RHODES UNIVERSITY
Where leaders learn

Department of Computer Science
2006

Writing a Yamaha mLAN™ Bus Driver using the Microsoft® Windows® Driver Foundation

A thesis submitted in partial fulfilment of the requirements for the degree of Bachelor of Science with Honours in Computer Science

by
Grant Carver

Supervised by
Prof. Richard Foss

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
This report is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation.
mLAN and mLAN are trademarks of the YAMAHA Corporation.

Abstract

The current mLAN™ PC driver for Microsoft® Windows® was developed according to the Windows® Driver Model (WDM). With the introduction of the Windows® Driver Foundation (WDF) a new simplified model and framework for driver development was provided. This new model provides a significant improvement in the ease of development of WDM compliant kernel mode drivers as well as a simplification of driver design. In implementing the Yamaha mLAN™ Bus driver using the new framework we expect to show a simplification in the design of the driver and a reduction in the complexity and amount of code required to achieve the same functionality. With the introduction of Windows® Vista™, a new kernel mode audio streaming method will be introduced and we examine this as an alternative to the currently used ASIO streaming.

Acknowledgements

Thank you to Richard for his help, support, and understanding through the difficulties of driver development. Also, to the departmental sponsors for providing facilities for research. Finally my wife Jenny, who put up with a student for yet another year.

SPONSORED BY

Telkom Business Connexion COMVERSE tellabs VERSO TECHNOLOGIES

THRIP TECHNOLOGY AND HUMAN RESOURCES FOR INDUSTRY PROGRAMME STORTECH For Your Information

Contents

1 Introduction

Yamaha have provided a suite of tools and drivers to enable the use of their mLAN (music Local Area Network) technology in conjunction with Windows based PC's. These applications provide the functionality to use the Microsoft Windows based PC as both an Enabler for connection management and as a virtual mLAN node or device. To facilitate this functionality, an mLAN Bus virtual device driver that facilitates communication with the IEEE 1394 bus is required.

The Yamaha mLAN Bus Driver that is currently in use, is a Microsoft Windows Driver Model (WDM) based implementation that also makes use of a proprietary C++ based framework called DriverStudio (Miles, 2005). This report discusses the use of a new driver development model, the Microsoft Windows Driver Foundation (WDF), to implement a new version of the mLAN Bus driver specifically for intended use on the upcoming version of Microsoft Windows - Windows Vista. It examines the differences between the old and new implementations and then analyses any benefits gained in using the new model.

A second objective is to analyse the current mLAN implementation (referred to in this document as the mLAN System) and discuss the roles of each player therein. This is so an understanding of the original design intention could be gained, especially in respect to the use of the current mLAN Bus driver. We can then take this a step further and propose a new design that may provide a simpler, more flexible system to use and maintain.

Lastly we examine a new specification for kernel streaming to be provided in Microsoft Windows Vista. This may provide an alternative to the current methods of audio streaming used in the mLAN system, namely WDM Audio and MIDI Streaming, and ASIO Audio streaming.

To begin our discussion we will briefly examine the technologies involved. We will look at the underlying IEEE 1394 serial bus technology and how it is suited to digital audio transport. IEC 61883 is the basis behind mLAN, and we examine how it facilitates audio transport across the IEEE 1394 bus. Yamaha's mLAN system is then discussed with specific emphasis on the use of the underlying technology.

Following this we look at the models available for driver development – WDM and WDF – and discuss the differences and similarities between the two. This will leave us in the position to examine the current mLAN™ Bus driver and how it functions in the current mLAN™ System. We will consider why a bus driver is required and how it provides the required functionality.

We then are able to consider implementing the current mLAN Bus driver using the new WDF model. Here we discuss the design decisions made in the logical and physical layout of a new driver. We then examine in some detail, the implementation of each section of the driver.

To close the discussion we examine the results of implementing the mLAN Bus driver using the WDF model and how these observations can be used in future design consideration. In suggestions for further work in this area, we consider integrating the new real-time streaming technology present in Windows Vista into the new driver as an alternative PC based audio streaming option. We evaluate our ideas for re-designing a new mLAN PC System, and propose reasons for further work in this area. We can also at this stage consider possible changes to the entire mLAN System architecture in order to simplify the use and future maintenance thereof.

We conclude by summarizing the important aspects of the report and discuss the value of the information gained from the research and implementation performed.

2 mLAN™, IEEE 1394, and IEC 61883 Technologies

To understand the role of the mLAN Bus Driver within the system, it is necessary to take a brief look at the technologies used by the mLAN system. This will be approached by examining the underlying technology specifications and then examining how the mLAN system uses these to implement an mLAN network.

2.1 IEEE 1394 Technology (Firewire™)

In 1986 engineers at Apple Computer designed a new, high speed serial bus capable of running over simple cabling for use within the Macintosh system. Realising the promise of using the technology, which they called Firewire, for high speed peripheral connectivity, they presented the design to the industry for standardisation. This was standardised by the IEEE in 1995 and called the IEEE 1394 specification. This was updated in 2000 to the IEEE 1394 specification which clarified ambiguities in the original specification and since then an altered specification – IEEE 1394b – has been introduced which provides for higher speeds, greater distances and even choice in physical media (1394 Trade Association, 2006). We will confine this discussion to the IEEE 1394a standard.

Firewire is a high speed serial bus which provides both asynchronous and isochronous transmission using time-based multiplexing. It can run at speeds of 100Mbps, 200Mbps, and 400Mbps (in the case of 1394a), and 800Mbps and 1600Mbps (in the case of 1394b). (IEEE, 2002)

Physical connectivity is achieved via extremely well shielded twisted pair cabling that allows the high speed data transfer without noise interference. These have 4 or 6 pin connectors. 2 pairs of twisted pair carry data in either direction and on the 6 pin there are 2 power cables that provide up to 40V at up to 1.5A. Differential data transmission is used on the twisted pair meaning that the cables carry the opposite voltage and any external noise affects both signals equally. The difference between the two is then used to determine the logical state. This is similar to the method used in balanced analogue audio cabling to maintain a noise free line signal.

Although there are two data pairs, the bus is half-duplex as only one is used for data transmission. The other is used to synchronise the clock rate on devices by changing its logical state when the other doesn't. This is a technique called NRZ (Non-Return-Zero) with Data-Strobe (DS) encoding.

A tree topology is used with a root node being the root of the tree. A maximum of 63 nodes can be present on a single bus with a maximum of 16 hops between nodes. 1024 busses can be connected via bridges. The ability to use mLAN on multiple busses is not yet realised and so will not be discussed. The bus is entirely self-managed and a node is selected as the bus manager to perform power management and bus optimization. The root node is responsible for line arbitration between all the nodes.

Addressing on the bus is in line with the Control and Status Register (CSR) architecture defined in the IEEE 13213 standard. This means that the entire serial bus is seen as a block of memory and each node on the bus has a section of memory that is allocated to it. In IEEE 1394, a fixed 64 bit addressing method is used. Each address comprises a 10 bit Bus ID, a 6 bit Node ID, and a 48 bit Offset value. This provides an address space of 256 terabytes for each node. Part of this address space is used for registers that provide CSR information to conform to the CSR architecture, the rest is used to implement the functionality of the device. Any communication with the node is done via asynchronous reads and writes to well known offsets within its address space.

Asynchronous communication is always a 2 way transaction. A read with a response, or a write with a response(Foss, 2006). Asynchronous packets are sent when possible and are addressed to a specific target node. On the other hand, isochronous transmission is not addressable and is only a write action. Isochronous packets are sent at regular intervals called cycles, which makes isochronous transmission deterministic. Nodes are configured using asynchronous packets to listen to an isochronous channel which removes the need to address each packet, and permits a channel to be listened to by multiple nodes. Asynchronous packets have a lower priority than the isochronous packets and will be delayed until the isochronous cycle has completed.

To permit isochronous transmission, 2 aspects need to be catered for: 1) there needs to be a cycle-master node that initiates every isochronous transmission cycle at a rate of 8 kHz, and 2) a resource-manager node that allocates channels and bandwidth for isochronous transmission. This is required to ensure that all of the data can be transmitted within the time allowed for a cycle. These roles are contended for along with the bus manager role after a bus reset occurs. Bus resets occur as a request by a node, or with the addition or removal of a node.

Microsoft Windows now has standard support for IEEE 1394 busses and provides a class driver to allow communication in a standardised manner to any vendor-specific 1394 device driver.

2.2 IEC 61883 Technology

With the underlying transport mechanism provided by IEEE 1394, the need for a standardised manner of transporting the audio in an isochronous packet was identified. This was provided by the IEC 61883 series of standards which describe standards for the transmission of audio, video, and multimedia over IEEE 1394.

The first of the series, IEC 61883-1, describes the general packet format, data flow and connection management (Foss, 2006). This general packet format is called a CIP packet or Common Isochronous Packet, and is used as the payload of an isochronous packet. It comprises a header containing a description of the packet data and timing information, and a data section with a CRC value. As this packet format is used for a variety of multimedia transport, Yamaha recommended an approach which was accepted as the IEC 61883-6 specification – the Audio and Music Data Transmission Protocol.

This described the manner in which multiple audio and MIDI samples could be transported in a single CIP packet. This is necessary because the sample rates required for digital music are much higher than the 8 kHz rate of isochronous packet transmission. For example: a 48 kHz sample rate would require 6 samples in each CIP packet. Each of these samples is represented by a data block in the CIP data section and can have multiple simultaneous audio and MIDI samples within the block.

