The *Loading* state implies the driver code is accessible by the application or ASIO client. The operating state is *Initialised* when the driver is allocated by the application and can accept inquiries. The *Prepared* state is attained when buffers are allocated and is ready to move to the *Running* state. The *Running* state signifies that the hardware is active and streaming is taking place.

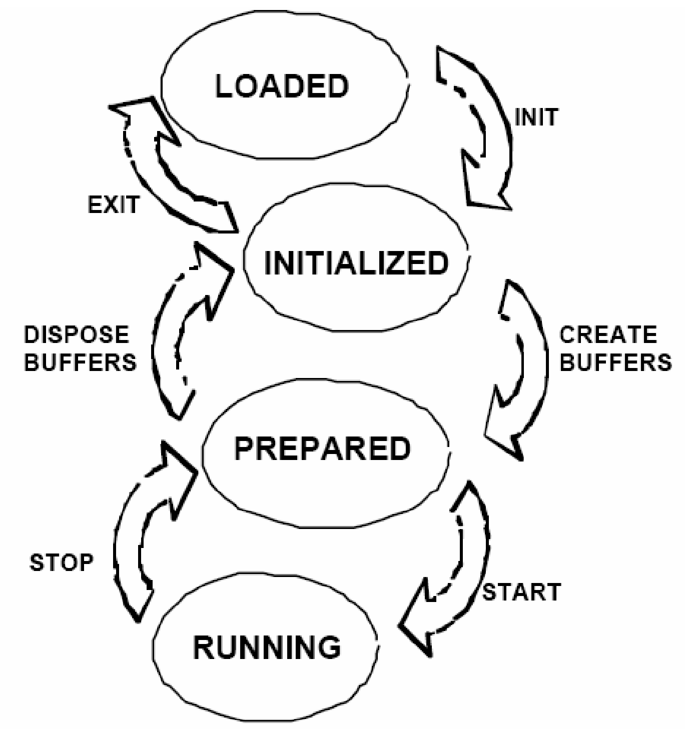


Figure 6.2: ASIO Finite State Diagram [ASIO, 2005]

The IOCTL calls mirror the operations that change states of the ASIO driver. There is an INIT IOCTL call that initialises the ASIO driver with mLAN specific information, and returns event objects to be used by the ASIO client. The ALLOC IOCTL allocates buffers to be used by the application, hardware and driver, while the FREE IOCTL releases those buffers. The CLEANUP IOCTL returns the ASIO driver to the *Loaded* state. There are other IOCTL codes that specify an ASIO reset, and retrieve information about the ASIO driver. The IOCTL that will serve as this section's example is IOCTL_MLAN_ASIO_START. The input struct is a single field specifying the result of the operation of the call. There is only ever one ASIO driver to be used by an application or ASIO client, though each stream can be associated with a set of ASIO buffers. This IOCTL call clears the buffers before recording begins. The ASIO driver is synchronised with the mLAN driver by pointing both drivers to same position in the buffer. The ASIO driver state is then set to *Running*.

Input Buffer:

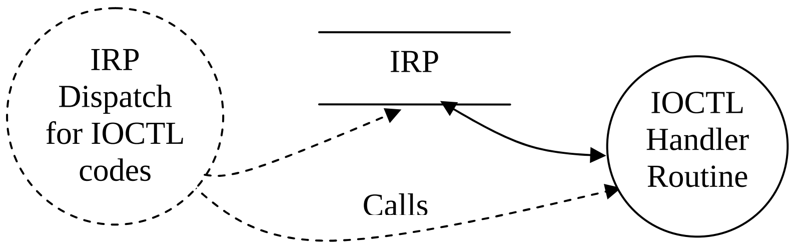
The input buffer takes a `MLAN_ASIO_COMMAND_PARAM` struct.

Output Buffer:

The same struct is returned.

Textual Description:

The request starts ASIO processing, synchronising with the ASIO driver.



```
typedef struct _MLAN_ASIO_COMMAND_PARAM {
      ULONG errorCode;
} MLAN_ASIO_COMMAND_PARAM, *PMLAN_ASIO_COMMAND_PARAM;
```

6.1.5 WDM Streaming based mLAN IOCTL codes

This group of IOCTLs is implemented for the express use by the WDM Streaming Minidrivers. The IOCTLs are used with the IRP_INTERNAL_DEVICE_CONTROL code. This is for inter-driver communication. Each minidriver object is created in order to control a stream resource. The minidriver object is created when there is a request to start streaming, and ownership is given to the mLAN bus driver. In essence, the minidriver becomes a child device of the mLAN bus driver, establishing another level in the driver stack.

When the mLAN bus driver is required to create an extra child device to control a stream, it makes a system call *IoCreateDevice.* This creates a device object in system memory and returns a pointer to the newly created device object. The creator of the device object is responsible for maintaining the object, such as deleting it when it is no longer required. The driver that is used to create the object is determined by parameters contained in the *IoCreateDevice* call. Specifying the device type will let the I/O manager know what device object to create, and will initialise the appropriate DEVICE_OBJECT fields.

After this call, the I/O manager will have a driver object and device object stored in system memory (refer to Chapter 3). It will know what driver to load by the type specified in the creation call and by accessing the mLAN driver INF file. The INF file specifies driver related information to be stored in the registry, as is used by the PnP manager when a driver is first loaded. The INF file also specifies the drivers to be used for each device the driver is responsible for. In this way, the mLAN driver publishes its usage of the WDM Stream minidrivers. An extract from the INF file is shown below. The *Version* section specifies the type of device and GUID of the class of device, and the version of the driver. The *mLanBus* section publishes the devices the driver is responsible for (a mLAN device and the streams).

```
[Version]
Class=MEDIA
ClassGUID={4d36e96c-e325-11ce-bfc1-08002be10318}
Provider=%mLanBus.Provider%
DriverVer=03/17/2004,1.1.53.0
CatalogFile=mLanBus.cat

[mLanBus]
%mLanBus.DeviceDesc%=mLanBus.INSTALL, V1394\MLAN_BUS

%mLanMIDIStrmDeviceDesc%=mLMIDIStrm, MLAN_BUS\COMPATIBLEMIDISTREAM
%mLanMIDIOutStrmDeviceDesc%=mLMIDIStrm, MLAN_BUS\MIDIOUTSTREAM
%mLanMIDIInStrmDeviceDesc%=mLMIDIStrm, MLAN_BUS\MIDIINSTREAM
%mLanAudioOutStrmDeviceDesc01%=mLAudioStrm,MLAN_BUS\AUDIOOUTSTREAM&ADAPTER_01
%mLanAudioOutStrmDeviceDesc02%=mLAudioStrm,MLAN_BUS\AUDIOOUTSTREAM&ADAPTER_02
%mLanAudioOutStrmDeviceDesc03%=mLAudioStrm,MLAN_BUS\AUDIOOUTSTREAM&ADAPTER_03
%mLanAudioOutStrmDeviceDesc04%=mLAudioStrm,MLAN_BUS\AUDIOOUTSTREAM&ADAPTER_04
%mLanAudioInStrmDeviceDesc01%=mLAudioStrm, MLAN_BUS\AUDIOINSTREAM&ADAPTER_01
%mLanAudioInStrmDeviceDesc02%=mLAudioStrm, MLAN_BUS\AUDIOINSTREAM&ADAPTER_02
%mLanAudioInStrmDeviceDesc03%=mLAudioStrm, MLAN_BUS\AUDIOINSTREAM&ADAPTER_03
%mLanAudioInStrmDeviceDesc04%=mLAudioStrm, MLAN_BUS\AUDIOINSTREAM&ADAPTER_04
```

Figure 6.3: Extract from mlanbus.INF

The WDM Streaming Minidriver created by Yamaha is a separate DriverStudio project. It is based on the C++ framework of classes that encapsulates the Windows Kernel Streaming Architecture (See Chapter 3 for an explanation of the KS Streaming Architecture). In terms of this implementation, the mLAN bus driver forms the framework's lower device object class for the adapter. This is to establish clear lines of communication within the driver stack. The diagram below describes the interaction of the three Kernel Streaming classes with the mLAN driver.

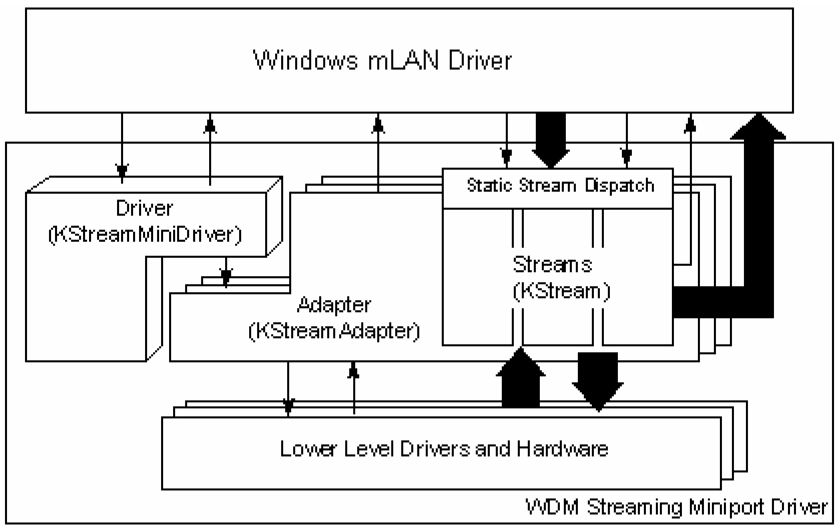

Windows mLAN Driver
Static Stream Dispatch
Driver (KStreamMiniDriver)
Streams (KStream)
Adapter (KStreamAdapter)
Lower Level Drivers and Hardware
WDM Streaming Miniport Driver

Figure 6.4: Kernel Streaming Architecture

The lower level drivers and hardware represents those drivers at the bottom of the stack closest to the HAL, responsible for receiving from and transmitting data onto the IEEE 1394 bus. Although the mLAN driver is the lower device for the WDM Streaming minidriver in the DriverStudio framework, conceptually the mLAN driver owns and is responsible for the Streaming minidriver.

6.2 Implementation Issues

As discussed in the DriverStudio chapter, versioning issues can halt progress in building a driver through incompatibility issues. The source code of the mLAN driver was released and built using version 2.7 of DriverStudio. Current analysis of the driver is done using the latest version 3.2, following several changes to the WDM. As a result, a fundamental change in the underlying functioning of at least one class (relevant to its usage) is found, leaving the class obsolete. To build the driver with the current version of DriverStudio requires that the deprecated classes in the code be replaced with the new classes. This requires the modification of the driver source code. This was unsuccessful as a number of critical errors were found during this stage, unfortunately further work required more time and was out of the scope of the project.

The purpose of rebuilding the driver is to include debugging information, and in particular is to use the “checked build” option of the DDK. This results in state information about variables and memory allocation being readily available to debugger systems. In terms of the DriverStudio environment, a powerful debugger system called SoftIce is able to translate debug information included in the “checked build” option into a symbol table. This allows SoftIce to step through the execution of the driver code line by line and track variable state information and memory allocation. This provides a powerful tool for analysis of the operation of driver code.

Lacking the use of the powerful debugger tool, analysis of the IOCTL code handler routines is carried out manually, examining the source code, requiring more time double checking the analysis.

6.3 Chapter Summary

The analysis of the mLAN driver is complete, although there remain areas lacking in clarity that can be resolved accurately through the use of a power debugger. It would be more conclusive if that form of analysis was carried out, unfortunately due to limitations of time and the scope of the project, it was not. The IOCTL codes can be grouped by the underlying usage of data structures and requests made, which neatly shows the modularisation of the driver. These modules represent a core area of functionality of the mLAN system by way of the driver.

7. Conclusion

7.1 Project Summary

Through the examination of the IEEE 1394 architecture, it is evident that Firewire is a flexible bus architecture that is ideally suited to networked real-time multimedia systems, as time-dependant data can be transferred in a deterministic manner. The mLAN extension demonstrates the flexible nature of Firewire, and with the use of integrated chips, it is possible to extend the IEEE 1394 architecture to interface with any media type. This raises the question about the usage of Firewire to transmit audio/visual media.

The Windows Driver Model is explicitly required for any kernel-mode driver, ensuring a reliable architecture for the representation of drivers and decoupled modularised driver stack for I/O operations. This separation and specialisation of drivers is inherently more flexible and sturdy than the legacy monolithic driver approach. In terms of development, drivers can be written to take advantage of established drivers supplied by Microsoft for generic I/O operations on standard device hardware. Through the modularisation and compartmentalisation of driver functionality into a driver stack, the WDM represents the grouping of services. It is this concept that allows the development of drivers to focus on the core features it is designed to expose. Yet there is still the overhead of making the driver conform to the WDM by requiring the handling of non-core driver services.

DriverStudio becomes a powerful tool for breaking down the development of the driver into its core and non-core components. DriverStudio smoothes the driver production process by providing a framework of classes to neatly encapsulate required generic driver functionality. This allows the development team to focus on the core aspects of the driver. This framework then forms the basis for the object model of the mLAN driver.

The object model of the mLAN driver relates to the WDM through the use of the DriverStudio framework. In chapter five, the structure of the driver is presented. From this, it is possible to understand how the classes conform to the WDM. It is unfortunate that the ideal methodology of analysis could not have been pursued. This limits the analysis, where clarity is sacrificed through the manual approach of analysis, which can be resolved accurately through the use of a power debugger. It would be more conclusive if that form of analysis was carried out, unfortunately due to the incompatibility issues experienced and the limitations of time and the scope of the project, it was not.

7.2 Future Extensions

As a possible extension, modifying the driver such that it builds should yield interesting results through further analysis. The driver would then be ready to be modified to further investigate the potentiality of the mLAN system in audio streaming. Further extensions would be to investigate the possibility of extending the driver to be compatible with the next version of the Windows operating system, or to investigate the usage of video streaming should and AV capable device be developed.
A further extension is to port the driver to the new driver architecture that will be seen in the next version of the Windows operating system.

8. References

[AE Notes, 2005] Audio Engineering Course Notes, 2005

[Anderson, 1999] Anderson, D. *FireWire System Architecture: IEEE1394a,* 2nd Edition, Mindshare Inc., Addison-Wesley, 1999

[ASIO, 2005] Steinberg, *Audio Streaming Input Output 2.1 SDK,* 2005

[Cant, 1999] Cant, C. *Writing Windows WDM Device Drivers,* R&D Books, Lawrence, 1999

[DriverStudio, 2004] Compuware Corporation, DriverStudio Version 3.2, 2004

[Haig, 2002] Haig, D. *Firewire Breakout Box Driver,* Honours Thesis, Rhodes University, 2002

[McKenzie, 2003] McKenzie, B. *1394 Node-Targeted Asynchronous Transfers* [Online] http://www.wd-3.com/archive/ 1394NodeTransfers.htm, 2003

[MSDN, 2005] Microsoft Developers Network (MSDN) Library, April 2005

[Oney, 1999] Oney, W. *Programming the Microsoft Windows Driver Model,* Microsoft Press, Washington, 1999

[Oney, 2003] Oney, W. *Programming the Microsoft Windows Driver Model,* 2nd Edition, E-BOOK 2003

[Ward, et al, 1985] Ward, P. and Mellor, S. *Structured Development for Real-Time Systems,* Prentice-Hall, Inc., New Jersey, 1985

Appendix

A. IOCTL analysis and documentation:

Each IOCTL has a specification of the input and output requirement for the IRP's buffer. The actual description of each struct is detailed in Part B of the Appendix. There is a brief textual description of what the IOCTL handler routine does.

Below the description is a diagram representing a hybrid between a Ward and Mellor Structured Data Diagram, and Larry Constantine's Data Flow Diagram. It is hybridised because neither method neatly encapsulates the IOCTL handler, but rather taking elements from both methods and combining them yields a clearer picture. The entire diagram can be viewed as process instead of each full circle, as in the Ward and Mellor approach. The dotted circle still represents a trigger for the process, responding to the IRP dispatch routine. The parallel lines represent a data store, common to both methods of modelling. Each full circle represents a sub-process that acts on a data store. The dotted lines represent the flow of the trigger, while the solid lines represent the flow of data and control.

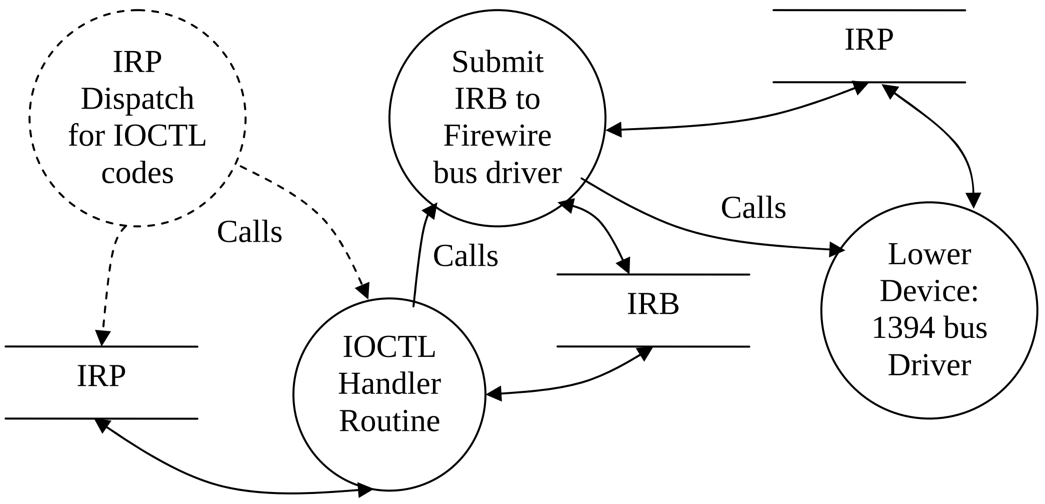
IEEE 1394 based mlan IOCTL codes

IOCTL_MLAN_ALLOCATE_ADDRESS_RANGE

Input Buffer:
A Pointer to an `ALLOCATE_ADDRESS_RANGE` struct.