To differentiate between separate audio samples that occur at the same instance, the samples within the data block are arranged in a specific order. This is termed sequencing and allows a node to listen on a particular isochronous channel and a particular sequence or position in the data block for the audio samples directed to it. (Vienna Institute of Technology, 2001)

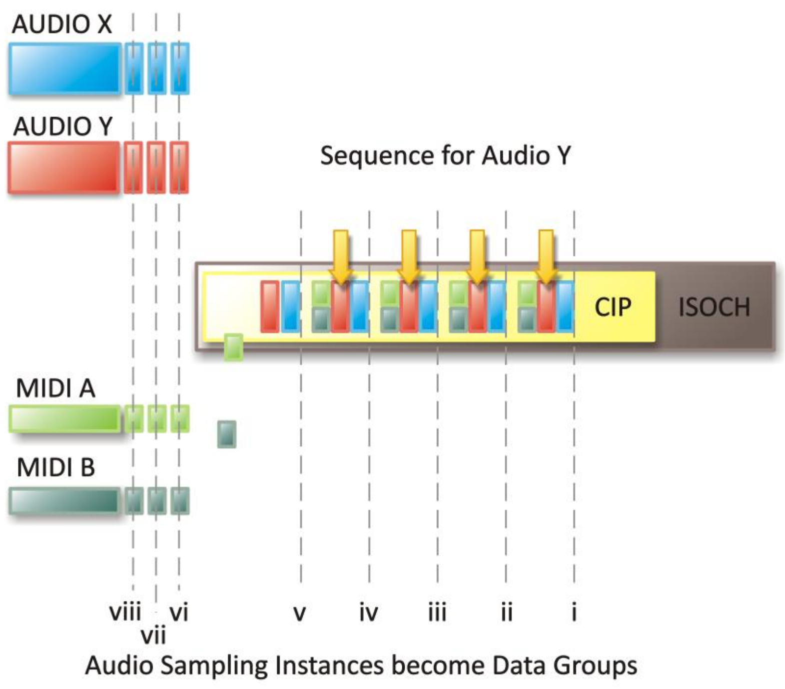


Figure 2-1 Sequencing in a Common Isochronous Packet

Figure 2-1 shows the sequencing of samples from different sources that occur at the same instance being grouped into data blocks in. Notice that the MIDI data is multiplexed onto a single sequence which has sufficient space for 8 MIDI channels.

2.3 Yamaha mLAN™ Technology

In a modern audio studio the connectivity between the myriad of different devices entails both a large number of leads, and a complex layout. Yamaha's mLAN technology, standing for Music Local Area Network, aims to simplify the connectivity and provide a standardised digital format for audio and MIDI data transfer. It provides two key qualities of transfer that are imperative for pro-audio use: low latency and deterministic transmission. (Foss, 2006)

These are important because of the large volume of data that pro-audio implementations need to transport; Pro-audio sampling of audio is now being performed at 96 kHz with a 24bit depth. This equates to the transmission of 288 kB/s for each stream. With an average studio there are many separate tracks of audio that need to be transported to and from devices resulting in a need for high bandwidth connectivity. IEEE1394 has provided these two qualities and has a simple connectivity and bus management model (Foss, 2006).

Because of the serial bus nature of IEEE1394, a manner of controlling how the audio data is transmitted and how the 'connections' are made between devices was required. This was provided by the IEC61883 specifications. (Vienna Institute of Technology, 2001)

mLAN abstracts the idea of plugs and connections a bit further. Each mLAN device on the network presents a number of input and output plugs. On the device side these can be connected to physical connectors or, as in an effects unit, be processed ready for immediate output on an alternative mLAN plug. Plugs can be of differing formats – i.e. MIDI or Audio, and are connected output to input, allowing data to be streamed over the connection.

In implementing mLAN plugs, Yamaha initially chose a distributed connection control architecture called mLAN Version 1. This was replaced by a more flexible arrangement of using a central Enabler to govern connection management in the subsequent mLAN Version 2. In the PC based mLAN System, we find an Enabler application that performs this role using the mLAN Bus driver to communicate with the devices. To allow communication with many different devices, the Enabler requires a Hardware Abstraction Layer (HAL) interface, in the form of a DLL plug-in, that provides a common set of functionality for each different device.(Foss, 2006)

Because of the need to provide a software HAL for each device, the Open Generic Transporter specification was introduced. This outlines a common Transport Control Interface that each device must implement in its address space(Foss, 2006). This allows communication with each device in the same manner and removes the need for the manufacturer to create a software HAL.

The Enabler uses asynchronous packets to manage the devices and the connectivity between them. Transmission of data over these connections is done via isochronous streaming using the CIP format specified in the IEC 61883 specification. (Vienna Institute of Technology, 2001)

2.4 Chapter Summary

In this chapter the technology used to implement the Yamaha mLAN system was examined. It was noted that FireWire provides the required low latency, high bandwidth, deterministic transmission of data required for digital audio transport. It also provides a self-managed, flexible, and robust single cable solution for connectivity.

The IEC 61883 specifications for transporting audio over FireWire was also discussed. This explained the process of packaging audio data into a common format for transport within an isochronous packet. Sequencing of audio samples within a Common Isochronous Packet was described including the inclusion of MIDI data by multiplexing multiple MIDI channels into a sequence.

Finally, the Yamaha mLAN system was described. The use of an Enabler application to govern connection management was explained. The use of software HAL plug-ins as well as the Open Generic Transporter architecture was explained.

3 Windows® Driver Development Models

To write a device driver for any operating system is a challenge and requires careful planning and a vast knowledge of the intricacies of the intended platform. Microsoft Windows is no exception. In this chapter we examine two driver development models provided by Microsoft. First there is the Windows Driver Model introduced to provide a common model and implementation across Windows 98 and Windows NT 4. Only released recently, the Windows Driver Foundation is based on the WDM but extends it and makes driver development a much less complicated task.

3.1 Windows® Driver Model

The Windows Driver Model (WDM) was defined to provide a common driver development model and paradigm for Windows 98, Windows 2000 and later Windows XP(Oney, 1999). It aimed to reduce the complexity of implementing a Plug ’n Play compatible device driver, and provide a specification for interacting with the operating system in a safe and predictable way. This is necessary as device drivers are loaded as part of the core OS system. Any bugs or irresponsible programming, especially with drivers that run in kernel-mode, can lead to overall system instability and a poor user experience(Cant, 1999).

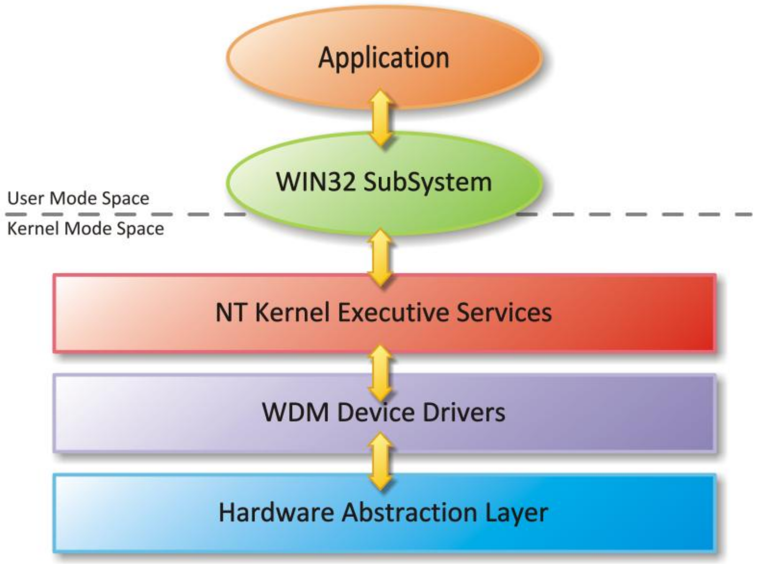


Figure 3-1 Microsoft Windows Driver System

A further feature implemented by the WDM is that of a driver stack. This allows layering and abstraction of those layers. For instance a USB device would make use of the USB class driver which implements functionality particular to the USB Bus. This driver would in turn make use of the underlying Hardware Abstraction Layer (HAL) that provides a common interface for all proprietary USB Bus implementations. This HAL would also in turn make use of a proprietary Bus Driver specific to the hardware. This allows the insertion of filter drivers to alter the functionality without affecting the underlying drivers.(Oney, 1999)

All WDM development is done in ANSI C and built in build environments provided in a Driver Development Kit provided by Microsoft(Microsoft Corporation (6), 2005). Each WDM driver must present a specified interface to the system and is required to implement it in its entirety. This results in the developer having to reproduce a large quantity of boiler-plate code that is often not directly related to the functionality of the device driver they are implementing.

A major concern of WDM driver writing is to provide consistent and predictable interaction with the OS. The Power management code, the PnP event managers, synchronisation and memory management are all left to the developer to control (Cant, 1999). Whilst this is very flexible, it is also very complicated.

Several proprietary solutions that provided a framework of utilities to assist the developer in implementing WDM were developed. These attempt to simplify the complexity of the system interaction. The current mLAN drivers were developed in such a framework called DriverWorks, a product created by Compuware(Miles, 2005).

DriverWorks provides a C++ implementation of libraries that facilitate the use of OOP in WDM and NT Kernel driver development. It provides libraries of common, reusable code and its own proprietary object model that functions on top of the WDM object model.

3.2 Windows® Driver Foundation

WDM achieved much in its time but was still a very complex model for developers to implement. Specifically, providing PnP capabilities entailed the use of a complex state-machine to allow differing actions according to the PnP and Power state. This complexity made writing safe and reliable drivers all the more difficult and caused developers to spend a large proportion of their time implementing boiler-plate code and not concentrating on the actual functionality of their device.(Microsoft Corporation (3), 2005)

The Windows Driver Foundation model is an evolution of the WDM model specifically aimed at reducing the complexity of device driver development(Microsoft Corporation (3), 2005). It does not replace the WDM model as such, and uses many similar principles and constructs, but it does provide a framework for implementation. It also re-examines the development life-cycle and has provision for device driver specific debugging and analysis(Microsoft Corporation (5), 2005).

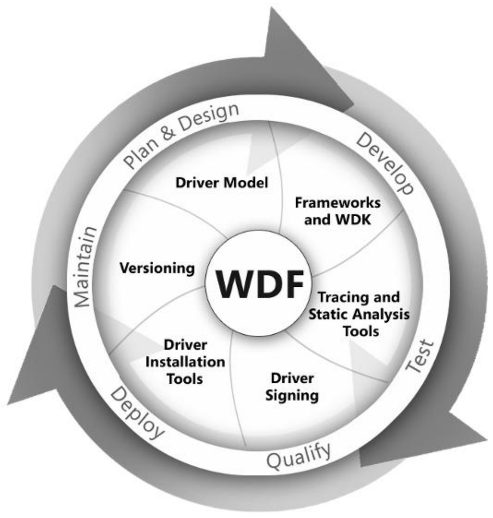


Figure 3-2 Microsoft Windows Driver Foundation Development Model

The diagram shown in Figure 3-2 shows the new development model. Note the inclusion of Driver Signing and Versioning in the life-cycle. This is specifically included because of the integral relationship of device drivers with the OS, and the need for quality control and testing to prevent unreliable drivers adversely affecting the system (Microsoft Corporation (4), 2006).

In many ways, the WDF is similar to the WDM in that it provides a design model, and implementation specification. The major difference is the fact that the emphasis in design is now on the functionality of the hardware device rather than on the inner workings of the OS(Microsoft Corporation (4), 2006). This is achieved by providing a framework that the driver can use to implement functionality and that also provides default functionality for that which the driver doesn't implement directly.

Consider the example of Power Event handling. In the WDM model, these were all required to be implemented by the driver – all 160 of the states had to be catered for. In the WDF model, only those of concern to the device are implemented (Microsoft Corporation (3), 2005).

3.2.1 KMDF & UMDF

The model provides two framework implementations specific to both Kernel-mode and User-mode. The ultimate goal is to move many device drivers that previously were required to run in kernel-mode into the user-mode space. (Microsoft Corporation (2), 2005)

The frameworks are in essence a very simple device driver implementation that can be customised to particular needs. This is done by presenting an object model with well known constructors, methods and properties that can be accessed by the developer. A diagram showing the relationships within the object model is shown in Figure 3-3.

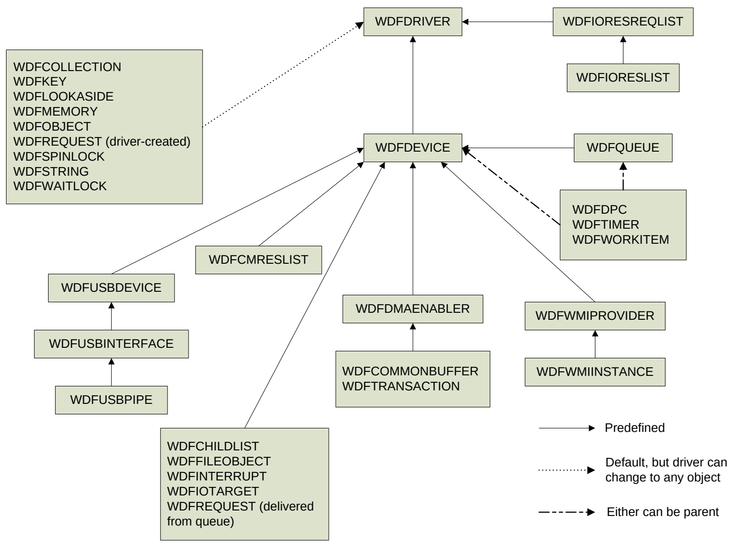


Figure 3-3 Object Relationships among the KMDF Objects (Microsoft Corporation (2), 2005)

The object model is used internally and by the framework I/O manager, but the driver is presented to the system in the same manner as any other driver. Every aspect of driver functioning is catered for within the model. Table 3-1 lists the objects and a few details regarding their use.

Table 3-1 KMDF Object Types (Microsoft Corporation (2), 2005)

Object	Type	Description
Child list	WDFCHILDLIST	Represents a list of the child devices for a device.
Collection	WDFCOLLECTION	Describes a list of similar objects, such as resources or the devices for which a filter driver filters requests.
Device	WDFDEVICE	Represents an instance of a device. A driver typically has one WDFDEVICE object for each device that it controls.
DMA common buffer	WDFCOMMONBUFFER	Represents a buffer that can be accessed by both the device and the driver to perform DMA.
DMA enabler	WDFDMAENABLER	Enables a driver to use DMA. A driver that handles device I/O operations has one WDFDMAENABLER object for each DMA channel within the device.
DMA transaction	WDFDMATRANSACTION	Represents a single DMA transaction.
Deferred procedure call (DPC)	WDFDPC	Represents a deferred procedure call.
Driver	WDFDRIVER	Represents the driver itself and maintains information about the driver, such as its entry points. Every driver has one WDFDRIVER object.
File	WDFFILEOBJECT	Represents a file object through which external drivers or applications can access the device.
Generic object	WDFOBJECT	Represents a generic object for use as the driver requires.
I/O queue	WDFQUEUE	Represents an I/O queue. A driver can have any number of WDFIOQUEUE objects.
I/O request	WDFREQUEST	Represents a request for device I/O.
I/O target	WDFIOTARGET	Represents a device stack to which the driver is forwarding an I/O request.
Interrupt	WDFINTERRUPT	Represents a device’s interrupt object. Any driver that handles device interrupts has one WDFINTERRUPT object for each IRQ or message-signalled interrupt (MSI) that the device can trigger.
Look-aside list	WDFLOOKASIDE	Represents a dynamically sized list of identical buffers that are allocated from the paged or nonpaged pool.

Object	Type	Description
Memory	WDFMEMORY	Represents memory that the driver uses, typically an input or output buffer that is associated with an I/O request.
Registry key	WDFKEY	Represents a registry key.
Resource list	WDFCMRESLIST	Represents the list of resources that have actually been assigned to the device.
Resource range list	WDFIORESLIST	Represents a possible configuration for a device.
Resource requirements list	WDFIORESREQLIST	Represents a set of I/O resource lists, which comprises all possible configurations for the device. Each element of the list is a WDFIORESLIST object.
String	WDFSTRING	Represents a counted Unicode string.
Synchronization: spin lock	WDFSPINLOCK	Represents a spin lock, which synchronizes access to data DISPATCH_LEVEL.
Synchronization: wait lock	WDFWAITLOCK	Represents a wait lock, which synchronizes access to data at PASSIVE_LEVEL.
Timer	WDFTIMER	Represents a timer that fires either once or periodically and causes a call-back routine to run.
USB device	WDFUSBDEVICE	Represents a USB device.
USB interface	WDFUSBINTERFACE	Represents an interface on a USB device.
USB pipe	WDFUSBPIPE	Represents a pipe in a USB interface.
Windows Management Instrumentation (WMI) instance	WDFWMIINSTANCE	Represents an individual WMI data block that is associated with a particular provider.
WMI provider	WDFWMIPROVIDER	Represents the schema for WMI data blocks that the driver provides.
Work item	WDFWORKITEM	Represents a work item, which runs in a system thread at PASSIVE_LEVEL.

The user processes interact with the driver via Win32 system calls. These calls are described in the research poster included in Appendix i. The Win32 calls forward the request on to the KMDF which routes the request to the appropriate driver, or manages the request itself. Once the request is forwarded to the driver by placing it in an appropriate queue, the driver can deal with the request and complete it, or forward the request to a driver lower down in the stack.

3.2.2 The Development Environment

To develop KMDF based drivers the DDK build environments are used. Build settings and environment variables are used to build a WDF driver. Any text editor can be used for the source coding but this is still limited to the ANSI C language.

3.3 Chapter Summary

This chapter presented the Windows Driver Model and the Windows Driver Foundations – 2 driver development models. The similarities and differences between these two models were briefly discussed.

The WDM is intended to provide a common development basis for Windows drivers for Windows 98 and beyond. It provides an object model and stipulates best practices when interacting with the operating system. The idea of a driver stack that separated the bus, device and hardware layers from each other was introduced. It was also noted that the developer was required to implement a significant proportion of OS interaction code, especially to cater for Plug ’n Play and Power modes and the transitions between them.

We then introduced the WDF and its associated frameworks: the User-Mode Driver Framework, and the Kernel-Mode Driver Framework. The reasons for Microsoft providing a new development model were discussed and we explained how the WDF was an evolution of the WDM.

4 Current Driver Specifications (WDM Version)

To identify the requirements for the WDF based mLAN Bus driver, we need to examine the WDM version within the setting of the entire mLAN PC System. This chapter examines the processes that rely upon the Bus driver to provide communication with the FireWire bus. We then analyse the two categories of drivers that are present.

4.1 mLAN™ System Design

The mLAN system is made up of a collection of processes that interact with applications and the mLAN drivers. These are shown in Figure 4-1 below.

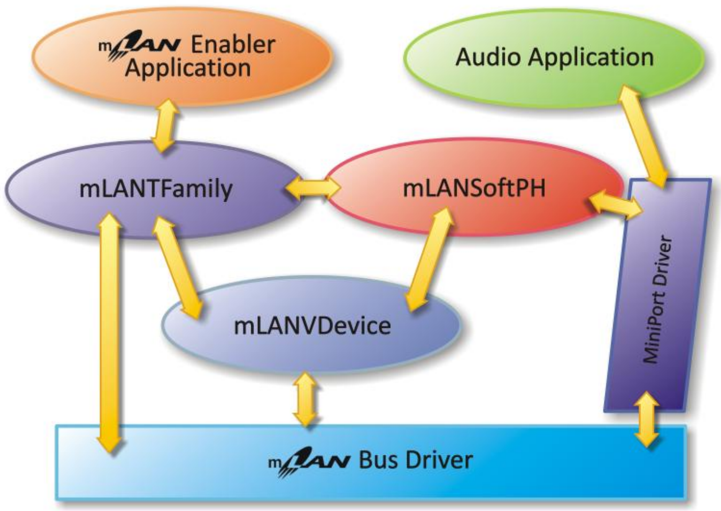


Figure 4-1 The mLAN System Processes

The mLANTFamily process is a resident process that maintains the Enabler object model and EnablerAPI for applications to interact with. The mLANSoftPH process is a software implementation of the PH1 mLAN IC and provides node functionality for the Windows PC. The mLANVDevice is a virtual mLAN device node on the IEEE1394 bus.