Output Buffer:
A pointer to the same `ALLOCATE_ADDRESS_RANGE` struct.

Textual Description:
The struct specifies the address range that will be allocated in Host Controller address space by the IEEE 1394 bus driver. An address range has to be allocated before the host controller can respond to any communication from that node. The mLAN driver sends an IRB containing the address range to the 1394 driver. The IRB function code is that of `REQUEST_ALLOCATE_ADDRESS_RANGE`. A handle to the address range is returned.

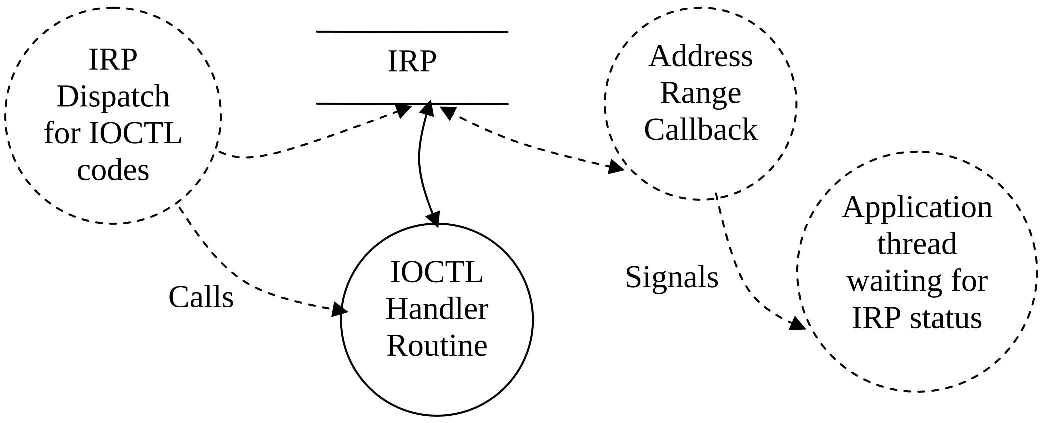

IRP Dispatch for IOCTL codes
Calls
IRP
Submit IRB to Firewire bus driver
Calls
IOCTL Handler Routine
IRP
Calls
IRB
Lower Device: 1394 bus Driver

IOCTL_MLAN_ADDRESS_RANGE_NOTIFY

Input Buffer:
A pointer to an `ADDRESS_RANGE_NOTIFY` struct.

Output Buffer:
A pointer to the same `ADDRESS_RANGE_NOTIFY` struct is returned, which will need a separate thread to wait for the completed IRP.

Textual Description:
The request adds the IRP to a notification queue for the allocated address ranges. The callback routine is signalled when an I/O operation is detected on that address range. The callback routine completes the IRP and fills in the node address of the source for that I/O transaction.

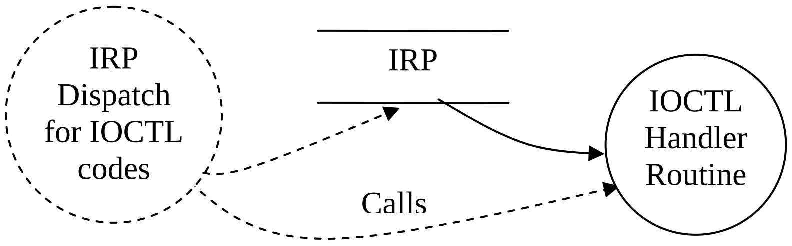

IRP Dispatch for IOCTL codes
IRP
Address Range Callback
Application thread waiting for IRP status
Calls
IOCTL Handler Routine
Signals

IOCTL_MLAN_SET_ADDRESS

Input Buffer:
A pointer to a `SET_ADDRESS_DATA` struct.

Output Buffer:
Nothing is returned.

Textual Description:
This request copies the data contained within the input struct to the node specified by the allocated address range handle.

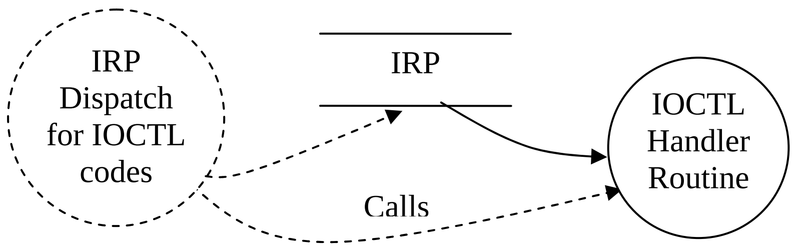


IOCTL_MLAN_GET_ADDRESS

Input Buffer:
A pointer to a `GET_ADDRESS_DATA` struct.

Output Buffer:
A pointer to the same `GET_ADDRESS_DATA` struct is returned.

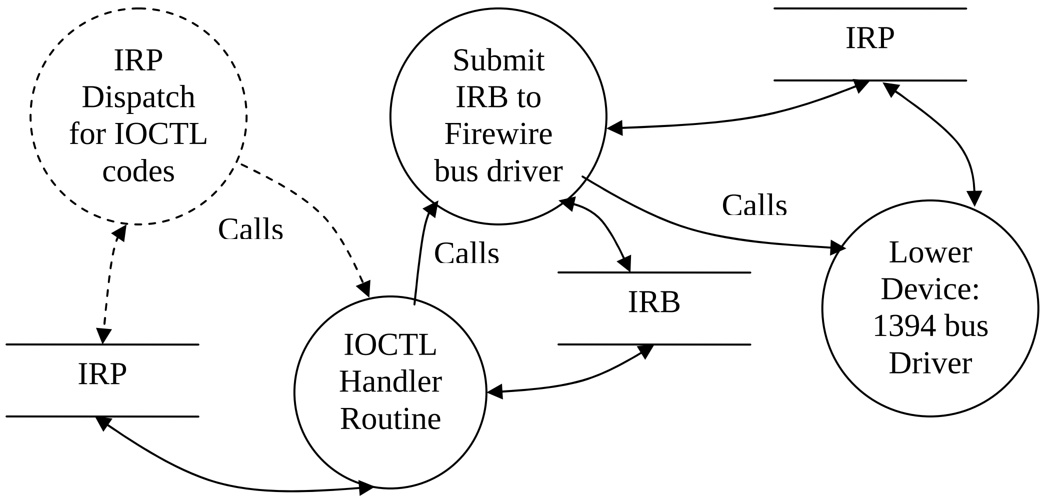
Textual Description:
The request returns the node address and the data contained within that address specified by the allocated address range handle.



IOCTL_MLAN_FREE_ADDRESS_RANGE
Input Buffer:
A `HANDLE` to the intended address range.

Output Buffer:
Nothing is returned.

Textual Description:
The IOCTL handler routine will try to free the address range by sending an IRB containing the address range to the 1394 driver. The IRB function code is that of `REQUEST_FREE_ADDRESS_RANGE`.

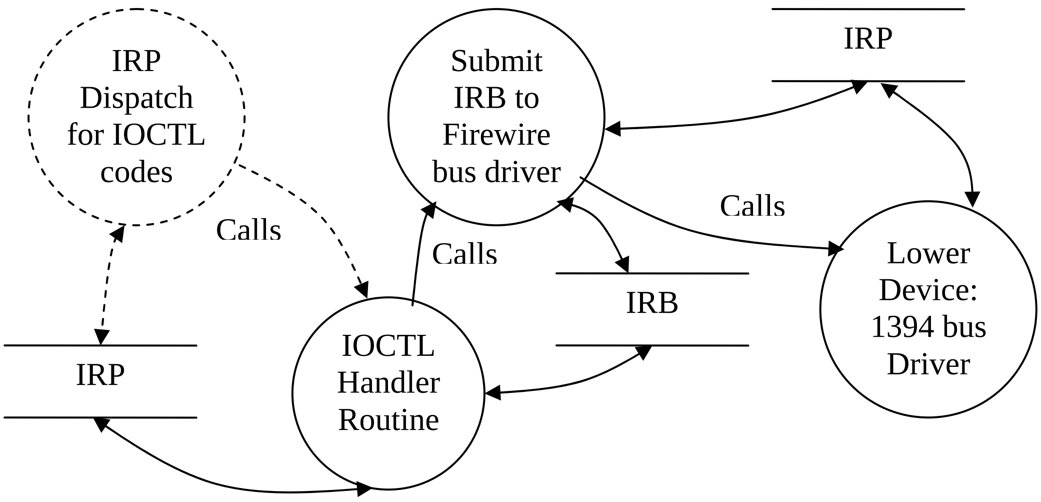

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_ASYNC_LOCK

Input Buffer:
A pointer to an `ASYNC_LOCK` struct.

Output Buffer:
A pointer to the same `ASYNC_LOCK` struct is returned.

Textual Description:
The IOCTL code is used to perform a locked asynchronous operation on the indicated destination address. This is achieved by summiting an IRB to the 1394 driver with a function code of `REQUEST_ASYNC_LOCK`.

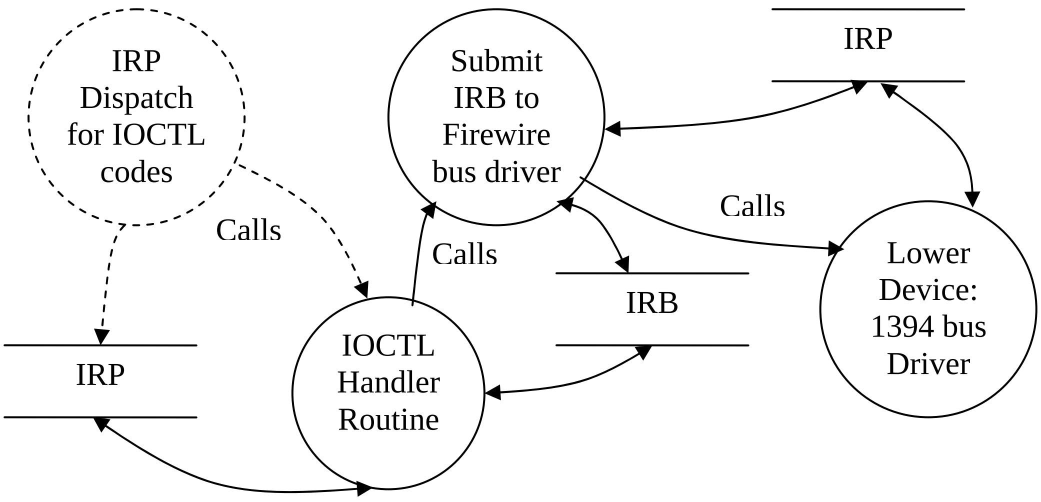

IRP Dispatch for IOCTL codes
Calls
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_ASYNC_READ

Input Buffer:
A pointer to an `ASYNC_READ` struct.

Output Buffer:
A pointer to the same `ASYNC_READ` struct is returned.

Textual Description:
The IOCTL code is used to perform an asynchronous read operation on the indicated destination address. This is achieved by summiting an IRB to the 1394 driver with a function code of `REQUEST_ASYNC_READ`.

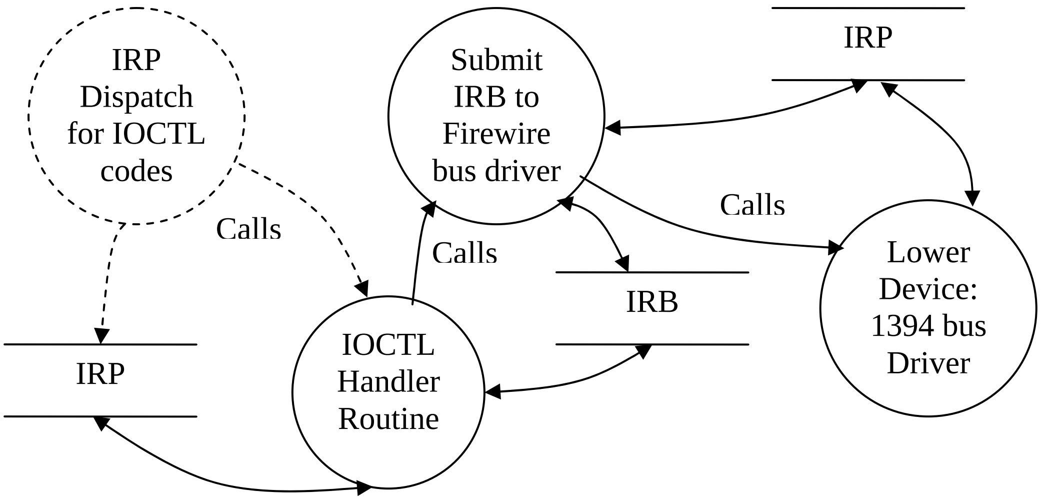

IRP Dispatch for IOCTL codes
Calls
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_ASYNC_WRITE

Input Buffer:
A pointer to an `ASYNC_WRITE` struct.

Output Buffer:
Nothing is returned.

Textual Description:
The IOCTL code is used to perform an asynchronous write operation on the indicated destination address. This is achieved by summiting an IRB to the 1394 driver with a function code of `REQUEST_ASYNC_WRITE`.

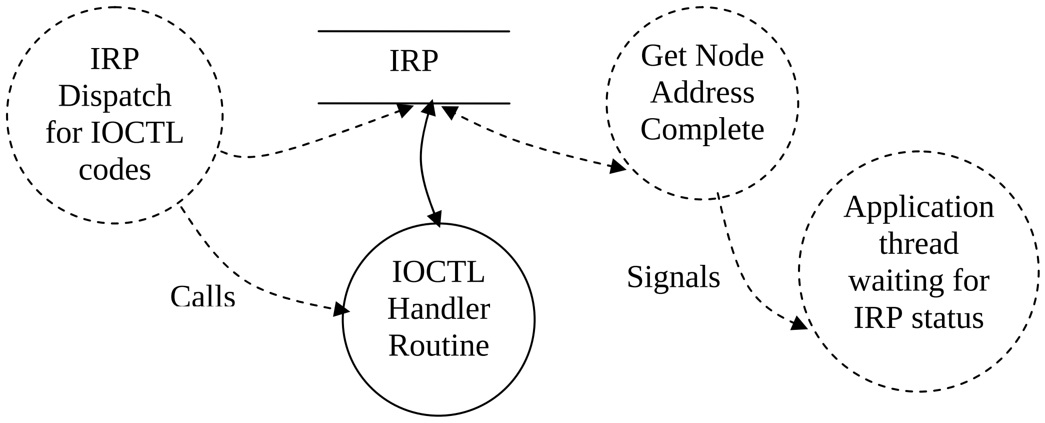

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_BUS_RESET_NOTIFY

Input Buffer:
A pointer to a `MLAN_BUS_RESET_NOTIFY` struct.

Output Buffer:
A pointer to the same `MLAN_BUS_RESET_NOTIFY` struct is returned.

Textual Description:
The request sends the IRP to a bus reset notification queue, which will be signalled by the driver when a bus reset occurs. The struct is returned with the node address of the device that caused the reset.

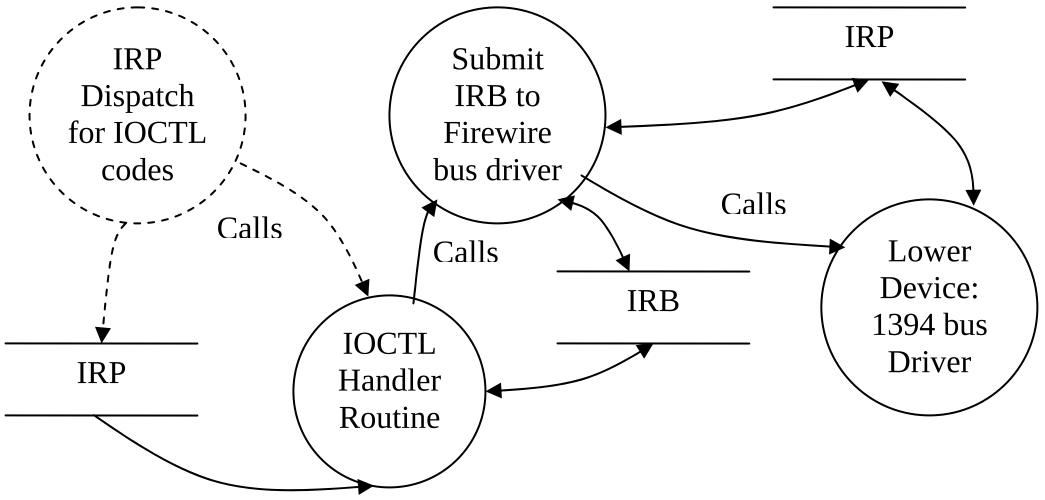

IRP Dispatch for IOCTL codes
IRP
Get Node Address Complete
Application thread waiting for IRP status
Calls
IOCTL Handler Routine
Signals

IOCTL_MLAN_BUS_RESET

Input Buffer:
A ULONG type.

Output Buffer:
Nothing is returned.

Textual Description:
The input type is to specify the reset flags of the IRB that will be sent to the 1394 driver. The IRB has a function code of `REQUEST_BUS_RESET`. This initiates a bus reset.