4.2 mLAN™ Drivers

The mLAN processes all rely on several underlying drivers to function. Figure 4-2 shows the positions of these drivers within the Windows platform. These drivers fall into two categories: Mini-port drivers and mLAN Bus driver.

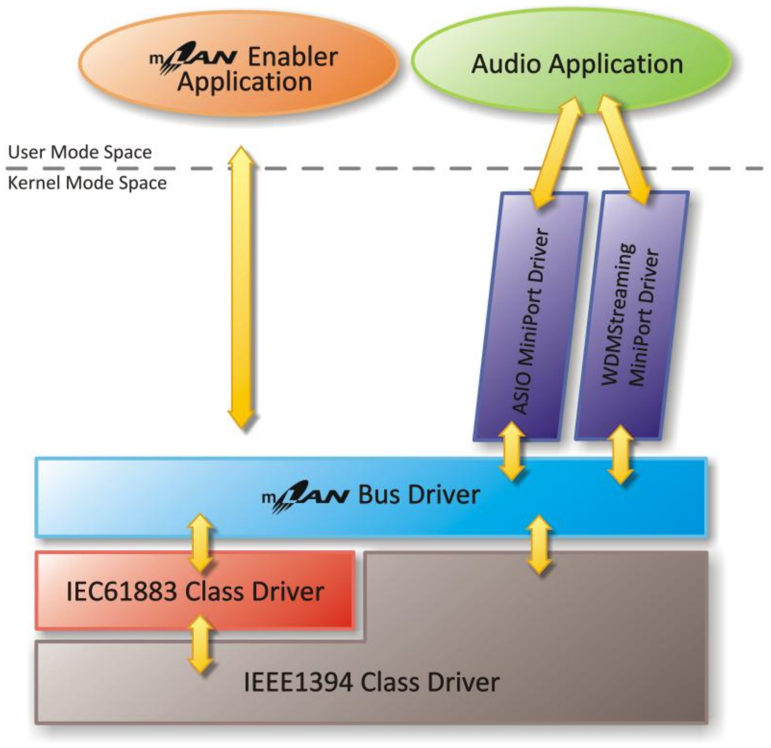


Figure 4-2 mLAN Bus Driver Position in the Windows Driver Stack

4.2.1 mLAN™ Mini-port Drivers

These drivers are used by the audio sub-system and user mode audio applications and are shown in purple in Figure 4-2 above. They expose audio and MIDI 'plugs' that can be used by any audio related application. These are standardised designs that the audio sub-system requires to interact with the plugs in a common way and provide functions like setting up a stream, starting a stream, and stopping a stream.

These drivers expose a specified format, but they achieve their functionality by making use of the mLAN Bus driver which allows them to interact with the bus.

4.2.2 mLAN™ Bus Driver

Shown in blue in Figure 4-2, this driver is responsible for communication with the 1394 and 61883 class drivers, and through these, the actual hardware bus. It provides user-mode processes with the ability to interact with the class drivers that lie in kernel-mode space. It does this by presenting IOCTL's related to the 1394 ad 61883 functions that processes can call, passing data structures in and receiving structures back. These IOCTL's are broadly discussed here and are listed in detail in Appendix ii. The processes communicate via Win32 calls which open a file handle attached to the driver, and then issue the IOCTL calls with pointers to memory structures.

This driver provides several categories of IOCTL's:

4.2.2.1 IEEE1394 Calls

To allow control of nodes, memory allocations and data transmission on the bus, the application needs access to the 1394 Bus driver. These calls mirror the 1394 class driver's functions, allowing the process to communicate with it via the mLAN Bus Driver.

4.2.2.2 IEC61883-1 Calls

As with the 1394 class driver, these IOCTL's provide an interface via which the process can communicate with the 61883 class driver.

4.2.2.3 Mini-port Calls

Mentioned previously, the mini-port related IOCTL's provide the miniport drivers with the functionality to implement audio and MIDI streaming over the 1394 bus.

4.2.2.4 mLAN™ Specific Calls

These calls are specifically for the mLAN system and provide information about the driver such as versioning.

4.3 Chapter Summary

In this chapter we briefly examined the mLAN PC System design. The fact that several processes, as well as the mini-port drivers rely upon the Bus driver for communication with the hardware bus was highlighted. We identified the IOCTL requirements for the mLAN Bus driver to provide similar functionality as the WDM version.

5 Re-Design of the mLAN™ Bus Driver (WDF Version)

After examining the WDM version, we can now consider how to implement the required functionality making use of the new WDF model. This entails designing a logical and physical layout, and aligning it with the requirements for the KMDF.

5.1 Logical Design

The WDM version of the mLAN Bus Driver provided an interface for user mode applications to access the 1394 Class Driver. It did this by exposing a range of IOCTL's similar to those provided by the class driver which allowed asynchronous packet sending and receiving, and isochronous stream control on the 1394 bus. In addition to the IOCTL's provided for the 1394 functionality, the IEC 61883-1 standard was catered for as well, allowing FCP control.(Miles, 2005)

To enable the PC to act as an mLAN device, it was found that the bus driver also provided IOCTL's that are used by miniport drivers to stream data onto the mLAN bus, where the bus driver is treated as a child device object to the miniport drivers. Functionality for WDM Audio and MIDI streaming, as well as ASIO streaming is provided.

This provides a single interface to the IEEE 1394 bus that is used by all parts of the mLAN System and allows central control over the function of the mLAN Bus.

It was decided to maintain this logical structure in a new driver and present the same IOCTL's to applications as the WDM version. This would allow the use of the new WDF version in the current solution without requiring changes to other layers within the mLAN System. The ability to make use of the driver separately from the current mLAN implementation to allow communication with the 1394 and 61883 class drivers is also catered for. This is achieved by providing two Win32 interfaces to the driver, one using the old mLAN Bus Driver GUID and one using a newly created GUID.

A new design proposal for the entire mLAN implementation is discussed in Section 6.4.

5.2 Physical Design and Build Environment

Even with the new technologies introduced to assist one, writing Windows Drivers is still a complicated task. There are several limitations and rules regarding how your driver interacts with the system, what it is permitted to do, and how you are allowed to implement functionality.

5.2.1 Coding Language

Because driver development interacts with the very bowels of the Windows system code, the availability of libraries and extensions to the coding language is not assured. This requires the use of standardised coding techniques and language specifications. Microsoft strongly recommend that

ANSI C is used throughout driver development to ensure the safety of the coding techniques(Microsoft Corporation (5), 2005). Thus no C++ object constructs, or use of the standard libraries is acceptable. Although this seems to limit ease of design, both the WDM specification as well as the new KMDF provide assistance to the developer via object creators, initialisation methods, and their own object models (Microsoft Corporation (2), 2005).

5.2.2 Build Environment

The build environment provided by Microsoft for driver compilation is contained within the Windows Driver Development Kit (DDK). This provides an array of build environments for different system builds. For example there is a checked and free build environment for Windows XP. The checked environment includes a large amount of debugging information in the compiled code and is used in the development cycle on a checked build of the operating system. This allows intuitive debugging information to be extracted from both the system and the driver. (Microsoft Corporation (6), 2005)

The DDK includes environments for Windows 2000, Windows XP, Windows Server 2003, and Windows Vista. If the operating system has a 64bit version, a 64bit checked and free environment is also provided. (Microsoft Corporation (6), 2005)

5.2.3 IDE Options

Because the build environment is an external implementation, any text editor can be used to code the source files for a driver. It was found that integrating the build process as a make-file project in the Visual Studio 2005 IDE provided an intuitive, highlighted coding interface and the ability to build directly using the configuration options within the DDK provided environment. It is also possible to use the debugging symbol files within VS2005 to allow code navigation and referencing.

5.2.4 Debugging

Because of the complicated and low-level nature of kernel-mode drivers, debugging is a significantly more complicated affair than the usual application level debugging scenario. Thankfully several tools are provided to allow detection of errors at an early stage in the development lifecycle. This includes both compile-time and run-time debugging tools.

5.2.4.1 Compile-time Analysis

The Windows DDK provides a tool called pre*fast* which analyses drivers at compilation for common design, logical and coding errors not normally detected by the compiler error detection (Microsoft Corporation (5), 2005). This tool was very helpful in identifying memory control problems.

5.2.4.2 Run-time Debugging

This is achieved with the use of the Debugging Tools for Windows kit. For kernel-mode debugging, a dual-PC system is required as the debugging host system uses kernel calls in the process of debugging and thus would also halt execution in the event of a breakpoint (Microsoft Corporation

(6), 2005). This requires that a target system is attached to the debugger via a communication link, most often a null-modem cable, to allow the host system to control the debugging process.

5.2.5 Code Layout

The driver code was broken up into related sections as described by Figure 5-1 below. This provided a logical and intuitive physical design to the source code allowing easy navigation to specific areas of interest.