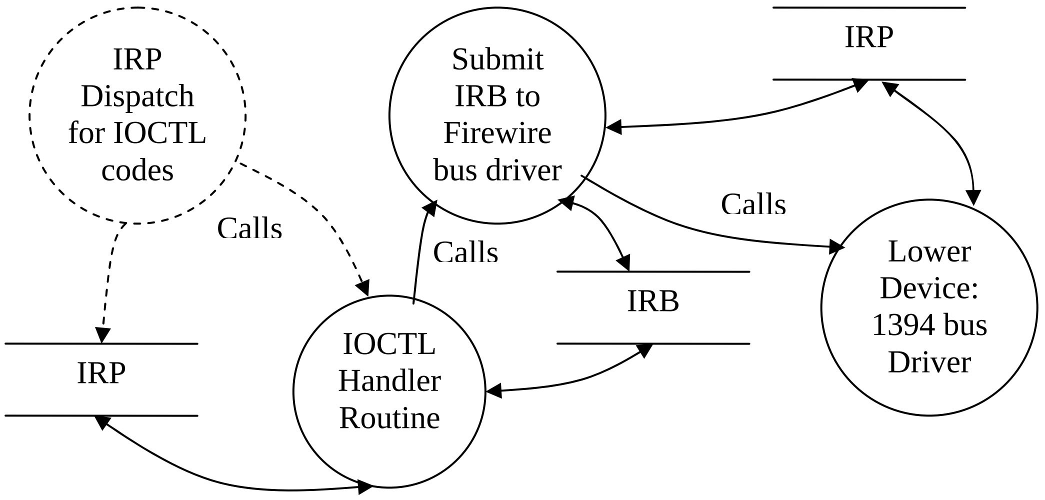

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_GET_LOCAL_HOST_INFO

Input Buffer:
A pointer to a `GET_LOCAL_HOST_INFORMATION` struct.

Output Buffer:
A pointer to the same `GET_LOCAL_HOST_INFORMATION` struct is returned.

Textual Description:
This receives a request for the local host information, which is attained by constructing an IRB with a function code of `REQUEST_GET_LOCAL_HOST_INFO`. The struct stores the input data as well as the host information when it is received.

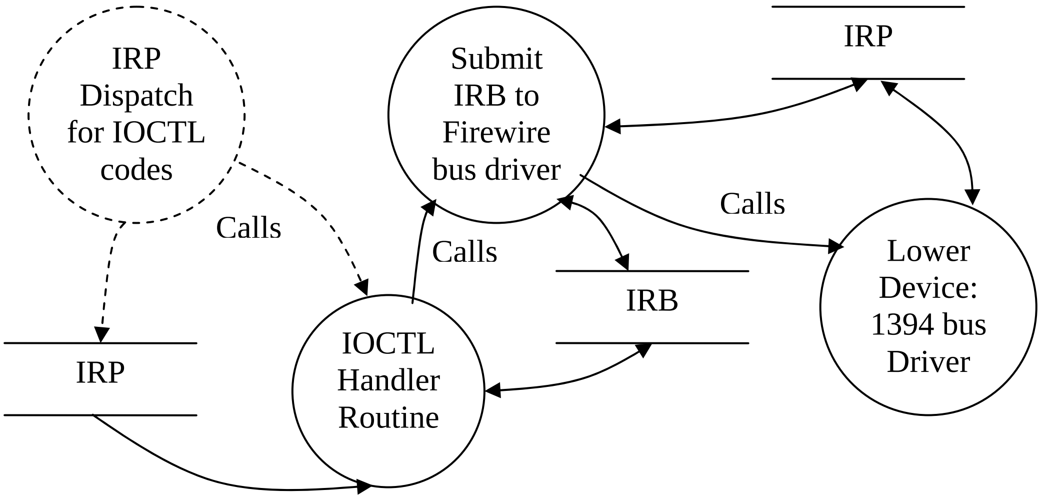

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_SEND_PHY_CONFIG_PACKET

Input Buffer:
A pointer to a `PHY_CONFIGURATION_PACKET` struct.

Output Buffer:
Nothing is returned.

Textual Description:
The handler forwards a send PHY config packet request to the 1394 bus driver. The IRB function code is `REQUEST_SEND_PHY_CONFIG_PACKET`.

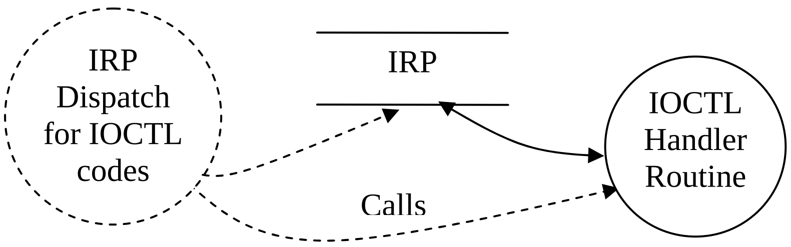

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_GET_LOCAL_NODE_ADDRESS

Input Buffer:
A pointer to a `GET_LOCAL_NODE_ADDRESS` struct.

Output Buffer:
A pointer to the same `GET_LOCAL_NODE_ADDRESS` struct is returned.

Textual Description:
This request returns the local host node bus and node number.

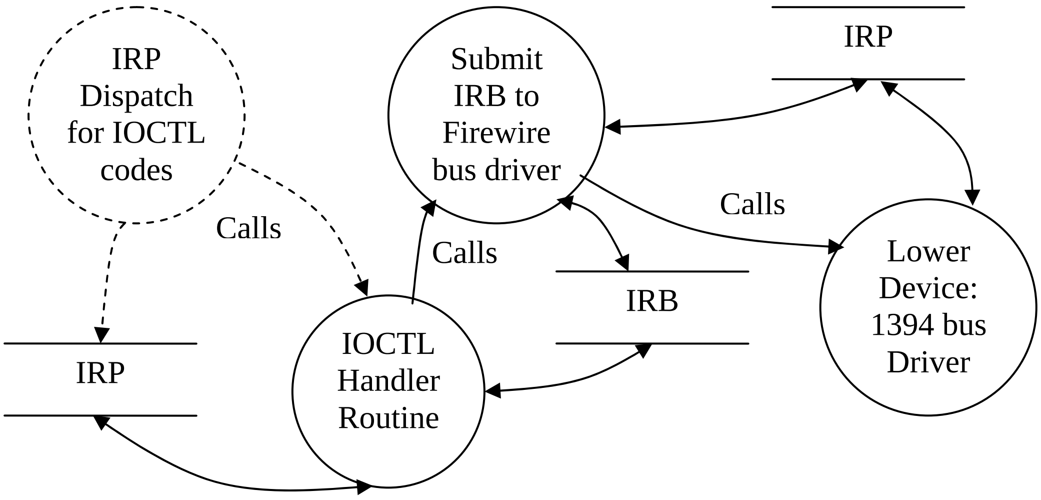


IOCTL_MLAN_GET_CHANNELS_AVAILABLE

Input Buffer:
This IOCTL code uses no input.

Output Buffer:
This returns a `LARGE_INTEGER` struct (refer to MSDN for specification).

Textual Description:
It makes a request that returns the bandwidth and channels currently available on the IEEE 1394 bus. The returning information specifies a 64-bit number with a high and low part of 32-bits each. The IRB has a function code of `REQUEST_ISOCH_QUERY_RESOURCES.`

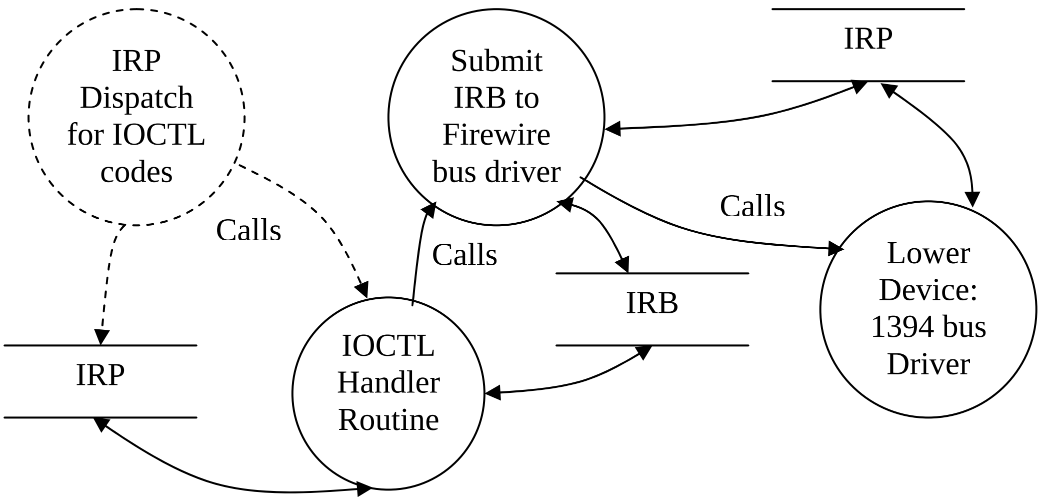

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_ALLOCATE_CHANNEL

Input Buffer:
A ULONG type.

Output Buffer:
This it returns the same type.

Textual Description:
This is to specify the requested channel number. The request is forwarded to the 1394 bus driver, which allocates an isochronous channel to be used in subsequent operations. The IRB function code is `REQUEST_ISOCH_ALLOCATE_CHANNEL.`

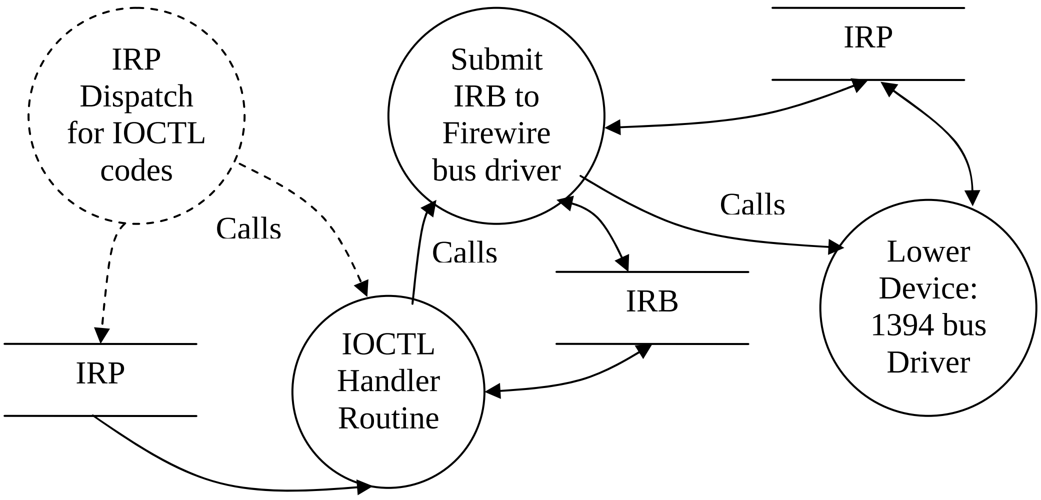

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_RELEASE_CHANNEL

Input Buffer:
A ULONG type.

Output Buffer:
No return value.

Textual Description:
This is to specify the channel number to release. The request is forwarded to the 1394 bus driver, which releases an isochronous channel that was previously allocated. The IRB function code is `REQUEST_ISOCH_FREE_CHANNEL.`

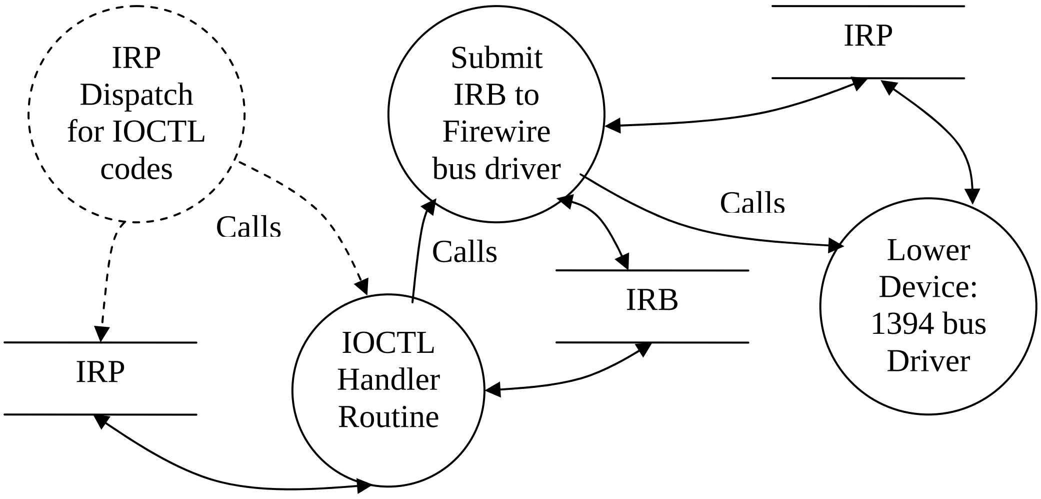

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_GET_BANDWIDTH_AVAILABLE

Input Buffer:
There is no required input for this IOTCL code.

Output Buffer:
A ULONG type.

Textual Description:
The request returns the bandwidth and channels currently available on the IEEE 1394 bus. It returns to the application the available bandwidth as expressed in bytes per isochronous frame. The IRB function code is `REQUEST_ISOCH_QUERY_RESOURCES.`

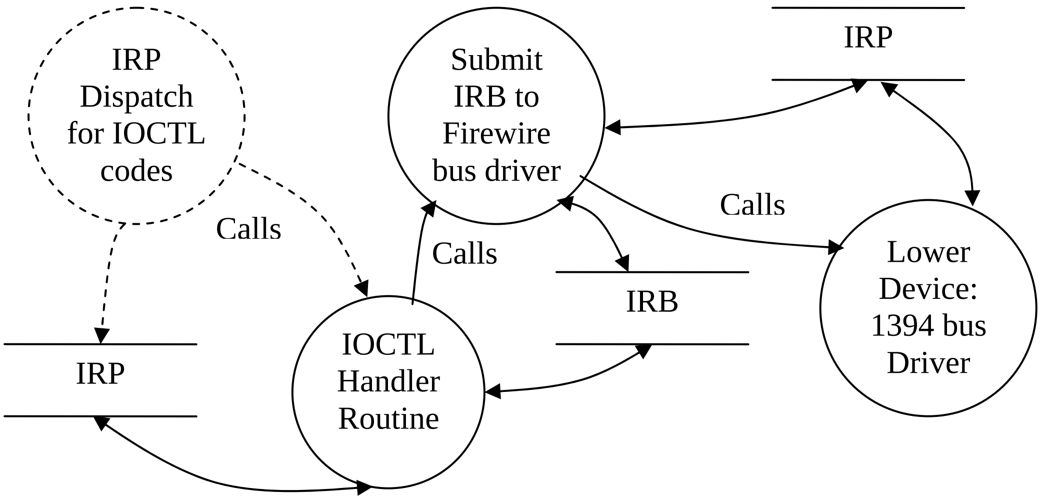

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_ALLOCATE_BANDWIDTH

Input Buffer:
A ULONG type.

Output Buffer:
It returns a handle to use to refer to the bandwidth resource.

Textual Description:
This is to specify the "max bytes per frame requested". The request allocates isochronous bandwidth to be used in subsequent operations. The IRB function code is `REQUEST_ISOCH_ALLOCATE_BANDWIDTH.`

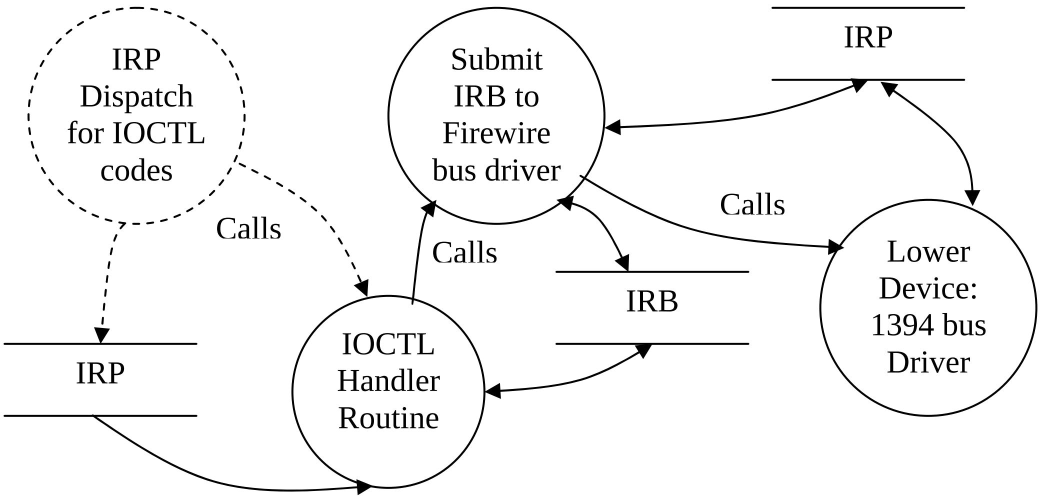

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_RELEASE_BANDWIDTH

Input Buffer:
The input buffer takes the handle that was used as a reference to the allocated bandwidth.

Output Buffer:
There is no output returned.