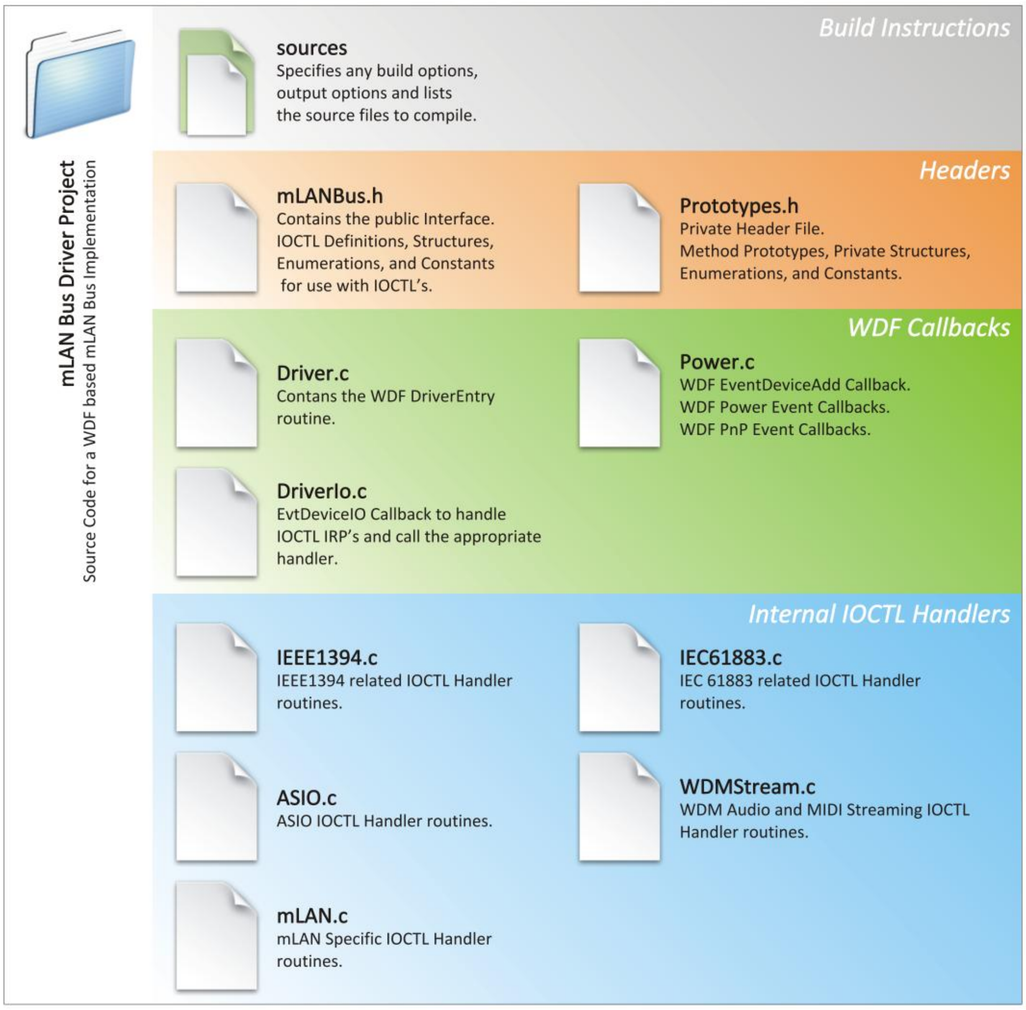

mLAN Bus Driver Project
Source Code for a WDF based mLAN Bus Implementation
Build Instructions
sources
Specifies any build options, output options and lists the source files to compile.
Headers
mLANBus.h
Contains the public Interface. IOCTL Definitions, Structures, Enumerations, and Constants for use with IOCTL's.
Prototypes.h
Private Header File. Method Prototypes, Private Structures, Enumerations, and Constants.
WDF Callbacks
Driver.c
Contans the WDF DriverEntry routine.
Power.c
WDF EventDeviceAdd Callback. WDF Power Event Callbacks. WDF PnP Event Callbacks.
DriverIo.c
EvtDeviceIO Callback to handle IOCTL IRP's and call the appropriate handler.
Internal IOCTL Handlers
IEEE1394.c
IEEE1394 related IOCTL Handler routines.
IEC61883.c
IEC 61883 related IOCTL Handler routines.
ASIO.c
ASIO IOCTL Handler routines.
WDMStream.c
WDM Audio and MIDI Streaming IOCTL Handler routines.
mLAN.c
mLAN Specific IOCTL Handler routines.

Figure 5-1 mLAN Physical Driver Design

Files that are used in the build process are included and provide instructions to the build environment at compilation. Two header files are used to separate the public and private definitions. This way, only the required IOCTL definitions, structures, and interface GUID's for interaction with the driver are included in the user application.

The WDF call-back routines (shown in Green) are used by the KMDF in response to actions and events within the driver and the system (Microsoft Corporation (2), 2005). For example, the

DriverIo.c file contains a call-back routine that is called when the framework queues an IRP in the default IO queue for the driver.

The private internal handlers (shown in Blue) are in turn called from within the IO call-back routine in response to the IOCTL codes that it receives within the IRP. This distribution of functionality allows grouping into KMDF Interaction, and Functionality areas.

5.2.5.1 KMDF Interaction (WDF Callbacks)

These allow the framework to interact with the driver. They will be discussed according to the file they are in. Note: All code is reduced in volume by removing debugging information and unnecessary clutter. Only the important actions are shown.

5.2.5.1.1 Driver.c

This contains the main entry point for the driver. A DriverEntry() function is called by the framework when the driver is being loaded (Microsoft Corporation (5), 2005). This creates a WDFDriver object to represent the driver class that is being added to the system (Microsoft Corporation (5), 2005). Driver.c also provides another required call-back – the EvtDeviceAdd call-back, which is invoked when a device that uses this driver is added to the system.

```
#pragma alloc_text(INIT, DriverEntry)

// Driver Entry Routine called when loading the driver.
// Specify Device Add event for PnP and create a Driver Object.
// This entry point is called directly by the I/O system.

NTSTATUS DriverEntry(
       IN PDRIVER_OBJECT     DriverObject,
       IN PUNICODE_STRING    RegistryPath
       )
{
       NTSTATUS             status = STATUS_SUCCESS;
       WDF_DRIVER_CONFIG    config;

       // Initialise the config structure and set
       // the DeviceAdd PnP Event Handler
       WDF_DRIVER_CONFIG_INIT(&config, mLANBus_EventDeviceAdd);

       // Create the Driver Object
       status = WdfDriverCreate(
              DriverObject,
              RegistryPath,
              WDF_NO_OBJECT_ATTRIBUTES,
              &config,
              WDF_NO_HANDLE
);

   return status;
}
```

5.2.5.1.2 Power.c

The Power and Plug and Play callbacks are located in this file. The only one required by a WDF driver is the Add Device handler that is called when the Driver object is created. In the new mLAN Bus driver this is called mLANBus_EventDeviceAdd() and this name is passed to the WdfDriverCreate() function in the DriverEntry() routine.

```
// PnP Device Add Event Handler

NTSTATUS mLANBus_EventDeviceAdd(
       IN WDFDRIVER          Driver,
       IN PWDFDEVICE_INIT    DeviceInit
       )
{
   NTSTATUS                      status = STATUS_SUCCESS;
   WDFDEVICE                     device;
    WDF_OBJECT_ATTRIBUTES         attributes;
   WDF_PNPPOWER_EVENT_CALLBACKS  pnpPowerCallbacks;
   WDF_IO_QUEUE_CONFIG           ioQConfig;



   // Initialise the Pnp/Power Callbacks structure.
   WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);

   // Set Callbacks for any of the functions we are interested in.
   // If no callback is set, Framework will take the default action

   // PnP Callbacks
   pnpPowerCallbacks.EvtDevicePrepareHardware =
       mLANBus_EventDevicePrepareHardware;

   // Power Callbacks
   pnpPowerCallbacks.EvtDeviceD0Entry = mLANBus_EventDeviceD0Entry;
   pnpPowerCallbacks.EvtDeviceD0Exit  = mLANBus_EventDeviceD0Exit;

   // Register the PnP and Power callbacks.
   WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);

   // Specify the size and type of device context.
   WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes, DEVICE_EXTENSION);

   // Create a Device object
   status = WdfDeviceCreate(&DeviceInit, &attributes, &device);

   // Tell the framework that this device will need an interface so that
   // User-Mode applications can interact with it.
   status = WdfDeviceCreateDeviceInterface(device,
       (LPGUID)&GUID_DEV_INTERFACE_MLAN, NULL);

   // Create a automanaged queue for receiving IOCTL requests.
   // All other requests are automatically failed by the framework.
   // By creating an automanaged queue we don't have to worry about
   // PNP/Power synchronization.
   WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQConfig,
       WdfIoQueueDispatchParallel);

   //Specify the Routine to be called to Handle Requests in the queue.
   ioQConfig.EvtIoDeviceControl = mLANBus_EventDeviceIoDeviceControl;

   status = WdfIoQueueCreate(
                     device,
                     &ioQConfig,
                     WDF_NO_OBJECT_ATTRIBUTES,
                     &deviceContext->IoctlQueue
                     );

   return status;

}
```

Any other Power or PnP events can have call-backs stipulated here (Microsoft Corporation (5), 2005). The important part is the creation of a managed IO queue that will receive any IOCTL requests directed at the driver. Any requests will be queued here and the mLANBus_EventDeviceIoDeviceControl() function will be called to handle the request.