Textual Description:
The request releases the isochronous bandwidth that was previously allocated. The IRB has a function code of `REQUEST_ISOCH_FREE_BANDWIDTH.`


IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

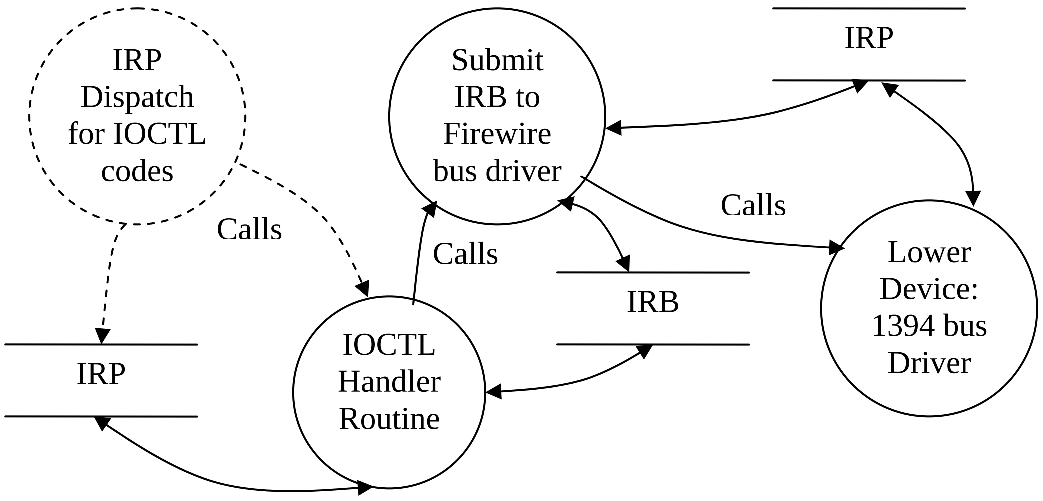
Isochronous Stream based mlan IOCTL codes

IOCTL_MLAN_ALLOCATE_STREAM

Input Buffer:
The input buffer uses a `MLAN_ISOCH_PARAM` struct, which specifies the type of the stream to be used.

Output Buffer:
The same struct is returned, specifying the allocated stream information.

Textual Description:
The request responds to the need to allocate an isochronous stream resource specified by the input struct. It goes through several steps to allocate a stream resource. It creates the specified stream type, and gets a channel from the underlying 1394 bus driver. The IRB function codes submitted to the Firewire driver are as follows: `REQUEST_ISOCH_ALLOCATE_BANDWIDTH`, `REQUEST_ISOCH_ALLOCATE_CHANNEL`, `REQUEST_ISOCH_ALLOCATE_RESOURCES`, `REQUEST_ISOCH_ATTACH_BUFFERS`. Inside the struct, a pointer is provided to a `MLAN_STREAM_STATUS` struct, which contains the information about the allocated stream.


IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

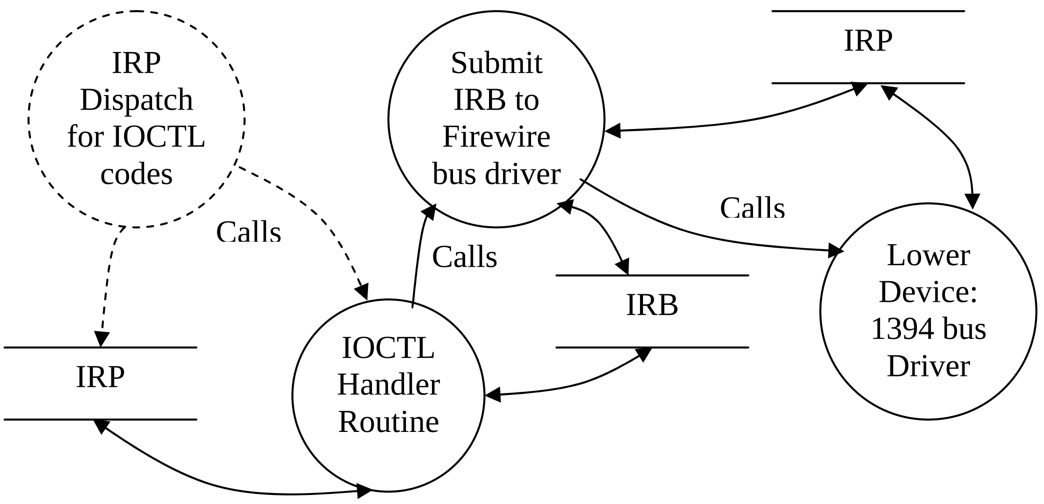
IOCTL_MLAN_START_STREAM

Input Buffer:
The input buffer takes a `MLAN_STREAM_COMMAND_PARAM` struct, which contains the allocated stream resource handle.

Output Buffer:
A pointer to the same struct is returned.

Textual Description:
The request uses the allocated stream handle from the allocate stream IOCTL, checks if it is a send or receive stream, and then starts the stream. IRBs with function codes

of `REQUEST_ISOCH_QUERY_CYCLE_TIME` and `REQUEST_ISOCH_TALK/LISTEN` (depends on send or receive streams) are used.

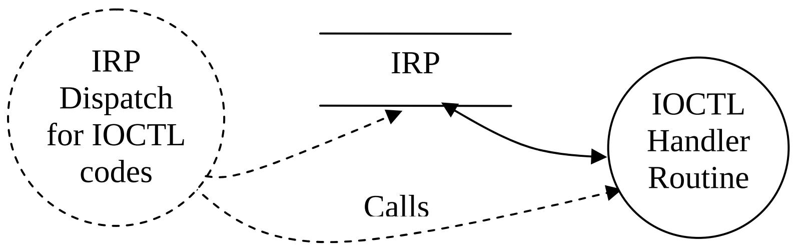

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_CONNECT_SEQUENCES_TO_DEVICES

Input Buffer:
The input buffer uses a `MLAN_ISOCH_PARAM` struct, containing a handle to the stream.

Output Buffer:
A pointer to the same struct is returned.

Textual Description:
The request uses the incoming handle of the stream that was previously allocated to process each sequence. It checks to see if the sequence type has changed and sets up the current connection map.

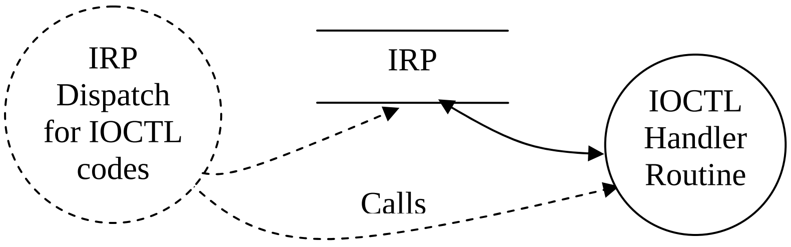


IOCTL_MLAN_SET_SYT_SOURCE

Input Buffer:
The input buffer uses a `MLAN_ISOCH_PARAM` struct.

Output Buffer:
The request returns a pointer to the same struct.

Textual Description:
The request is used to set the SYT source associated with the stream information struct. The allocated stream has to be a sending stream for this to work, and sets the SYT for the stream. If the stream is not the clock master it gets the SYT value from the corresponding receive stream.

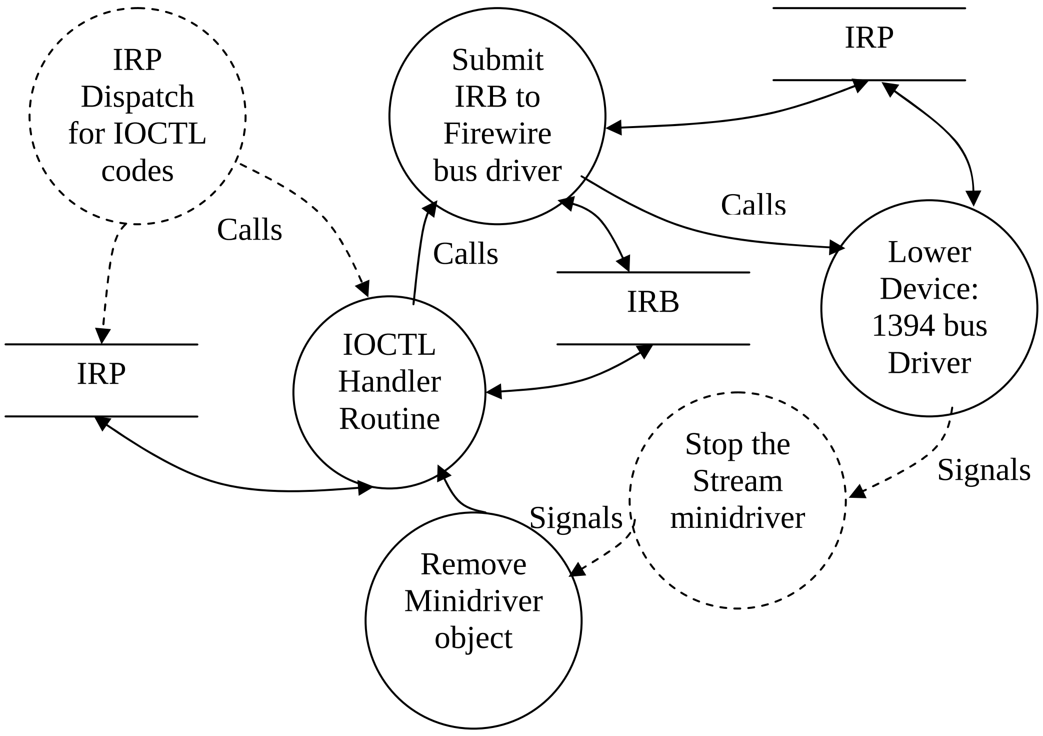


IOCTL_MLAN_STOP_STREAM

Input Buffer:
The input buffer uses a `MLAN_STREAM_COMMAND_PARAM` struct.

Output Buffer:
The request returns a pointer to the same struct.

Textual Description:
The request stops the stream specified by the handle in the struct. The stream has to have all sequences free of use, in other words, the sequences have to be finished streaming. The same struct is returned regardless of whether the function succeeds or fails. The IRB uses a function code of `REQUEST_ISOCH_STOP.`

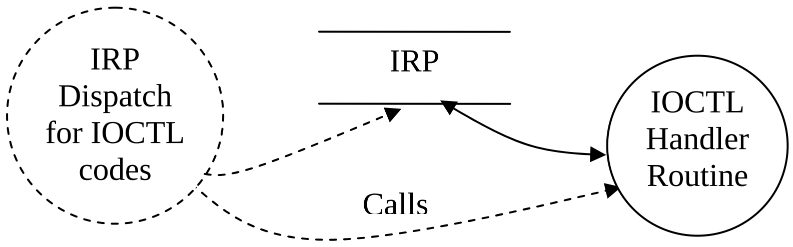

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
IRB
Lower Device: 1394 bus Driver
IRP
IOCTL Handler Routine
Stop the Stream minidriver
Signals
Signals
Remove Minidriver object

IOCTL_MLAN_GET_STREAM_INFO

Input Buffer:
The input buffer uses a `MLAN_ISOCH_PARAM` struct.

Output Buffer:
The request returns a pointer to the same struct.

Textual Description:
The request updates the status information of the allocated stream, including which sequences are in use.

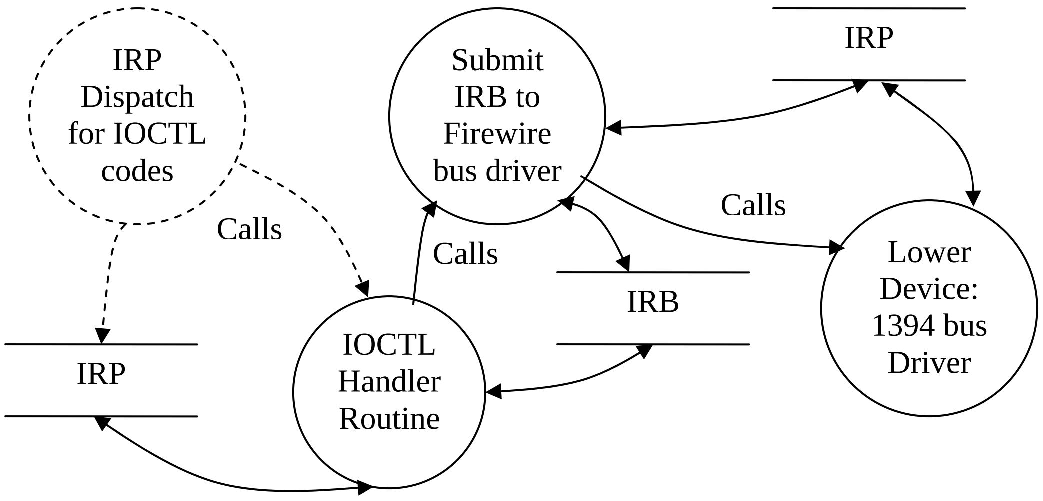


IOCTL_MLAN_FREE_STREAM

Input Buffer:
The input buffer uses a `MLAN_STREAM_COMMAND_PARAM` struct.

Output Buffer:
The request returns a pointer to the same struct.

Textual Description:
The request frees up the memory allocated for the stream. The stream has to be stopped first before freeing up the stream. IRBs are dispatched with function codes of `REQUEST_ISOCH_DETACH_BUFFERS`, `REQUEST_ISOCH_FREE_RESOURCES`, `REQUEST_ISOCH_FREE_CHANNEL` and `REQUEST_ISOCH_FREE_BANDWIDTH`.

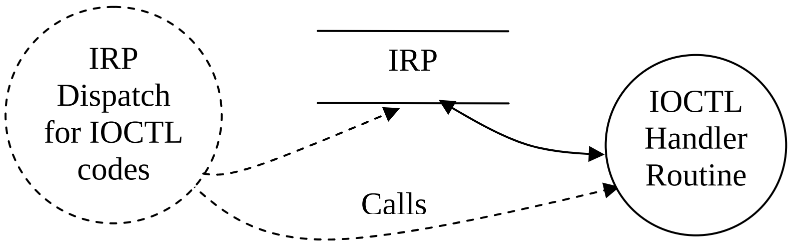

IRP Dispatch for IOCTL codes
Submit IRB to Firewire bus driver
IRP
Calls
Calls
Calls
Lower Device: 1394 bus Driver
IRB
IRP
IOCTL Handler Routine

IOCTL_MLAN_GET_DRIVER_VERSION

Input Buffer:
There is no required input.

Output Buffer:
A ULONG type is returned specifying the driver interface version.

Textual Description:
This request returns the driver version.



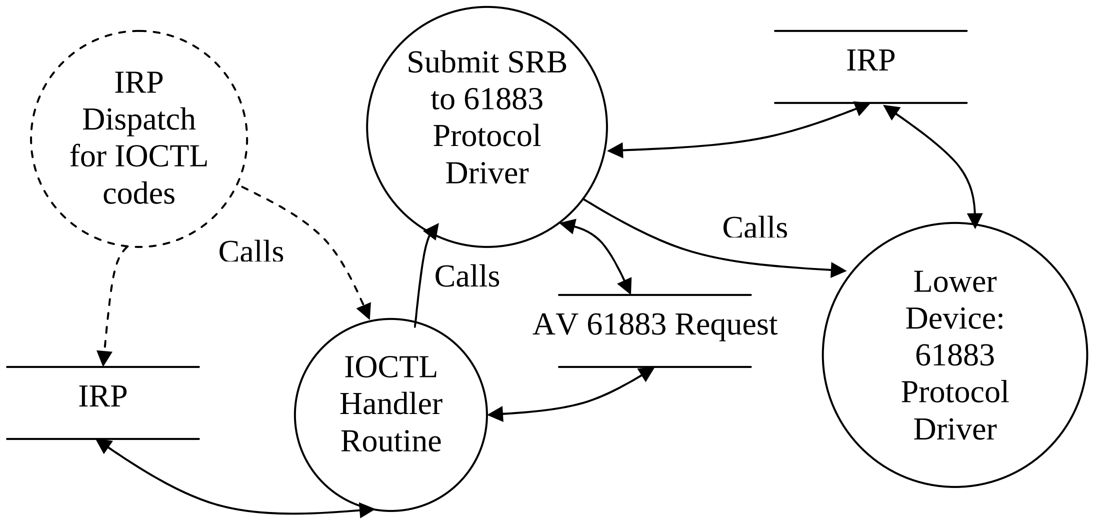
IEC 61883 based mlan IOCTL codes

IOCTL_MLAN_61883_CONNECT_PLUG

Input Buffer:
The input buffer takes a `CMP_CONNECT` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request is forwarded to the underlying 61883 protocol driver, using the handles of plugs contained in the input struct to specify which plugs to connect, using a function code of `Av61883_Connect`. A handle to the connection is returned.

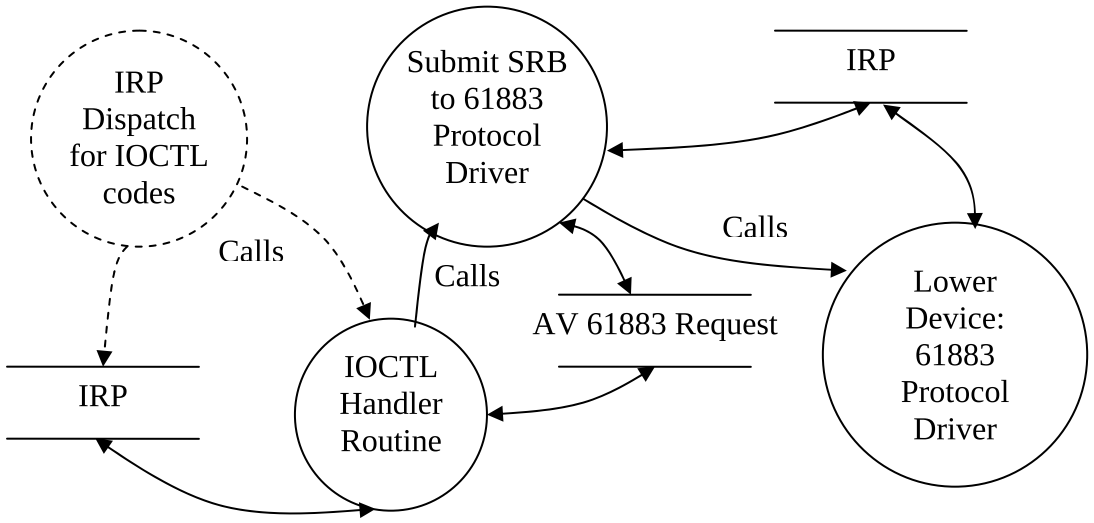

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
IRP
IOCTL Handler Routine
Lower Device: 61883 Protocol Driver

IOCTL_MLAN_61883_CREATE_PLUG

Input Buffer:
The input buffer takes a `CMP_CREATE_PLUG` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request is sent to 61883 protocol driver to create a plug, using a function code of `Av61883_CreatePlug`. A handle to the plug is returned in the struct.

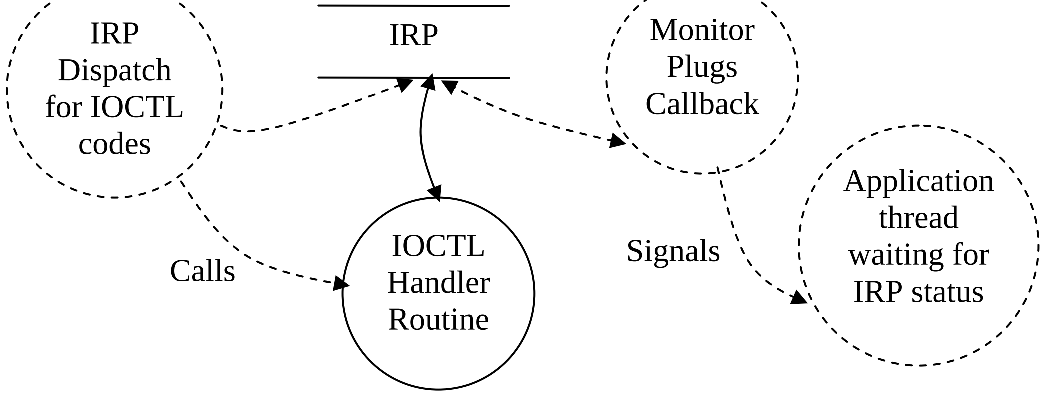

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_PLUG_NOTIFY

Input Buffer:
The input buffer takes a `MLAN_CMP_PLUG_NOTIFY` struct.

Output Buffer:
The notification routine returns the same struct when it is signalled to run.

Textual Description:
The request adds the IRP to a queue for notification from the 61883 protocol driver that monitors access to the plug and then returns information in the struct.

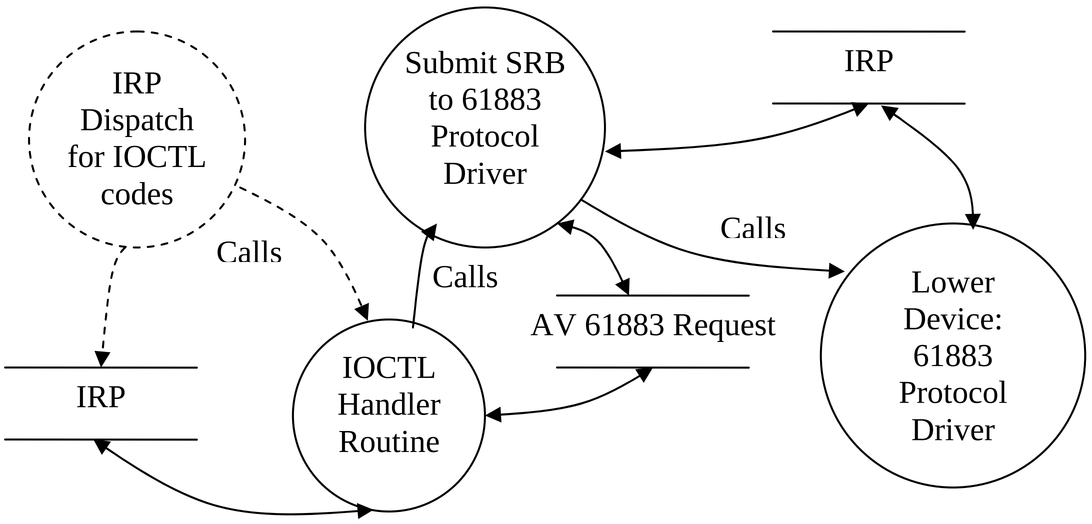

IRP Dispatch for IOCTL codes
IRP
Monitor Plugs Callback
Application thread waiting for IRP status
Signals
Calls
IOCTL Handler Routine

IOCTL_MLAN_61883_DELETE_PLUG

Input Buffer:
The input buffer takes a `CMP_DELETE_PLUG` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request tells the 61883 protocol driver to delete the plug (as a handle in the struct), using a function code of `Av61883_DeletePlug`.

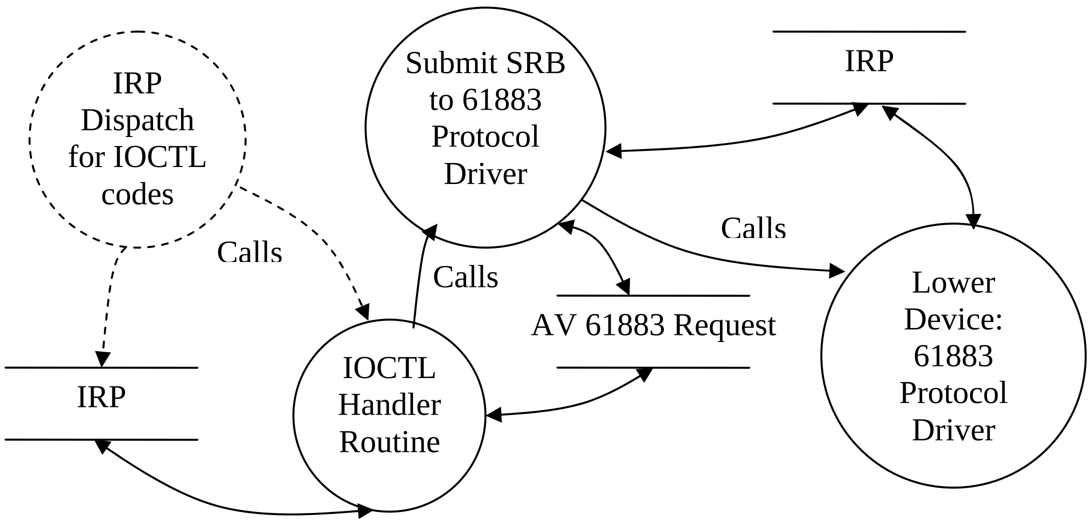

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
Lower Device: 61883 Protocol Driver
AV 61883 Request
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_DISCONNECT_PLUG

Input Buffer:
The input buffer takes a `CMP_DISCONNECT` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request extracts the handle to the plug connection from the input struct. The connection is released by making the appropriate call to the underlying 61883 protocol driver, using a function code of `Av61883_Disconnect`.

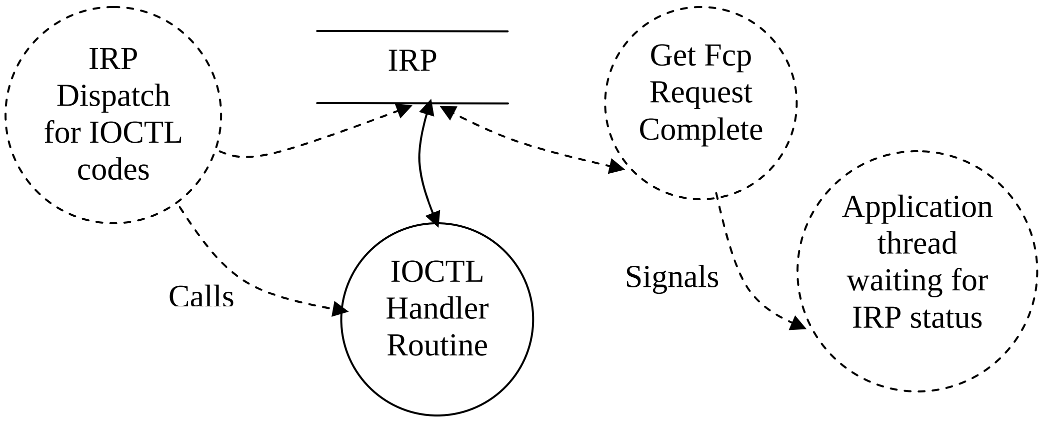

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
Lower Device: 61883 Protocol Driver
AV 61883 Request
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_GET_FCP_REQUEST

Input Buffer:
The input buffer takes a `MLAN_FCP_GET_REQUEST` struct.

Output Buffer:
The same struct is returned, but if the asynchronous method of communicating with the 61883 protocol driver then the IRP is sent to completion status queue routine. This routine will return the struct when that completion routine fires.

Textual Description:
The request uses the 61883 get FCP request of the 61883 driver, using a function code of `Av61883_GetFcpReqeuest`.

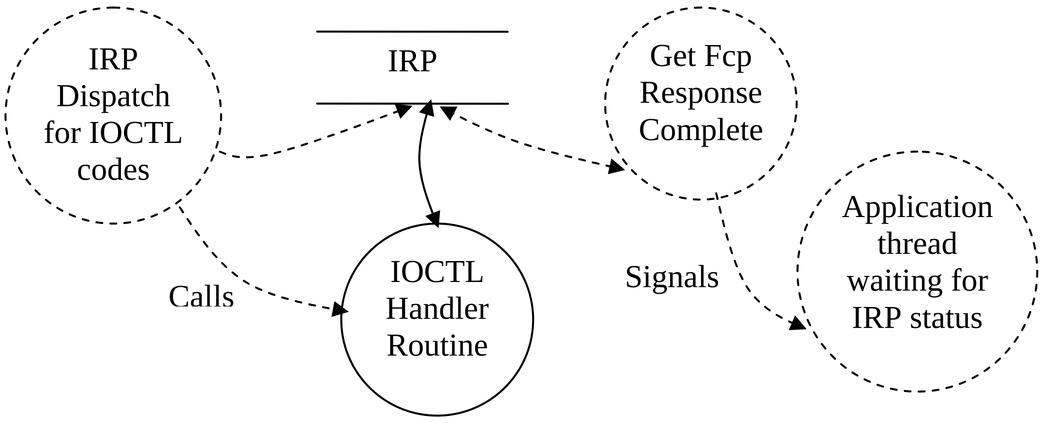

IRP Dispatch for IOCTL codes
IRP
Get Fcp Request Complete
Application thread waiting for IRP status
Calls
IOCTL Handler Routine
Signals

IOCTL_MLAN_61883_GET_FCP_RESPONSE

Input Buffer:
The input buffer takes a `MLAN_FCP_GET_RESPONSE` struct.

Output Buffer:
The same struct is returned, but if the asynchronous method of communicating with the 61883 protocol driver then the IRP is sent to completion status queue routine. This routine will return the struct when that completion routine fires.

Textual Description:
The request uses the 61883 get FCP response of the 61883 driver, using a function code of `Av61883_GetFcpResponse`.

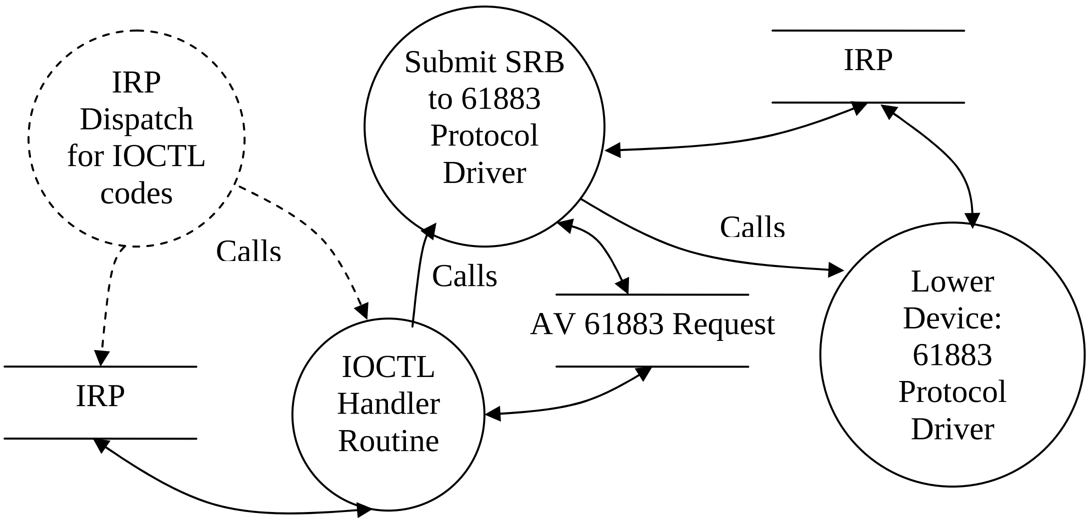

IRP Dispatch for IOCTL codes
IRP
Get Fcp Response Complete
Application thread waiting for IRP status
Calls
IOCTL Handler Routine
Signals

IOCTL_MLAN_61883_GET_PLUG_HANDLE

Input Buffer:
The input buffer takes a `CMP_GET_PLUG_HANDLE` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request returns a handle to the plug specified by parameters in the struct, using a function code of `Av61883_GetPlugHandle`.

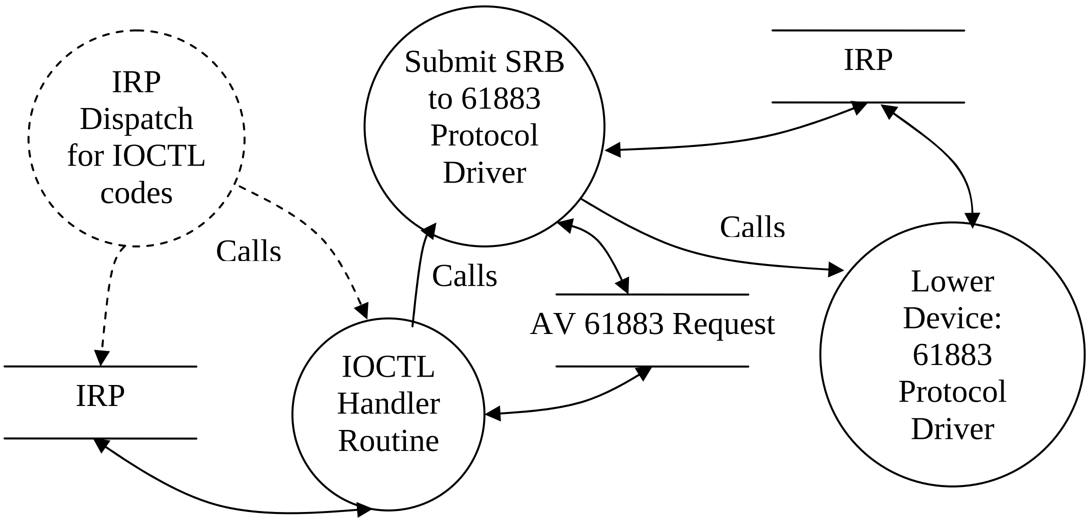

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_GET_PLUG_STATE

Input Buffer:
The input buffer takes a `CMP_GET_PLUG_STATE` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request returns the current status and information of the plug specified by the handle in the struct, using a function code of `Av61883_GetPlugState`.

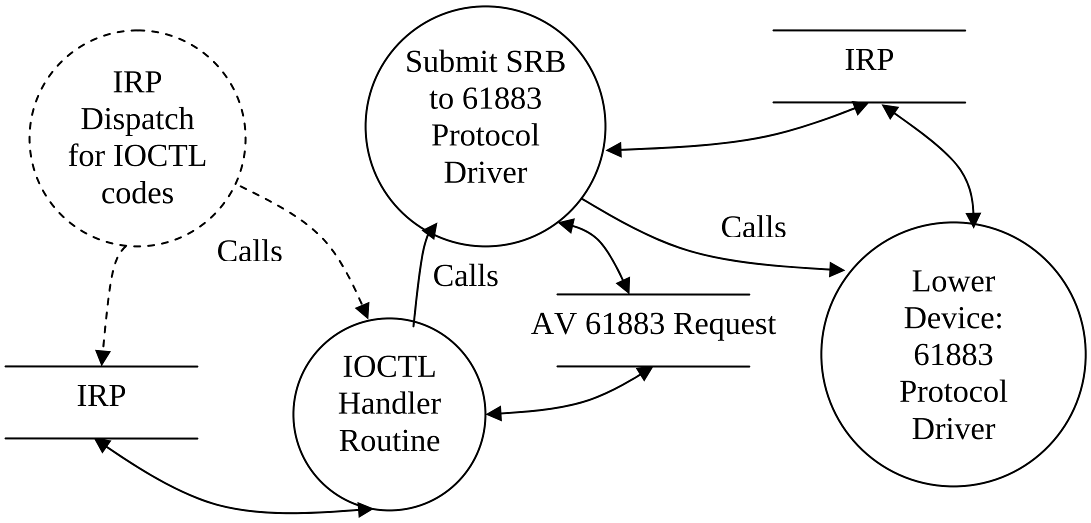

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_SEND_FCP_REQUEST

Input Buffer:
The input buffer takes a `MLAN_FCP_SEND_REQUEST` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request uses the 61883 send FCP request AV_request of the 61883 driver, using a function code of `Av61883_SendFcpRequest`.