5.2.5.1.3 DriverIO.c

This file contains the mLANBus_EventDeviceIoDeviceControl() function that will handle the requests forwarded to this driver. This function simply has a select case statement that examines the IOCTL code of the request, checks that the sizes of the input and output buffers are appropriate for the IOCTL, and forwards the information on to an internal handler.

```
// Receives all IO Requests and calls the appropriate handler.
// Called by the framework for every IO Request object in the managed queue
created in EvtDeviceAdd().

VOID mLANBus_EventDeviceIoDeviceControl(
            IN WDFQUEUE      Queue,
            IN WDFREQUEST    Request,
            IN size_t        OutputBufferLength,
            IN size_t        InputBufferLength,
            IN ULONG         IoControlCode
            )
{
   NTSTATUS ntStatus = STATUS_SUCCESS;
   PDEVICE_EXTENSION   deviceExtension;
   PVOID               ioBuffer = NULL;
   WDFDEVICE           device;
   size_t              bufLength;

   device = WdfIoQueueGetDevice(Queue);
   deviceExtension = GetDeviceContext(device);

   // Since all the IOCTLs handled here are buffered,
   // WdfRequestRetrieveOutputBuffer &
   // WdfRequestRetrieveInputBuffer return the same buffer pointer.
   // So make sure you read all the information you need from
   // the buffer before you write to it. Also requiredLength of the
   // buffer vary from ioctl to ioctl, so we will pretend that we need
   // zero length buffer and do the length check later in the specific
   // ioctl case.
   ntStatus = WdfRequestRetrieveInputBuffer(
                      Request,
                      0,
                      &ioBuffer,
                      &bufLength
                      );
   // Switch case for each IOCTL presented by the driver.
   switch (IoControlCode)
   {

      case IOCTL_MLAN_ASYNC_READ:
      {
         PASYNC_READ pAsyncRead;

         // Check for valid input buffer size
         if (InputBufferLength < sizeof(ASYNC_READ))
         {
            ntStatus = STATUS_BUFFER_TOO_SMALL;
         }
         else
         {
            pAsyncRead = (PASYNC_READ)ioBuffer;

            // Check for valid output buffer size
            if ((OutputBufferLength < sizeof(ASYNC_READ)) ||
                (OutputBufferLength-sizeof(ASYNC_READ) <
                      pAsyncRead->nNumberOfBytesToRead))
            {
               ntStatus = STATUS_BUFFER_TOO_SMALL;
            }
            else
            {
               ntStatus = IEEE1394_AsyncRead_Handler(
                  device,
                  Request,
                  *pAsyncRead,
                  pAsyncRead
                  );

               // Return data
               if (NT_SUCCESS(ntStatus))
               {
                  WdfRequestSetInformation(Request,  OutputBufferLength);
               }
            }
         }

      }
      break;

       // Case for each IOCTL
       …
   }

   // only complete if the device is there
   if (ntStatus != STATUS_PENDING)
   {
      WdfRequestComplete(Request, ntStatus);
   }

}
```

5.2.5.2 Internal Handlers

These implement the functionality for each IOCTL and are within the files in the Blue section in Figure 5-1. They are grouped according to IEEE1394, IEC61883, WDMMIDI, WDMAudio, and ASIO Streaming functionality. The data received from the IO handler is examined, altered if needed, and forwarded on to the appropriate class driver in the required format.

The IOCTL_MLAN_ASYNC_READ handler is given below as an example:

```
NTSTATUS IEEE1394_AsyncRead_Handler(
       IN WDFDEVICE            Device,
       IN WDFREQUEST       Request,
       IN ASYNC_READ       AsyncRead,
       IN PASYNC_READ      pAsyncRead
       )
{
       NTSTATUS            ntStatus            = STATUS_SUCCESS;
       PDEVICE_EXTENSION   deviceExtension     = GetDeviceContext(Device);
       PIRB                pIrb                = NULL;
       PMDL                pMdl                = NULL;

       RtlZeroMemory (pIrb, sizeof (IRB));
       pIrb->FunctionNumber = REQUEST_ASYNC_READ;
       pIrb->Flags = 0;
       pIrb->u.AsyncRead.DestinationAddress =
                      AsyncRead.DestinationAddress;
       pIrb->u.AsyncRead.nNumberOfBytesToRead =
                      AsyncRead.nNumberOfBytesToRead;
       pIrb->u.AsyncRead.nBlockSize = AsyncRead.nBlockSize;
       pIrb->u.AsyncRead.fulFlags = AsyncRead.fulFlags;
       pIrb->u.AsyncRead.ulGeneration = AsyncRead.ulGeneration;

       // creat a MDL buffer to hold info read from device
       pMdl = IoAllocateMdl(
              AsyncRead.Data,
              AsyncRead.nNumberOfBytesToRead,
              FALSE, FALSE, NULL);
       pIrb->u.AsyncRead.Mdl = pMdl;

   // submit the irb to class driver
   ntStatus = IEEE1394_SubmitIrpSynch(deviceExtension, pIrb);

   // rerieve returned data
   pAsyncRead = (PASYNC_READ)&(pIrb->u.AsyncRead);

   return(ntStatus);

}
```

To forward a request to the underlying class driver, a new I/O Request Block, referred to by pIRB in the above code, needs to be created. pIRB is assigned the appropriate function number defined in the class driver – in this case REQUEST_ASYNC_READ. Now the appropriate fields within the IRB are populated and the newly created request is forwarded by the IEEE1394_SubmitIrpSynch() function to the lower class driver. Once the call completes, the pointer information is updated and the request marked complete. The application that called the IOCTL can now access the data via the pointer returned.

The layering within the code of the WDF Bus driver allows reuse of common code, and logical separation into related sections of code.

5.3 Chapter Summary

In this chapter we discussed the logical and physical layout of the WDF mLAN Bus Driver. We began by describing the high-level logical design required to implement the required functionality. We then moved on to choices in coding language, build environments, and IDE where we described facilities provided by the DDK. Debugging and analysis were briefly mentioned and the special requirements for kernel-mode debugging were highlighted with reference to the Debugging Tools for Windows.

The physical layout of code within the project was examined. The separation into areas of Interaction and Functionality was noted as well as the close alignment of the physical and logical design. We then discussed each area of functionality in some detail with specific reference to the framework methods and the interaction with framework objects.

6 Results and Possible Future Directions

After the completion of the WDF Bus driver, we are able to examine the success or failure of our solution. We also discuss prospects for future work in this area or research.

6.1 Improvements On WDM Version

6.1.1 Design

The internal design of the WDF based driver has a clean logical layout of code with easily traceable paths of execution. The separation into areas of interaction with the KMDF, and implementation of functionality allows easy troubleshooting and debugging. The absence of code catering for all PnP and Power related functionality results in a simplified design compared to the WDM implementation, with fewer areas of possible problems.

The call-back based design of the KMDF provides an effective manner for the framework to manage the driver communication and interaction with the OS. The provision of default actions for those events not provided by the driver allows the focus of design to be on the functionality of the driver rather than OS interaction.

6.1.2 Implementation

The fact that only the functionality of the driver is coded results in a significant reduction in the amount of code. The original WDM implementation has 7460 lines of code, where the WDF implementation has only 4711 (Calculated without functionality not yet implemented by the WDF version i.e. ASIO, and WDMAudio related IOCTL's).

The KMDF object model is intuitive and easy to use with sufficient inherent functionality to not require any external libraries. The ability to refer to objects allows self-describing and streamlined code.

The new version showed the ability to interact with the class drivers, and hence the bus, in the same manner and providing the same functionality as the WDM version. This indicates that the ability to implement the entire system in WDF is present.

6.2 Non-Functional Aspects

The mini-port related IOCTL's were not sufficiently completed to be included in this report. Thus the driver only provides IEEE1394 and IEC61883 functionality. This is sufficient for its use in an Enabler scenario which does not require any further functionality – perfect for the mLAN Bus Driver suggested in the re-designed mLAN System later.

It was decided that due to the complexity of the original mLAN Bus driver, the use of a proprietary C++ based framework for implementation which obscured the WDM code, and my inexperience in the WDM and WDF, the goal of completing the implementation of the entire array of functionality within the mLAN Bus driver was not achievable in the time allowed for this research.

Also, the mini-port driver that would allow the option of WaveRT streaming under Windows Vista was not implemented. This required the completion of the other mini-port related functionality in the Bus driver first.

6.3 Possible Extensions

6.3.1 Completion of the Mini-port Related IOCTL's

This would entail examining the functionality provided for the ASIO and WDMStreaming mini-port drivers by the WDM implementation and reproducing this in the WDF implementation. Completion of these IOCTL's would allow the use of the WDF implementation in the current mLAN System as both the enabler interface and an interface to the bus for the mini-port drivers.

6.3.2 Porting the Mini-port Drivers to WDF

To complete the conversion of the current mLAN System from WDM based drivers to WDF, the mini-port drivers used to present connections to audio applications need to be analysed and re-implemented using the WDF model. These would make use of the mini-port related IOCTL's provided by the Bus driver.

6.3.3 Implementing Vista™ Specific Audio Streaming

One of the original goals of this project was to examine the use of the mLAN system on Windows Vista. The current implementation will function on Windows Vista, as it does on Windows XP, without modification. The conversion to the WDF model of driver design is recommended for use in Vista and later versions of Microsoft Windows as this will be become the standard model for driver development.

The WaveCyclic port driver used in the WDM version, requires continual polling by a thread which increases the latency when using it. The WaveRT port driver does not require this and also makes use of a Real-Time scheduling policy available in Windows Vista.(Microsoft Corporation (1), 2006)

Windows Vista does provide the opportunity to use a new priority mode for kernel audio streaming and this could provide an alternative to the current ASIO streaming implementation. Implementing this would entail the creation of a new WaveRT mini-port driver or the modification of the current WaveCyclic one to support the new WaveRT model. It would also entail the inclusion of related IOCTL's in the mLAN Bus driver to provide communication with the bus.

6.4 Suggestions For Further Design Changes

The current system entails the use of many individual user processes with many interactions between each. It would make sense, especially in the light of new developments in the mLAN specification – namely the Open Generic Transporter – to re-examine the current design and consider any simplifications that could be achieved.