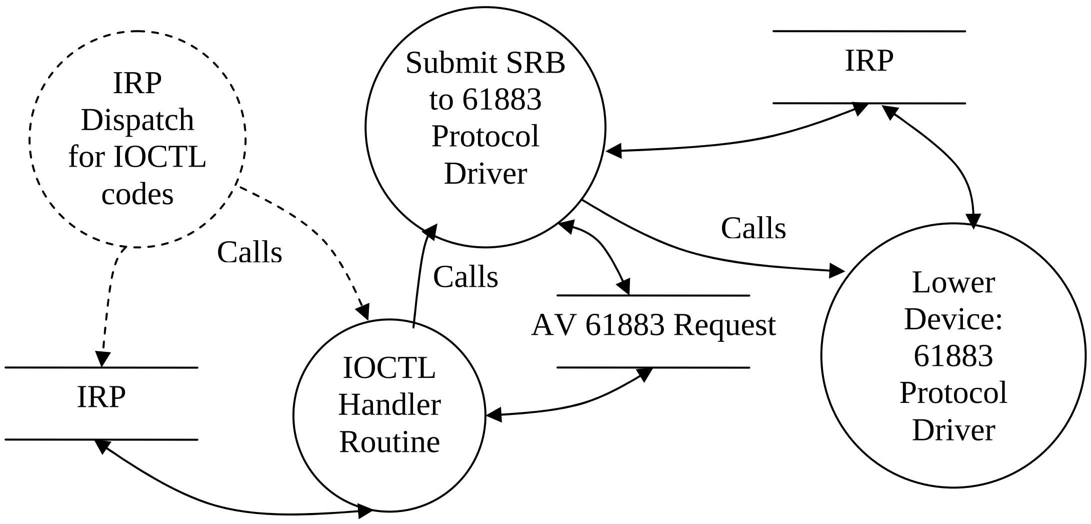

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_SEND_FCP_RESPONSE

Input Buffer:
The input buffer takes a `MLAN_FCP_SEND_RESPONSE` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request uses the 61883 send FCP response AV_request of the 61883 driver, using a function code of `Av61883_SendFcpResponse`.

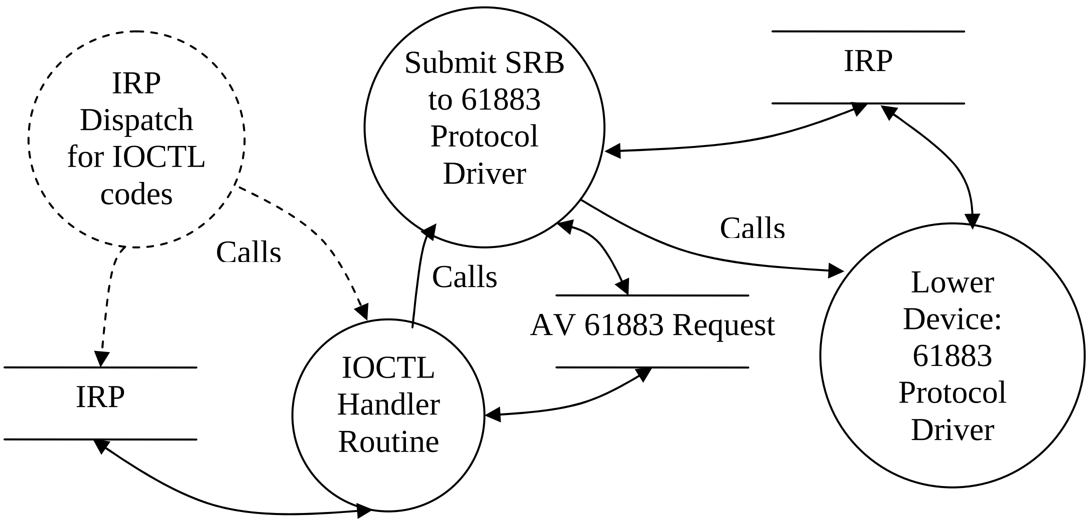

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_SET_FCP_NOTIFY

Input Buffer:
The input buffer takes a `SET_FCP_NOTIFY` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request uses the 61883 set FCP notify AV_request of the 61883 driver, using a function code of `Av61883_SetFcpNotify`.

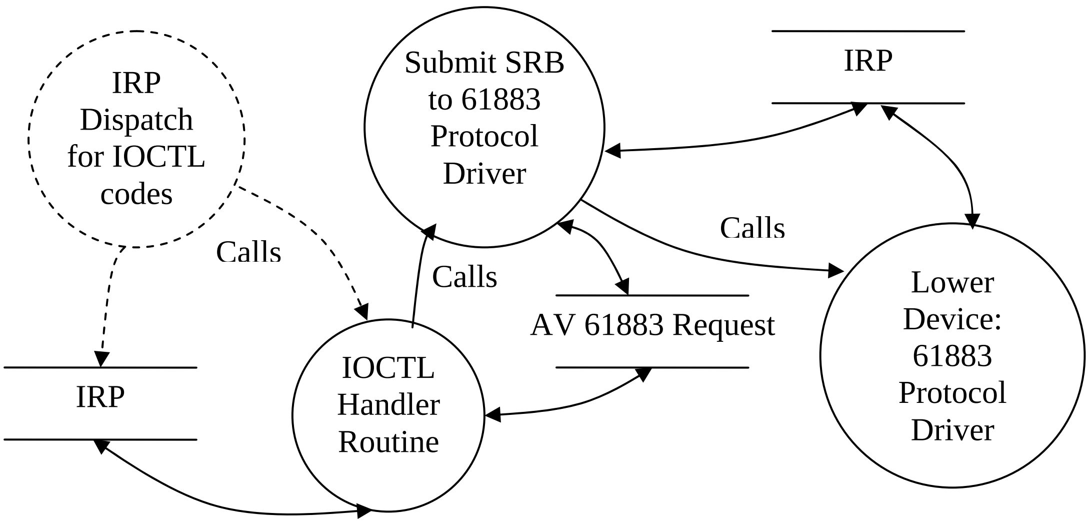

IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

IOCTL_MLAN_61883_SET_PLUG

Input Buffer:
The input buffer takes a `CMP_SET_PLUG` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request uses the 61883 set plug AV_request of the 61883 driver, using a function code of `Av61883_Setplug`.


IRP Dispatch for IOCTL codes
Submit SRB to 61883 Protocol Driver
IRP
Calls
Calls
Calls
AV 61883 Request
Lower Device: 61883 Protocol Driver
IRP
IOCTL Handler Routine

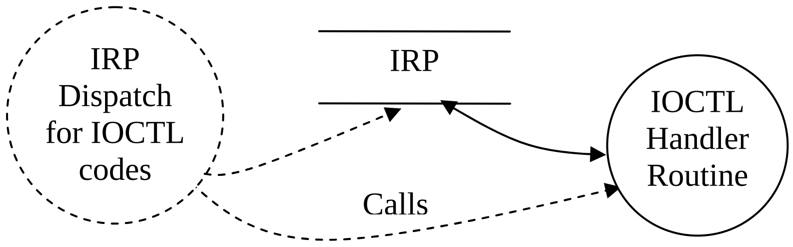
ASIO based mlan IOCTL codes

IOCTL_MLAN_ASIO_INFO

Input Buffer:
The input buffer takes a `MLAN_ASIO_INIT_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request finds an output stream that has ASIO sequences, regardless of it being a send or receive stream. It extracts information about that stream it finds to put into the input struct. It does not modify the driver state.

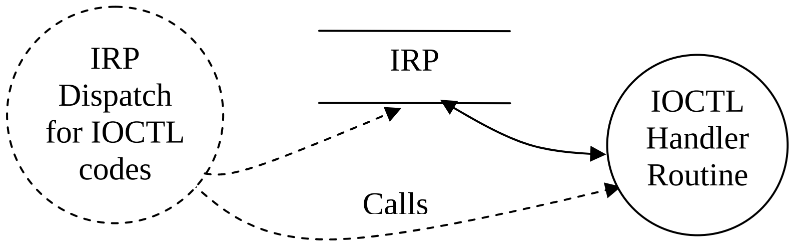


IOCTL_MLAN_ASIO_INIT

Input Buffer:
The input buffer takes a `MLAN_ASIO_INIT_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request finds an output stream that has ASIO sequences, regardless of it being a send or receive stream. It extracts information about that stream it finds to put into the input struct. It creates the handles to event related fields so the application can be aware of changes of the ASIO driver state. The mLAN driver sets the state of the ASIO driver to be active.

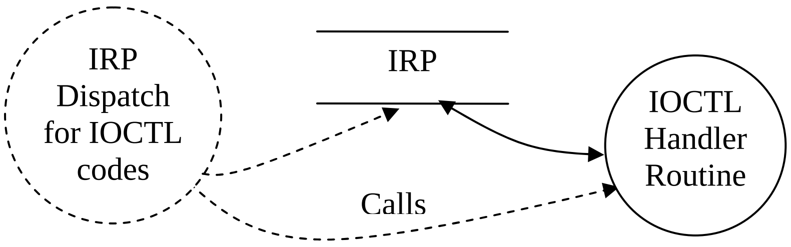


IOCTL_MLAN_ASIO_ALLOC

Input Buffer:
The input buffer takes a `MLAN_ASIO_ALLOC_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request will allocate the requested number of channels for use by the ASIO client sitting in user mode. The sequences will be identified as ASIO sequences, and input and output buffers will be allocated.

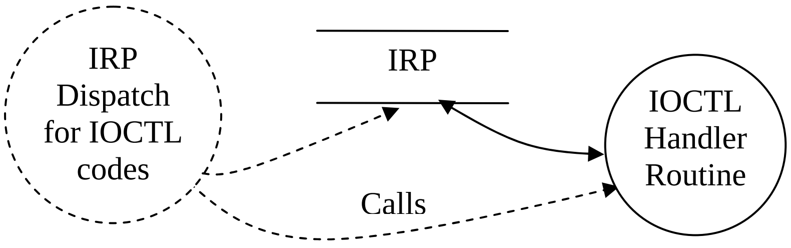


IOCTL_MLAN_ASIO_FREE

Input Buffer:
The input buffer takes a `MLAN_ASIO_COMMAND_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request frees up any previously allocated buffers and ASIO data structures, and removes the sequences identified as ASIO sequences. These sequences can now be used as any other type of sequence.

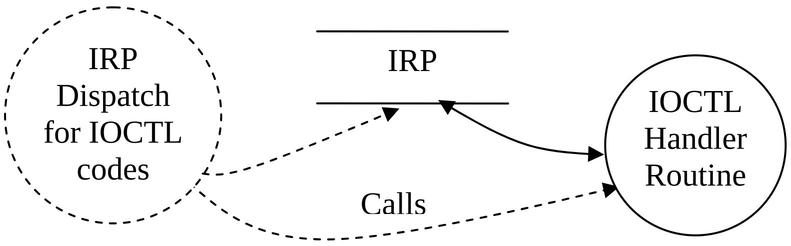


IOCTL_MLAN_ASIO_START

Input Buffer:
The input buffer takes a `MLAN_ASIO_COMMAND_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request starts ASIO processing, synchronising with the ASIO driver.

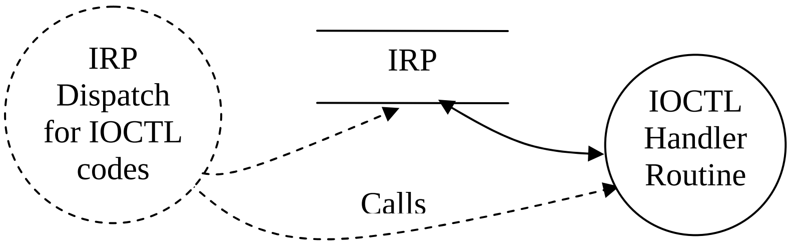


IOCTL_MLAN_ASIO_STOP

Input Buffer:
The input buffer takes a `MLAN_ASIO_COMMAND_PARAM` struct.

Output Buffer:
The same struct is returned.

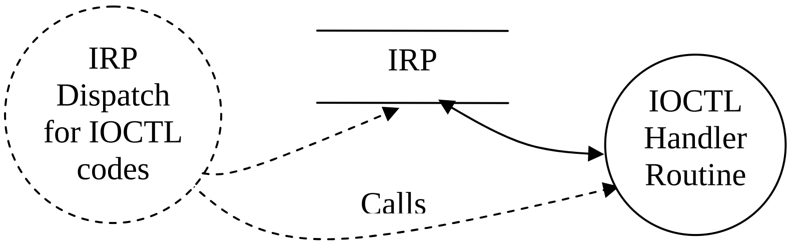
Textual Description:
The request stops ASIO processing, synchronising with the ASIO driver.



IOCTL_MLAN_ASIO_CLEANUP
Input Buffer:
The input buffer takes a `MLAN_ASIO_COMMAND_PARAM` struct.

Output Buffer:
The same struct is returned.

Textual Description:
The request removes event objects generated by ASIO processing, and limits the access of the ASIO client in user mode.

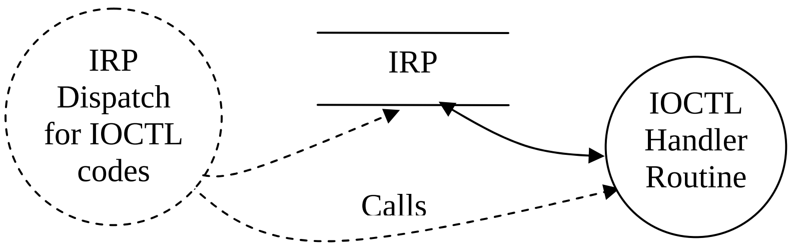


IOCTL_MLAN_ASIO_RESET

Input Buffer:
There is nothing for input.

Output Buffer:
There is nothing for output.

Textual Description:
The request will specify an ASIO reset event which will be handled when it is convenient.



Obsolete IOCTL codes

IOCTL_MLAN_GET_ADDR_FROM_DEVICE_OBJECT

Input Buffer:
N/A

Output Buffer:
N/A

Textual Description:
This is not implemented.

IOCTL_MLAN_GET_SPEED_TOPOLOGY_MAPS

Input Buffer:
Pointer to a `GET_SPEED_TOPOLOGY_MAPS` struct

Output Buffer:
If it does work, then it returns the same struct

Textual Description:
The IOCTL code handler uses an underlying request to the 1394 bus driver that is obsolete from Windows 2000 onwards.

B. Analysis and documentation of data structures used:

The structs shown below may contain other structs, refer from page 101 onwards to view their contents.

IEEE 1394 based structs

ALLOCATE_ADDRESS_RANGE

```
typedef struct _ALLOCATE_ADDRESS_RANGE {
    IN ULONG            fulAllocateFlags;
    IN ULONG            fulFlags;
    IN ULONG            nLength;
    IN ULONG            MaxSegmentSize;
    IN ULONG            fulAccessType;
    IN ULONG            fulNotificationOptions;
    OUT ADDRESS_OFFSET  Required1394Offset;
    OUT HANDLE          hAddressRange;
    IN UCHAR            Data[1];
} ALLOCATE_ADDRESS_RANGE, *PALLOCATE_ADDRESS_RANGE;
```

fulAllocateFlags
Contains a flag that indicates the type of address scheme.
fulFlags
Specifies whether the address ranges use big-endian byte order.
nLength
Specifies the number of the IEEE 1394 addresses to allocate.
MaxSegmentSize
Specifies the maximum size for each range of addresses the bus driver allocates.
fulAccessType
Specifies access type using one or more of the following flags.
fulNotificationOptions
Specifies which asynchronous I/O request types will trigger the bus driver to the notify the device driver upon completion.
Required1394Offset
Specifies a hard-coded address in the computer's IEEE 1394 address space.
hAddressRange
Specifies the handle to use when freeing the allocated address ranges.
Data[1]
Provides the beginning address of the backing store for the allocated address range. The IEEE 1394 bus driver maps all asynchronous requests to this address space.
[MSDN, 2005]

ADDRESS_RANGE_NOTIFY

```
typedef struct _ADDRESS_RANGE_NOTIFY{
      IN HANDLE         hAddressRange;
      OUT ULONG         ulOffset;
      OUT ULONG         fulNotificationOptions;
      IN OUT ULONG      nLength;
      OUT NODE_ADDRESS  NodeAddress;
      OUT UCHAR         Data[1];
} ADDRESS_RANGE_NOTIFY, *PADDRESS_RANGE_NOTIFY;
```

hAddressRange
Specifies the handle to the allocated address range.
ulOffset
Specifies the offset for the notification.
fulNotificationOptions
Specifies the reason for the notification.
nLength
Specifies the length for the notification.
NodeAddress
The address of the source node.
Data[1]
Returns a fresh copy of the data from the notification IRP.

SET_ADDRESS_DATA

```
typedef struct _SET_ADDRESS_DATA {
    IN HANDLE          hAddressRange;
    IN ULONG           nLength;
    IN ULONG           ulOffset;
    IN UCHAR           Data[1];
} SET_ADDRESS_DATA, *PSET_ADDRESS_DATA;
```

hAddressRange
Contains a handle that specifies the IEEE 1394 addresses to which the data is written.
nLength
Indicates the length in bytes of the data in the buffer specified by the **Data** member.
ulOffset
Indicates the offset into the buffer at **Data** where the data is located.
Data
Contains a data buffer that holds the data that the SetAddressData writes to the IEEE 1394 address range specified in member **hAddressRange**.
[MSDN, 2005]

GET_ADDRESS_DATA

```
typedef struct _GET_ADDRESS_DATA {
    IN HANDLE          hAddressRange;
    IN ULONG           nLength;
    IN ULONG           ulOffset;
    OUT NODE_ADDRESS   NodeAddress;
    OUT UCHAR          Data[1];
} GET_ADDRESS_DATA, *PGET_ADDRESS_DATA;
```

hAddressRange
Contains a handle that specifies the IEEE 1394 addresses where the data to retrieve is stored.

nLength

Indicates the length in bytes of the data in the buffer specified by the **Data** member.

ulOffset

Indicates the offset into the buffer at **Data** where the retrieved data is located.

Data

Contains a data buffer where the GetAddressData routine stores the data retrieved from the IEEE 1394 address range specified in member **hAddressRange**.

[MSDN, 2005]

ASYNC_LOCK

```
typedef struct _ASYNC_LOCK {
    IN ULONG           bRawMode;
    IN ULONG           bGetGeneration;
    IN IO_ADDRESS      DestinationAddress;
    IN ULONG           nNumberOfArgBytes;
    IN ULONG           nNumberOfDataBytes;
    IN ULONG           fulTransactionType;
    IN ULONG           fulFlags;
    IN ULONG           Arguments[2];
    IN ULONG           DataValues[2];
    IN ULONG           ulGeneration;
    IN ULONG           Buffer[2];
} ASYNC_LOCK, *PASYNC_LOCK;
```

bRawMode

Indicates, if TRUE, that bus and node number specified in the **DestinationAddress** will be used. If FALSE, the values in **DestinationAddress** are ignored. This flag must be set to TRUE when the caller sends an asynchronous read request to a virtual diagnostic driver (1394vdev.sys).

bGetGeneration

Indicates, when TRUE, that the generation count will automatically be set to the current generation count. If FALSE, the **ulGeneration** member holds the generation count.

DestinationAddress

Contains a structure of type **IO_ADDRESS** that specifies the 1394 64-bit destination address for this read operation. Caller must fill in the **IA_Destination_Offset** member of structure. The bus driver fills in the **IA_Destination_ID** member.

nNumberOfArgBytes

Specifies the number of argument bytes used in performing this lock operation. May be zero, 4 or 8. See the **fulTransactionType** member for details

nNumberOfDataBytes

Specifies the number of data bytes used in performing this lock operation. May be 4 or 8. See the **fulTransactionType** member for details.

fulTransactionType

Specifies which atomic transaction to execute on the 1394 node.

nBlockSize
Specifies the size of each individual block within the data stream. If the value assigned to this parameter is zero, the bus driver breaks up the stream into packets that are the maximum packet size.

fulFlags
Specifies the settings for this operation that are different from the default settings.

Arguments
Contains a bitmap that indicates which bits to change in **DestinationAddress**.

DataValues
Contains the bit values to be assigned to **DestinationAddress** in the bit positions specified by the bitmask in **Arguments**.

ulGeneration
Specifies the bus reset generation count. If the generation count specified does not match the actual generation of the bus, this request is returned with a status of STATUS_INVALID_GENERATION.

Buffer
Contains a buffer to store the results of the operation. Caller must ensure that the buffer contains at least **nNumberOfDataBytes** bytes of data.

[MSDN, 2005]

ASYNC_READ

```
typedef struct _ASYNC_READ {
    IN ULONG           bRawMode;
    IN ULONG           bGetGeneration;
    IN IO_ADDRESS      DestinationAddress;
    IN ULONG           nNumberOfBytesToRead;
    IN ULONG           nBlockSize;
    IN ULONG           fulFlags;
    IN ULONG           ulGeneration;
    OUT UCHAR          Data[1];
} ASYNC_READ, *PASYNC_READ;
```

bRawMode
Indicates, if TRUE, that bus and node number specified in the **DestinationAddress** will be used. If FALSE, the values in **DestinationAddress** are ignored. This flag must be set to TRUE when the caller sends an asynchronous read request to a virtual diagnostic driver (1394vdev.sys).

bGetGeneration
Indicates, when TRUE, that the generation count will automatically be set to the current generation count. If FALSE, the generation count will be taken from the **ulGeneration** variable.

DestinationAddress
Contains a structure of type **IO_ADDRESS** that specifies the 1394 64-bit destination address for this read operation. Caller must fill in the **IA_Destination_Offset** member of structure. The bus driver fills in the **IA_Destination_ID** member.

nNumberOfBytesToRead
Specifies the number of bytes to read.

nBlockSize
Specifies the size of each individual block within the data stream. If the value assigned to this parameter is zero, the bus driver breaks up the stream into packets that are the maximum packet size.

fulFlags
Specifies the settings for this operation that are different from the default settings.

ulGeneration
Specifies the bus reset generation count. If the generation count specified does not match the actual generation of the bus, this request is returned with a status of STATUS_INVALID_GENERATION.

Data
Contains a buffer to store the results of the asynchronous read operation. Caller must ensure that the buffer contains at least **nNumberOfBytesToRead** bytes of data.

[MSDN, 2005]

ASYNC_WRITE

```
typedef struct _ASYNC_WRITE {
    IN ULONG           bRawMode;
    IN ULONG           bGetGeneration;
    IN IO_ADDRESS      DestinationAddress;
    IN ULONG           nNumberOfBytesToWrite;
    IN ULONG           nBlockSize;
    IN ULONG           fulFlags;
    IN ULONG           ulGeneration;
    IN UCHAR           Data[1];
} ASYNC_WRITE, *PASYNC_WRITE;
```

bRawMode
Indicates, if TRUE, that bus and node number specified in the **DestinationAddress** will be used. If FALSE, the values in **DestinationAddress** are ignored.

bGetGeneration
Indicates, when TRUE, that the generation count will automatically be set to the current generation count. If FALSE, the generation count will be taken from the **ulGeneration** variable.

DestinationAddress
Contains a structure of type **IO_ADDRESS** that specifies the 1394 64-bit destination address for this write operation. Caller must fill in the **IA_Destination_Offset** member of structure. The bus driver fills in the **IA_Destination_ID** member.

nNumberOfBytesToWrite
Specifies the number of bytes to write.

nBlockSize
Specifies the size of each individual block within the data stream. If the value assigned to this parameter is zero, the bus driver breaks up the stream into packets that are the maximum packet size.

fulFlags

Specifies the settings for this operation that are different from the default settings.

ulGeneration

Specifies the bus reset generation count. If the generation count specified does not match the actual generation of the bus, this request is returned with a status of STATUS_INVALID_GENERATION.

Data

Contains a buffer to store the results of the asynchronous write operation. Caller must ensure that the buffer contains at least **nNumberOfBytesToRead** bytes of data.

[MSDN, 2005]

MLAN_BUS_RESET_NOTIFY

```
typedef struct _MLAN_BUS_RESET_NOTIFY{
      in NODE_ADDRESS  LocalNodeAddress;
}MLAN_BUS_RESET_NOTIFY, *PMLAN_BUS_RESET_NOTIFY;
```

LocalNodeAddress

The address of the node that should be notified of the bus reset event.

GET_LOCAL_HOST_INFORMATION

```
typedef struct _GET_LOCAL_HOST_INFORMATION {
    OUT ULONG          Status;
    IN ULONG           nLevel;
    IN ULONG           ulBufferSize;
    IN OUT UCHAR       Information[1];
} GET_LOCAL_HOST_INFORMATION, *PGET_LOCAL_HOST_INFORMATION;
```

Status

Indicates, on output, the status code set by the **GetLocalHostInformation** routine.

nLevel

Specifies what level of information requested in this call.

ulBufferSize

Indicates the size in bytes of the buffer at **Information**.

Information

Points to a buffer where the **GetLocalHostInformation** routine stores the local host information.

[MSDN, 2005]

GET_SPEED_TOPOLOGY_MAPS

```
typedef struct _GET_SPEED_TOPOLOGY_MAPS {
      OUT SPEED_MAP     speedMap;
      OUT TOPOLOGY_MAP  topologyMap;
} GET_SPEED_TOPOLOGY_MAPS, *PGET_SPEED_TOPOLOGY_MAPS;
```

speedMap

Stores an IEEE 1394 bus speed map.

topologyMap

Used to store an IEEE 1394 bus topology map.

PHY_CONFIGURATION_PACKET

```
typedef struct _PHY_CONFIGURATION_PACKET {
    IN ULONG            PCP_Phys_ID:6;
    IN ULONG            PCP_Packet_ID:2;
    IN ULONG            PCP_Gap_Count:6;
    IN ULONG            PCP_Set_Gap_Count:1;
    IN ULONG            PCP_Force_Root:1;
    IN ULONG            PCP_Reserved1:8;
    IN ULONG            PCP_Reserved2:8;
    IN ULONG            PCP_Inverse;
} PHY_CONFIGURATION_PACKET, *PPHY_CONFIGURATION_PACKET;
```

PCP_Phys_ID

Specifies the node address of the root. This member contains bits 0-5 of byte 0 of the packet.

PCP_Packet_ID

This member must be PHY_PACKET_ID_CONFIGURATION to indicate it is a PHY configuration packet. This member contains bits 6-7 of byte 0 of the packet.

PCP_Gap_Count

If the **PCP_Set_Gap_Count** bit is set, the PHY register gap_count field is set to this value. This member contains bits 0-5 of byte 1 of the packet.

PCP_Set_Gap_Count

If this bit is set, the PHY register gap_count field is set to **PCP_Gap_Count**. This member contains bit 6 of byte 1 of the packet.

PCP_Force_Root

If set, the caller becomes the root node. This member contains bit 7 of byte 1 of the packet.

PCP_Reserved1

Reserved. This member contains bits 0-7 of byte 2 of the packet.

PCP_Reserved2

Reserved. This member contains bits 0-7 of byte 3 of the packet.

PCP_Inverse

Specifies the logical inverse of the first quadlet of the packet.

[MSDN, 2005]

GET_LOCAL_NODE_ADDRESS

```
typedef struct _GET_LOCAL_NODE_ADDRESS{
      OUT NODE_ADDRESS  LocalNodeAddress;
}GET_LOCAL_NODE_ADDRESS, *PGET_LOCAL_NODE_ADDRESS;
```

LocalNodeAddress

Specifies the 10-bit bus number and 6-bit node number that serve as the node address for a 1394 node.

Isochronous Stream based structs

MLAN_ISOCH_PARAM

```
typedef struct _MLAN_ISOCH_PARAM {
      OUT HANDLE                 hStream;
      OUT MLAN_STREAM_STATUS     StreamStatus;
      OUT ULONG                  ErrorCode;
      OUT ULONG                  aBitmapClient[MLAN_MAX_NUM_SEQUENCES
/ ULONG_BITSIZE];
      IN MLAN_STREAM_CONFIG      mLANStreamConfig;
      IN MLAN_DATA_FIELD_CONFIG  mLANDataFieldConfig;
} MLAN_ISOCH_PARAM, *PMLAN_ISOCH_PARAM;
```

hStream
The handle to the stream resource that was allocted.
StreamStatus
The struct that contains the status of each sequence.
ErrorCode
The status code from the IOCTL.
aBitmapClient[MLAN_MAX_NUM_SEQUENCES / ULONG_BITSIZE]
Not sure.
mLANStreamConfig
A struct that contains specific information about that stream.
mLANDataFieldConfig
A struct that contains audio specific information about the sequences.

MLAN_STREAM_COMMAND_PARAM

```
typedef struct _MLAN_STREAM_COMMAND_PARAM {
      IN HANDLE        hStream;
      OUT ULONG        ErrorCode;
} MLAN_STREAM_COMMAND_PARAM, *PMLAN_STREAM_COMMAND_PARAM;
```

hStream
The handle to the allocated stream resource.
ErrorCode
The error code of the status from the completed IOCTL

IEC 61883 based structs

CMP_CONNECT

```
typedef struct _CMP_CONNECT {

    IN HANDLE                hOutputPlug;
    IN HANDLE                hInputPlug;
    IN CMP_CONNECT_TYPE      Type;
    IN CIP_DATA_FORMAT       Format;
    OUT HANDLE               hConnect;
} CMP_CONNECT, *PCMP_CONNECT;
```

hOutputPlug
Output plug handle.

hInputPlug;
Input plug handle.

Type
Requested connect type.

Format
Requested data format – transmission only.

hConnect
Returned connect handle.

CMP_CREATE_PLUG

```
typedef struct _CMP_CREATE_PLUG {

    IN CMP_PLUG_TYPE            PlugType;
    IN AV_PCR                   Pcr;
    IN PCMP_NOTIFY_ROUTINE      pfnNotify;
    IN PVOID                    Context;
    OUT ULONG                   PlugNum;
    OUT HANDLE                  hPlug;
} CMP_CREATE_PLUG, *PCMP_CREATE_PLUG;
```

PlugType
Type of plug to create.

Pcr
PCR Settings.

pfnNotify
Notification Routine for Register.

Context
Notification Context.

PlugNum
Plug Number.

hPlug
Plug Handle.

MLAN_CMP_PLUG_NOTIFY

```
typedef struct _MLAN_CMP_PLUG_NOTIFY {
      OUT ULONG         State;
      OUT ULONG         PlugNum;
      OUT ULONG         PlugType;
      OUT AV_PCR        Pcr;
} MLAN_CMP_PLUG_NOTIFY, *PMLAN_CMP_PLUG_NOTIFY;
```

State
Returns the state of the plug.

PlugNum
Plug number.

PlugType
Type of plug.

Pcr

PCR settings.

CMP_DELETE_PLUG

```
typedef struct _CMP_DELETE_PLUG {
    IN HANDLE                   hPlug;
} CMP_DELETE_PLUG, *PCMP_DELETE_PLUG;
```

hPlug

Plug Handle

CMP_DISCONNECT

```
typedef struct _CMP_DISCONNECT {
    IN HANDLE               hConnect;
} CMP_DISCONNECT, *PCMP_DISCONNECT;
```

hConnect

Connection handle to disconnect

MLAN_FCP_GET_REQUEST

```
typedef struct _MLAN_FCP_GET_REQUEST {
    OUT NODE_ADDRESS    NodeAddress;
    IN OUT ULONG        Length;
    IN OUT FCP_FRAME   Frame;
} MLAN_FCP_GET_REQUEST, *PMLAN_FCP_GET_REQUEST;
```

NodeAddress

The node address of the device that sent the get FCP request, provided it is a virtual device.

Length

The maximum available length of the frame payload, including FCP header.

Frame

Defines a function control protocol (FCP) request.

MLAN_FCP_GET_RESPONSE

```
typedef struct _MLAN_FCP_GET_RESPONSE {
    OUT NODE_ADDRESS    NodeAddress;
    IN OUT ULONG        Length;
    IN OUT FCP_FRAME   Frame;
} MLAN_FCP_GET_RESPONSE, *PMLAN_FCP_GET_RESPONSE;
```

NodeAddress

The node address of the device that sent the get FCP response, provided it is a virtual device.

Length

The maximum available length of the frame payload, including FCP header.

Frame

Defines a function control protocol (FCP) request.

CMP_GET_PLUG_HANDLE

```
typedef struct _CMP_GET_PLUG_HANDLE {
    IN ULONG                PlugNum;
    IN CMP_PLUG_TYPE        Type;
    OUT HANDLE              hPlug;
} CMP_GET_PLUG_HANDLE, *PCMP_GET_PLUG_HANDLE;
```

PlugNum
Requested plug number.
Type
Requested plug type.
hPlug
Returned plug handle.

CMP_GET_PLUG_STATE

```
typedef struct _CMP_GET_PLUG_STATE {
    IN HANDLE               hPlug;
    OUT ULONG               State;
    OUT ULONG               DataRate;
    OUT ULONG               Payload;
    OUT ULONG               BC_Connections;
    OUT ULONG               PP_Connections;
} CMP_GET_PLUG_STATE, *PCMP_GET_PLUG_STATE;
```

hPlug
The handle of the plug to retrieve state information.
State
The state of the plug.
DataRate
The data rate of the plug.
Payload
The payload size for the plug.
BC_Connections
The number of broadcast connections associated with the plug.
PP_Connections
The number of point-to-point connections associated with the plug.

MLAN_FCP_SEND_REQUEST

```
typedef struct _MLAN_FCP_SEND_REQUEST {
    IN NODE_ADDRESS     NodeAddress;
    IN ULONG            Length;
    IN FCP_FRAME       Frame;
} MLAN_FCP_SEND_REQUEST, *PMLAN_FCP_SEND_REQUEST;
```

NodeAddress
The node address of the device that sent the send FCP request, provided it is a virtual device.

Length

The maximum available length of the frame payload, including FCP header.

Frame

Defines a function control protocol (FCP) request.

MLAN_FCP_SEND_RESPONSE

```
typedef struct _MLAN_FCP_SEND_RESPONSE {
    IN NODE_ADDRESS     NodeAddress;
    IN ULONG            Length;
    IN FCP_FRAME       Frame;
} MLAN_FCP_SEND_RESPONSE, *PMLAN_FCP_SEND_RESPONSE;
```

NodeAddress

The node address of the device that sent the send FCP response, provided it is a virtual device.

Length

The maximum available length of the frame payload, including FCP header.

Frame

Defines a function control protocol (FCP) request.

SET_FCP_NOTIFY

```
typedef struct _SET_FCP_NOTIFY {
    IN ULONG            Flags;    // Flags
    IN NODE_ADDRESS     NodeAddress;    // Node Address
} SET_FCP_NOTIFY, *PSET_FCP_NOTIFY;
```

Flags

Specifies the notification requested by the driver.