Two major roles that the PC can play in an mLAN environment have been identified. The first is that of an Enabler. This role is responsible for controlling the mLAN network and the connections within it. Within the Windows environment, a System Service process would be ideal for this role. The service could be automatically started upon booting the system, could maintain an object model of current connections, and provide an interface to allow any application to interact with it. The service would require access to the IEEE1394 bus, which could be provided by an mLAN Bus Driver presenting IEEE1394 IOCTL's to the service. Here, only IOCTLS related to the IEEE 1394 bus and IEC 61883-1 FCP functionality are required.

The second is that of an mLAN node. This is required if the PC is to be used to stream audio/MIDI to or from another mLAN device. To achieve this, a virtual mLAN device would need to be implemented and expose audio/MIDI plugs to audio applications on the PC. This could be achieved without making use of a process, but by incorporating the functionality into a single mLAN Node driver that could interact with the mLAN Bus Driver. It would also require the use of mini-port drivers for interaction with the audio sub-system of the OS.

The separation of these two roles allows the user more flexibility in design choices. It also keeps the implementation of each role separate, resulting in a cleaner design and implementation of each. The current system is very interwoven and a complicated model to use, let alone analyse or modify. This model also allows the ability to constantly have the facilities available. In the current system, the user has to enable the mLAN system when they require it, and then has a significant delay whilst the various processes and drivers are loaded.

6.5 Chapter Summary

The results of implementing the mLAN Bus driver using the KMDF were discussed in this chapter. It was shown that the use of the WDF model proved successful and the ability to implement all the mLAN drivers using the KMDF was available.

We also listed the aspects of the original design that were not implemented and discussed the possible reasons for this. We then discussed possible areas of further work including completion of the mini-port related IOCTL's in the Bus driver, porting the current WDM mini-port drivers to WDF versions, and the inclusion of a new mini-port for WaveRT streaming in Windows Vista.

7 Conclusion

The current Yamaha mLAN System, although effective in its role, is a complicated design and often difficult to use. This can easily be seen if one consults any online community web site of Yamaha's mLAN technology. Here we find that the criticism for the product is most often due to the PC tools and driver implementation. By demonstrating the ability to re-design the mLAN Bus Driver using the new Windows Driver Foundation (WDF) development model, we are able to not only simplify the current implementation, but also propose a new design for the entire system that may provide a simpler, more reliable solution.

We began by examining the Windows Driver Model (WDM) implementation of the mLAN Bus driver to identify the requirements for a new implementation. This was complicated by the use of a proprietary driver development framework to implement the WDM version which obscured the WDM code.

After analysing the role of the driver in the mLAN System, we designed a new implementation making use of the Kernel-Mode Driver Framework (KMDF). Here we placed special emphasis on maintaining a close parallel relationship between the logical and physical layout. The driver was divided into two main areas: KMDF Interaction and Functionality.

KMDF Interaction contained all the call-back functions required to interact with the framework. This consisted of two functions for the creation of Driver and Device objects, as well as the call-back for a managed I/O queue. The I/O queue call-back was then responsible for calling the appropriate Internal Handler routine within the Functionality section for each IOCTL presented by the driver.

The Functionality section was concerned with forwarding on any requests to the underlying class drivers for both the 1394 and 61883 related IOCTL's. This entailed constructing a new request and passing it down the driver stack. Once completed, pointers were returned to the output buffer of the managed queue where the calling application could access the information.

Use of the WDF development model and its associated framework proved to be very successful. The provision in the framework for managed queues, default functionality and simplified interaction enables the developer to concentrate on implementing the functionality required by the device driver instead of the interaction with the OS as with the previous WDM model. The framework will definitely result in more reliable, smaller, and less complicated device drivers for the Windows operating system.

Implementation of the driver was successful and made use of the Microsoft Driver Development Kit (DDK) build environments. The Visual Studio 2005 IDE was used for editing and navigating code, whilst the WinDBG debugger was used for kernel-mode debugging on a separate test bed. The result was clean, logical code that is easy to maintain and understand. The driver is installed with the use of a .inf file which specifies the class of device, and was shown to communicate with the hardware bus as expected. The driver has enough functionality to be able to support an Enabler application in the connection management role within the mLAN System.

With the completion of the 1394 and 61883 areas of functionality, the mini-port related areas are still to be addressed. The completion of these IOCTL's would allow the use of the WDF Bus driver for streaming audio onto the 1394 bus.

Inclusion of the WaveRT streaming option for users of Windows Vista will provide another option for audio transport allowing even more flexibility from the mLAN System. More attention needs to be paid to how the operating system implements audio transport and how best to make use of the facilities provided though.

A re-design of the system would allow Yamaha to address the concerns of its users and provide a reliable, streamlined PC solution that would add value to its product line. This is definitely achievable by making use of new technologies in driver design and catering for the differences in implementation between different platforms.

References

1394 Trade Association. (2006). *1394 Technology - About.* Retrieved October 12, 2006, from 1394 Trade Association: http://www.1394ta.org/Technology/About/faq.htm

Cant, C. (1999). *Writing Windows® WDM Device Drivers.* Lawrence, Kansas: Miller Freeman.

Foss, R. (2006). *Honours Audio Engineering Course Notes.* Grahamstown: Rhodes University.

IEEE. (2002, April 02). *IEEE Approves Amendment to IEEE 1394™ Standard.* Retrieved October 12, 2006, from IEEE Announcements: http://standards.ieee.org/announcements/1394bapp.html

Microsoft Corporation (1). (2006, January 10). A Wave Port Driver for Real-Time Audio Streaming - Windows Vista™ Version. Redmond, Washington, USA.

Microsoft Corporation (2). (2005, December 2). Architecture of the Kernel-Mode Driver Framework. Redmond, Washington, USA.

Microsoft Corporation (3). (2005, October 25). Architecture of the Windows® Driver Foundation. Redmond, Washington, USA.

Microsoft Corporation (4). (2006, September 20). Introdution to Kernel-Mode Driver Development for Application Developers. Redmond, Washington, USA.

Microsoft Corporation (5). (2005, October 29). KMDF Documentation. Redmond, Washington, USA.

Microsoft Corporation (6). (2005, May 27). Windows® DDK Documentation. Redmond, Washington, USA.

Miles, S. (2005). *Analysis and Modelling of the Windows mLAN Driver.* Rhodes University. Rhodes University.

Oney, W. (1999). *Programming the Microsoft® Windows® Driver Model.* Redmon, Washington: Microsoft Press.

Vienna Institute of Technology. (2001). *Consumer Audio Video Equipment - Digital Interface (IEC 61883).* Retrieved October 12, 2006, from Vienna University of Technology ICT IEEE1394 Multimedia Group: http://www.ict.tuwien.ac.at/ieee1394/iec61883/iec61883-en.html

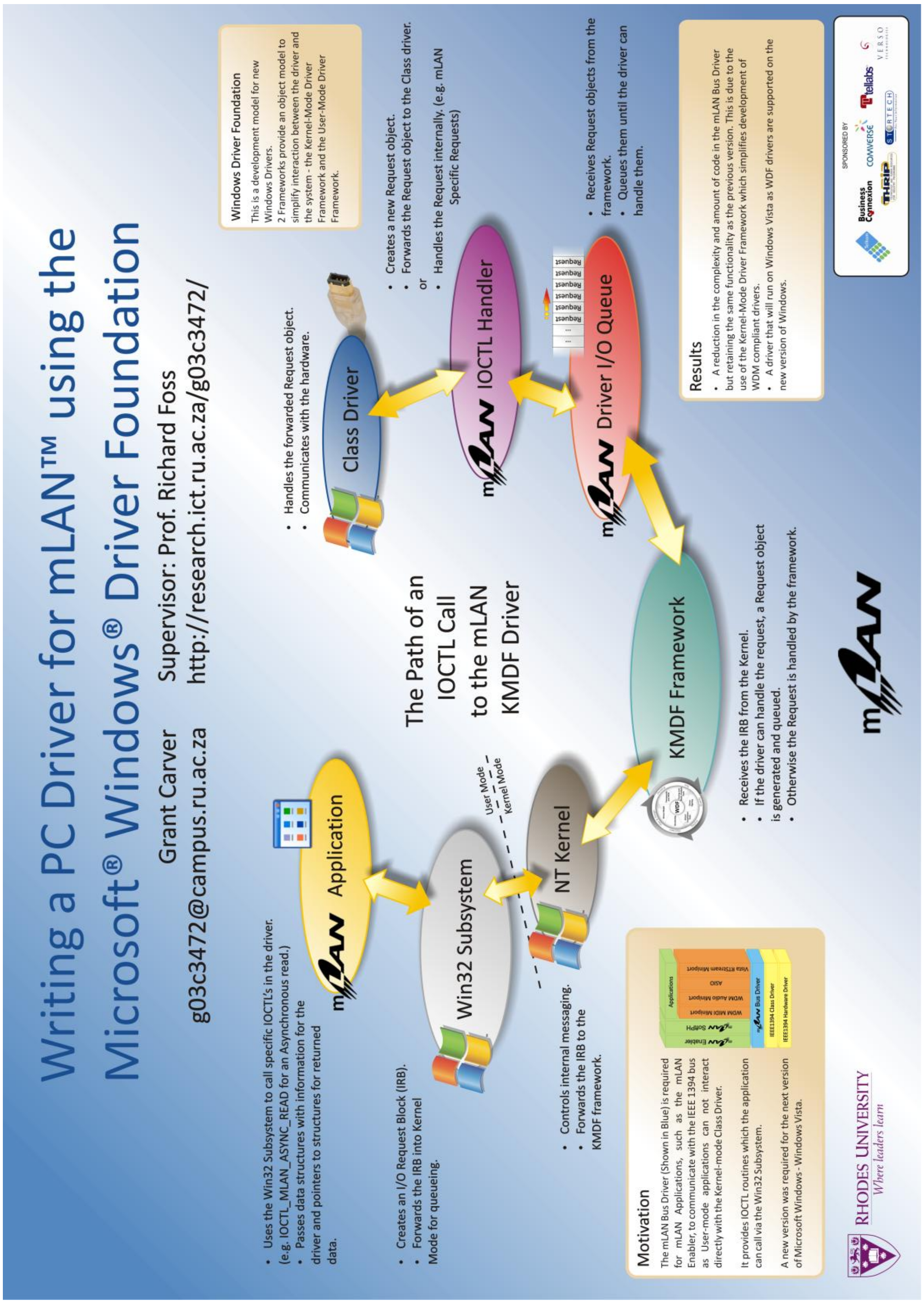
Table of Figures

Table of Tables

Index

Appendices

i. Research Poster – Path of an IOCTL



Writing a PC Driver for mLAN™ using the Microsoft® Windows® Driver Foundation

Grant Carver
g03c3472@campus.ru.ac.za

Supervisor: Prof. Richard Foss
http://research.ict.ru.ac.za/g03c3472/

The Path of an IOCTL Call to the mLAN KMDF Driver

- Uses the Win32 Subsystem to call specific IOCTL's in the driver. (e.g. IOCTL_MLAN_ASYNC_READ for an Asynchronous read.)
- Passes data structures with information for the driver and pointers to structures for returned data.