NodeAddress

Reserved for use by the operating system.

CMP_SET_PLUG

```
typedef struct _CMP_SET_PLUG {
    IN HANDLE                 hPlug;
    IN AV_PCR                 Pcr;
} CMP_SET_PLUG, *PCMP_SET_PLUG;
```

hPlug

Plug handle.

Pcr

PCR settings.

MLAN_ASIO_INIT_PARAM

```
typedef struct _MLAN_ASIO_INIT_PARAM {
      OUT ULONG numOutChannelsAvailable[MLAN_MAX_SEND_STREAMS];
      OUT ULONG bitmapOutSequencesAvailable[MLAN_MAX_SEND_STREAMS]
      [MLAN_MAX_NUM_SEQUENCES / ULONG_BITSIZE];
      OUT ULONG numInChannelsAvailable[MLAN_MAX_RECEIVE_STREAMS];
      OUT ULONG bitmapInSequencesAvailable[MLAN_MAX_RECEIVE_STREAMS]
      [MLAN_MAX_NUM_SEQUENCES / ULONG_BITSIZE];
      OUT ULONG sampleRateBeingUsed;
      OUT ULONG minimumBufferSizePerChannel;
      OUT ULONG playbackBufferSizePerChannel;
      OUT ULONG recordBufferSizePerChannel;
      OUT HANDLE hBufferSwitchEvent;
      OUT HANDLE hDriverResetEvent;
      OUT ULONG sendBitSize;
      OUT ULONG recvBitSize;
      OUT BOOLEAN useIEEE32bitFloat;
      OUT ULONG errorCode;
      OUT ULONG *pNumEventsRequestedByDriver;
      OUT ULONG *pOffsetForClientBufferByDriver;
} MLAN_ASIO_INIT_PARAM, *PMLAN_ASIO_INIT_PARAM;
```

numOutChannelsAvailable
The number of ASIO output channels available.
bitmapOutSequencesAvailable
Not sure.
numInChannelsAvailable
The number of ASIO input channels available
bitmapInSequencesAvailablesampleRateBeingUsed
Not sure.
minimumBufferSizePerChannel
in samples - not bytes, 24bit data in 32bit containers, this is the mininum available buffersize.
playbackBufferSizePerChannel
in samples - used to report latency.
recordBufferSizePerChannel
in samples - usde to report latency.
hBufferSwitchEvent
A handle to a buffer switch event (received in ASIO_INIT).
hDriverResetEvent
A handle to the ASIO driver reset event (received in ASIO_INIT).
sendBitSize
Not sure.
recvBitSize
Not sure.
useIEEE32bitFloat
This will force the driver to report 32bit float as the format.
errorCode
The error code status reported by the IOCTL after it returns.
pNumEventsRequestedByDriver
(received in ASIO_INIT).
pOffsetForClientBufferByDriver
(received in ASIO_INIT).

MLAN_ASIO_ALLOC_PARAM

```
typedef struct _MLAN_ASIO_ALLOC_PARAM {
      IN ULONG          asioBufferSize;
      IN ULONG          outputBuffersRequiredMap_StreamIndex
      [MAX_ASIO_CHANNELS];
      IN ULONG          outputBuffersRequiredMap_SequenceIndex
      [MAX_ASIO_CHANNELS];
      OUT PULONG        pOutBuffers[MAX_ASIO_CHANNELS * 2];
      IN ULONG          inputBuffersRequiredMap_StreamIndex
      [MAX_ASIO_CHANNELS];
      IN ULONG          inputBuffersRequiredMap_SequenceIndex
      [MAX_ASIO_CHANNELS];
      OUT PULONG        pInBuffers[MAX_ASIO_CHANNELS * 2];
      OUT ULONG         errorCode;
} MLAN_ASIO_ALLOC_PARAM, *PMLAN_ASIO_ALLOC_PARAM;
```

asioBufferSize
This is the requested buffersize, should not be smaller than the size reported in INIT_PARAM
outputBuffersRequiredMap_StreamIndex
Specifies the requested output buffers to be used for streams.
outputBuffersRequiredMap_SequenceIndex
Specifies the requested output buffers to be used for sequence.
pOutBuffers
A pointer to the output buffer that has 24bit data in 32bit containers.
inputBuffersRequiredMap_StreamIndex
Specifies the requested input buffers to be used for streams.
inputBuffersRequiredMap_SequenceIndex
Specifies the requested input buffers to be used for sequences.
pInBuffers
A pointer to the input buffer that has 24bit data in 32bit containers.
errorCode
The error code specified by the completion of the IOCTL call.

MLAN_ASIO_COMMAND_PARAM

```
typedef struct _MLAN_ASIO_COMMAND_PARAM {
      ULONG errorCode;
} MLAN_ASIO_COMMAND_PARAM, *PMLAN_ASIO_COMMAND_PARAM;
```

errorCode
The error code returned by the completion of the IOCTL.

Sundry structs contained within main structs

ADDRESS_OFFSET

```
typedef struct _ADDRESS_OFFSET {
    USHORT              Off_High;
    ULONG               Off_Low;
} ADDRESS_OFFSET, *PADDRESS_OFFSET;
```

Off_High
Specifies the high order offset for a IEEE 1394 address.

Off_Low

Specifies the low order offset for a IEEE 1394 address.
[MSDN, 2005]

IO_ADDRESS

```
typedef struct _IO_ADDRESS {
    NODE_ADDRESS        IA_Destination_ID;
    ADDRESS_OFFSET      IA_Destination_Offset;
} IO_ADDRESS, *PIO_ADDRESS;
```

IA_Destination_ID
Holds a structure of type **NODE_ADDRESS** containing the destination node address.
IA_Destination_Offset
Holds a structure of type **ADDRESS_OFFSET** that specifies the index of the 1394 address within the address array.
[MSDN, 2005]

SPEED_MAP

```
typedef struct _SPEED_MAP {
    USHORT              SPD_Length;
    USHORT              SPD_CRC;
    ULONG               SPD_Generation;
    UCHAR               SPD_Speed_Code[4032];
} SPEED_MAP, *PSPEED_MAP;
```

SPD_Length
Specifies the number of quadlets in the speed map.
SPD_CRC
Specifies the CRC value for the speed map.
SPD_Generation
Specifies the generation count for the bus reset that corresponds to this speed map.
SPD_Speed_Code
Specifies an array of speed codes.
[MSDN, 2005]

TOPOLOGY_MAP

```
typedef struct _TOPOLOGY_MAP {
    USHORT              TOP_Length;
    USHORT              TOP_CRC;
    ULONG               TOP_Generation;
    USHORT              TOP_Node_Count;
    USHORT              TOP_Self_ID_Count;
    SELF_ID             TOP_Self_ID_Array[];
    SELF_ID             TOP_Self_ID_Array[4032];
} TOPOLOGY_MAP, *PTOPOLOGY_MAP;
```

TOP_Length

Specifies the length in quadlets of the topology map.

TOP_CRC

Specifies the CRC value for the topology map.

TOP_Generation

Specifies the bus reset generation for which the topology map was created.

TOP_Node_Count

Specifies the number of nodes in the topology map.

TOP_Self_ID_Count

Specifies the number of entries in **TOP_Self_ID_Array**. (Not used)

TOP_Self_ID_Array

Pointer to an array of **SELF_ID** and **SELF_ID_MORE** structures (the two structures are the same size).

[MSDN, 2005]

MLAN_STREAM_CONFIG

```
typedef struct _MLAN_STREAM_CONFIG {
      ULONG StreamType;
      ULONG Speed;
      ULONG Channel;
      ULONG NumBufferGroups;
      ULONG NumPacketsPerBufferGroup;
} MLAN_STREAM_CONFIG, *PMLAN_STREAM_CONFIG;
```

StreamType

Specifies the type of the stream.

Speed

Specifies the speed of the bus.

Channel

Specifies the channel the stream belongs to.

NumBufferGroups

Specifies the number of buffers.

NumPacketsPerBufferGroup

Specifies the number of packets per group of buffers.

MLAN_STREAM_STATUS

```
typedef struct  _MLAN_STREAM_STATUS{
      BOOLEAN bRunning;
      ULONG   ReceivingSampleRate;
      BOOLEAN receivingMatch;
      ULONG   NumDBCErrorPacket;
      ULONG   NumSequences;
      ULONG streamIndex;
      MLAN_SEQUENCE_STATUS  SequenceStatus[MLAN_MAX_NUM_SEQUENCES];
} MLAN_STREAM_STATUS, *PMLAN_STREAM_STATUS;
```

bRunning

Specifies whether the stream is running or not.

ReceivingSampleRate

Specifies the sample rate when receiving packets.

receivingMatch

Specifies whether the stream type required matches the stream type received

NumDBCErrorPacket

Specifies the number of Data Block Count error packets.

NumSequences

Specifies the number of sequences the stream has.

streamIndex

Specifies the stream index in relation to the rest of the active streams.

SequenceStatus[MLAN_MAX_NUM_SEQUENCES]

Specifies the individual status of each sequence.

MLAN_DATA_FIELD_CONFIG

```
typedef struct _MLAN_DATA_FIELD_CONFIG{
      ULONG SampleRate;
      BOOLEAN useIEEE32bitFloat;
      ULONG bitDepthForWDMAudio;
      MLAN_SYT_LOCAL_OR_EXT SytParam;
      ULONG NumSequences;
      MLAN_SEQUENCE_DATA_CONFIG SequenceData[MLAN_MAX_NUM_SEQUENCES];
} MLAN_DATA_FIELD_CONFIG, *PMLAN_DATA_FIELD_CONFIG;
```

SampleRate

Specifies the sample rate of the isochronous stream.

useIEEE32bitFloat

Specifies whether the data is represented in IEEE 1394 32bit float format or not.

bitDepthForWDMAudio

Specifies the resolution of the WDM audio sample.

SytParam

Specifies the stream that is bus master.

NumSequences

Specifies the number of sequences the stream has.

SequenceData[MLAN_MAX_NUM_SEQUENCES]

Specifies the sequence type and corresponding allocated device number.

MLAN_SEQUENCE_STATUS

```
typedef struct  _MLAN_SEQUENCE_STATUS{
      MLAN_SEQUENCE_DATA_CONFIG SequenceData;
      ULONG   AudioSampleSize;
      BOOLEAN    bCopyProtect;
      ULONG   NumParityError;
} MLAN_SEQUENCE_STATUS, *PMLAN_SEQUENCE_STATUS;
```

SequenceData

Specifies the sequence type and corresponding allocated device number.

AudioSampleSize

Specifies the size of each audio sample.

bCopyProtect

Specifies whether the sequence data can be overwritten or not.

NumParityError

Records the number of parity errors.

MLAN_SYT_LOCAL_OR_EXT

```
typedef struct _MLAN_SYT_LOCAL_OR_EXT {
      BOOLEAN bMaster;
      HANDLE hStream;
} MLAN_SYT_LOCAL_OR_EXT, *PMLAN_SYT_LOCAL_OR_EXT;
```

bMaster
Specifies if the owner of the stream is the bus master or not.

hStream
Specifies the handle to the stream resource that is the bus master.

MLAN_SEQUENCE_DATA_CONFIG

```
typedef struct _MLAN_SEQUENCE_DATA_CONFIG{
      ULONG SequenceType;
      ULONG DeviceNumber;
} MLAN_SEQUENCE_DATA_CONFIG, *PMLAN_SEQUENCE_DATA_CONFIG;
```

SequenceType
The number of the sequence.

DeviceNumber
The device number associated with the sequence. The value is valid from number 1 to the number of allocated devices.

CMP_CONNECT_TYPE

```
typedef enum {
    CMP_Broadcast = 0,
    CMP_PointToPoint
} CMP_CONNECT_TYPE;
```

An enumeration of the connect type used in connecting 61883 plugs.

CIP_DATA_FORMAT

```
typedef struct _CIP_DATA_FORMAT {
    UCHAR        FMT;
    UCHAR        FDF_hi;
    UCHAR        FDF_mid;
    UCHAR        FDF_lo;
    BOOLEAN      bHeader;
    UCHAR        Padding;
    UCHAR        BlockSize;
    UCHAR        Fraction;
    ULONG        BlockPeriod;
} CIP_DATA_FORMAT, *PCIP_DATA_FORMAT;
```

FMT
This is the data type used in the CIP. FMT and FDF are either known, or discovered.

FDF_hi
    Ascertained via AV/C command.
FDF_mid
    Not sure.
FDF_lo
    Not sure.
bHeader
    SPH (whether a transport time delay stamp has been added) as defined by IEC-61883.
Padding
    QPC (quadlet padding count) as defined by IEC-61883.
BlockSize
    DBS (data block size) as defined by IEC-61883.
Fraction
    FN (number of fractions the original packet was divided into) as defined by IEC-61883.
BlockPeriod
    BlockPeriod - transmission only.

CMP_PLUG_TYPE

```
typedef enum {
    CMP_PlugOut = 0,
    CMP_PlugIn
} CMP_PLUG_TYPE;
```

An enumeration of the plug type.

AV_PCR

```
typedef struct _AV_PCR {
    union {
        OPCR    oPCR;
        IPCR    iPCR;
        ULONG   ulongData;
    };
} AV_PCR, *PAV_PCR;
```

oPCR
    Contains an **OPCR** structure that contains initialization values for an output plug.
iPCR
    Contains an **IPCR** structure that contains initialization values for an input plug.
ulongData
    Reserved for internal use.
[MSDN, 2005]

IPCR

```
typedef struct _IPCR {
    ULONG   Reserved0:16;
    ULONG   Channel:6;
    ULONG   Reserved1:2;
    ULONG   PPCCounter:6;
    ULONG   BCCCounter:1;
    ULONG   OnLine:1;
} IPCR, *PIPCR;
```

Payload
Specifies the connection speed.
OverheadID
Specifies, for an unconnected output plug, the upper bounds for the bandwidth that the output plug needs for the transmission of an isochronous packet.
DataRate
Indicates the bit rate that the output plug uses to transmit an isochronous packet.
Channel
Indicates the channel number that the output plug shall use to transmit the isochronous data flow, for a suspended output plug. For an active output plug it indicates the actual channel number that the output plug uses to transmit the isochronous data flow. For an unconnected output plug it has no meaning.
Reserved
Reserved.
PPCCounter
Indicates the number of point-to-point connections to the output plug.
BCCCounter
Indicates, when one, that there is a broadcast-out connection to the output plug. When zero it indicates that there is no connection.
OnLine
Indicates, when one, that the corresponding output plug is on-line. When zero it indicates that the output plug is off-line.
[MSDN, 2005]

OPCR

```
typedef struct _OPCR {
    ULONG   Payload:10;
    ULONG   OverheadID:4;
    ULONG   DataRate:2;
    ULONG   Channel:6;
    ULONG   Reserved:2;
    ULONG   PPCCounter:6;
    ULONG   BCCCounter:1;
    ULONG   OnLine:1;
} OPCR, *POPCR;
```

Reserved0
Reserved.
Channel
Indicates the channel number that the input plug shall use to transmit the isochronous data flow, for a suspended input plug. For an active input plug it indicates the actual channel number that the input plug uses to transmit the isochronous data flow. For an unconnected input plug it has no meaning.

Reserved1
Reserved.

PPCCounter
Indicates the number of point-to-point connections to the input plug.

BCCCounter
Indicates, when one, that there is a broadcast-in connection to the input plug. When zero it indicates that there is no connection.

OnLine
Indicates, when one, that the corresponding input plug is on-line. When zero it indicates that the input plug is off-line.

[MSDN, 2005]

FCP_FRAME

```
typedef struct _FCP_FRAME {
    UCHAR                   ctype:4;
    UCHAR                   cts:4;
    UCHAR                   payload[511];
} FCP_FRAME, *PFCP_FRAME;
```

ctype
The command or response type as defined by the Command/Transaction Set (CTS) used for this request.

cts
The CTS used for this FCP request. The CTS specifies the command set, the structure of the command/response field, and the rules of transactions used for sending FCP commands and responses.

payload
The FCP request for this frame.

ERR_MLAN_STREAM

```
typedef enum{
      ERR_MLAN_SUCCESS = 0,
      ERR_MLAN_STREAM_EXIST,
      ERR_MLAN_BAND_OVER,
      ERR_MLAN_CH_USED,
      ERR_MLAN_NO_RES,
      ERR_MLAN_STREAM_NOT_ALLOC,
      ERR_MLAN_STREAM_RUNNING,
      ERR_MLAN_FS_MISMATCH,
      ERR_MLAN_CANNOT_GET_SYT
}ERR_MLAN_STREAM_E;
```

This is an enumeration of the error codes used in some of the data structures. Should an IOCTL call fail, depending on the IOCTL, it will return an error code.

ERR_ASIO_MLAN

```
typedef enum{
      ERR_ASIO_MLAN_SUCCESS = 0,
      ERR_ASIO_MLAN_NOMEM,
      ERR_ASIO_MLAN_NORES,
      ERR_ASIO_MLAN_NOTSETUP,
      ERR_ASIO_MLAN_ALREADY_INITIALIZED,
      ERR_ASIO_MLAN_NOTRUNNING,
      ERR_ASIO_MLAN_NOTINITIALIZED,
      ERR_ASIO_MLAN_ALREADYRUNNING,
      ERR_ASIO_MLAN_ALREADYALLOCATED,
      ERR_ASIO_MLAN_NOTALLOCATED,
      ERR_ASIO_MLAN_INVALID_BUFFER_SIZE,
}ERR_ASIO_MLAN_E;
```

An enumeration of the error codes that are produced by ASIO related IOCTL calls.