- Creates an I/O Request Block (IRB).
- Forwards the IRB into Kernel Mode for queueing.

- Controls internal messaging.
- Forwards the IRB to the KMDF framework.

- Receives the IRB from the Kernel.
- If the driver can handle the request, a Request object is generated and queued.
- Otherwise the Request is handled by the framework.

- Receives Request objects from the framework.
- Queues them until the driver can handle them.

- Creates a new Request object.
- Forwards the Request object to the Class driver.

or

- Handles the Request internally. (e.g. mLAN Specific Requests)

- Handles the forwarded Request object.
- Communicates with the hardware.

Windows Driver Foundation

This is a development model for new Windows Drivers.
2 Frameworks provide an object model to simplify interaction between the driver and the system - the Kernel-Mode Driver Framework and the User-Mode Driver Framework.

Motivation

The mLAN Bus Driver (Shown in Blue) is required for mLAN Applications, such as the mLAN Enabler, to communicate with the IEEE 1394 bus as User-mode applications can not interact directly with the Kernel-mode Class Driver.

It provides IOCTL routines which the application can call via the Win32 Subsystem.

A new version was required for the next version of Microsoft Windows - Windows Vista.

Results

- A reduction in the complexity and amount of code in the mLAN Bus Driver but retaining the same functionality as the previous version. This is due to the use of the Kernel-Mode Driver Framework which simplifies development of WDM compliant drivers.
- A driver that will run on Windows Vista as WDF drivers are supported on the new version of Windows.

ii. mLAN WDF Bus Driver IOCTL Calls

IEEE1394
IOCTL_MLAN_ALLOCATE_ADDRESS_RANGE
Input: A pointer to an ALLOCATE_ADDRESS_RANGE struct.
Output: A pointer to an altered ALLOCATE_ADDRESS_RANGE struct.

IOCTL_MLAN_ADDRESS_RANGE_NOTIFY
Accepts: A pointer to an ADDRESS_RANGE_NOTIFY struct.
Returns: A pointer to an altered ADDRESS_RANGE_NOTIFY strut.

IOCTL_MLAN_SET_ADDRESS
Accepts: A pointer to a SET_ADDRESS_DATA struct.

IOCTL_MLAN_GET_ADDRESS
Accepts: A pointer to a GET_ADDRESS_DATA struct.
Returns: A pointer t an altered GET_ADDRESS_DATA struct.

IOCTL_MLAN_FREE_ADDRESS_RANGE
Accepts: A HANDLE to the address range.

IOCTL_MLAN_ASYNC_LOCK
Accepts: A pointer to an ASYNC_LOCK struct.
Returns: A pointer to an altered ASYNC_LOCK struct.

IOCTL_MLAN_ASYNC_READ
Accepts: A pointer to an ASYNC_READ struct.
Returns: A pointer to an altered ASYNC_READ struct.

IOCTL_MLAN_ASYNC_WRITE
Accepts: A pointer to an ASYNC_WRITE struct.

IOCTL_MLAN_BUS_RESET_NOTIFY
Accepts: A pointer to an MLAN_BUS_RESET_NOTIFY struct.
Returns: A pointer to an altered MLAN_BUS_RESET_NOTIFY struct.

IOCTL_MLAN_BUS_RESET
Accepts: A ULONG value.

IOCTL_MLAN_GET_LOCAL_HOST_INFO
Accepts: A pointer to a GET_LOCAL_HOST_INFORMATION struct.
Returns: A pointer to an altered GET_LOCAL_HOST_INFORMATION struct.

IOCTL_MLAN_SEND_PHY_CONFIG_PACKET
Accepts: A pointer to a PHY_CONFIGURATION_PACKET struct.

IOCTL_MLAN_GET_LOCAL_NODE_ADDRESS
Accepts: A pointer to a GET_LOCAL_NODE_ADDRESS struct.
Returns: A pointer to an altered GET_LOCAL_NODE_ADDRESS struct.

IOCTL_MLAN_GET_CHANNELS_AVAILABLE
Returns: A pointer to a LARGE_INTEGER struct.

IOCTL_MLAN_ALLOCATE_CHANNEL
Accepts: A ULONG value.

IOCTL_MLAN_RELEASE_CHANNEL
Accepts: A ULONG value.

IOCTL_MLAN_GET_BANDWIDTH_AVAILABLE
Returns: A ULONG value.

IOCTL_MLAN_ALLOCATE_BANDWIDTH
Accepts: A ULONG value.

IOCTL_MLAN_RELEASE_BANDWIDTH
Accepts: A HANDLE to the bandwidth.

IOCTL_MLAN_ALLOCATE_STREAM
Accepts: A pointer to an MLAN_ISOCH_PARAM struct.
Returns: A pointer to an altered MLAN_ISOCH_PARAM struct.

IOCTL_MLAN_START_STREAM
Accepts: A pointer to an MLAN_STREAM_COMMAND struct.
Returns: A pointer to an altered MLAN_STREAM_COMMAND struct.

IOCTL_MLAN_CONNECT_SEQUENCES_TO_DEVICES
Accepts: A pointer to an MLAN_ISOCH_PARAM struct.
Returns: A pointer to an altered MLAN_ISOCH_PARAM struct.

IOCTL_MLAN_SET_SYT_SOURCE
Accepts: A pointer to an MLAN_ISOCH_PARAM struct.
Returns: A pointer to an altered MLAN_ISOCH_PARAM struct.

IOCTL_MLAN_STOP_STREAM
Accepts: A pointer to an MLAN_STREAM_COMMAND struct.
Returns: A pointer to an altered MLAN_STREAM_COMMAND struct.

IOCTL_MLAN_GET_STREAM_INFO
Accepts: A pointer to an MLAN_ISOCH_PARAM struct.
Returns: A pointer to an altered MLAN_ISOCH_PARAM struct.

IOCTL_MLAN_FREE_STREAM
Accepts: A pointer to an MLAN_STREAM_COMMAND struct.
Returns: A pointer to an altered MLAN_STREAM_COMMAND struct.

IEC61883
IOCTL_MLAN_61883_CONNECT_PLUG
IOCTL_MLAN_61883_CREATE_PLUG
IOCTL_MLAN_61883_PLUG_NOTIFY
IOCTL_MLAN_61883_DELETE_PLUG
IOCTL_MLAN_61883_DISCONNECT_PLUG
IOCTL_MLAN_61883_GET_FCP_REQUEST
IOCTL_MLAN_61883_GET_FCP_RESPONSE
IOCTL_MLAN_61883_GET_PLUG_HANDLE
IOCTL_MLAN_61883_GET_PLUG_STATE
IOCTL_MLAN_61883_SEND_FCP_REQUEST
IOCTL_MLAN_61883_SEND_FCP_RESPONSE
IOCTL_MLAN_61883_SET_FCP_NOTIFY
IOCTL_MLAN_61883_SET_PLUG

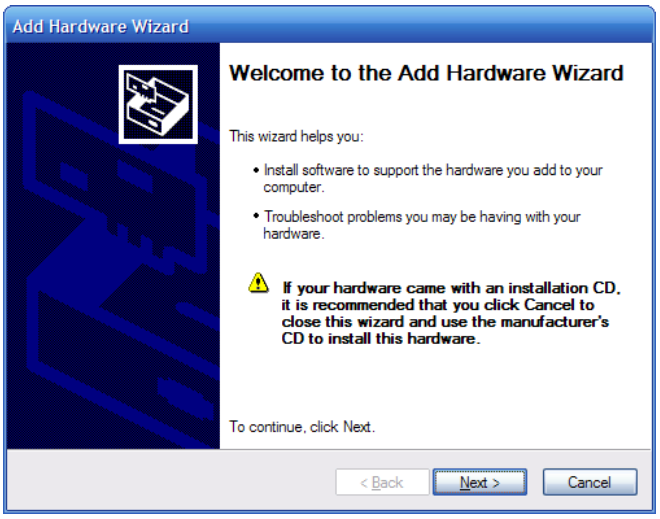
ASIO
Not implemented yet.

WDMSTREAMING
Not implemented yet.

MLAN SPECIFIC
IOCTL_MLAN_GET_DRIVER_VERSION

iii. mLAN WDF Bus Driver Installation

The mLAN WDF Bus Driver can be installed on Windows XP, Windows 2000, Windows 2003, and Windows Vista. To begin installation, go to the *Control Panel* and start the *Add Hardware* wizard.



Welcome to the Add Hardware Wizard

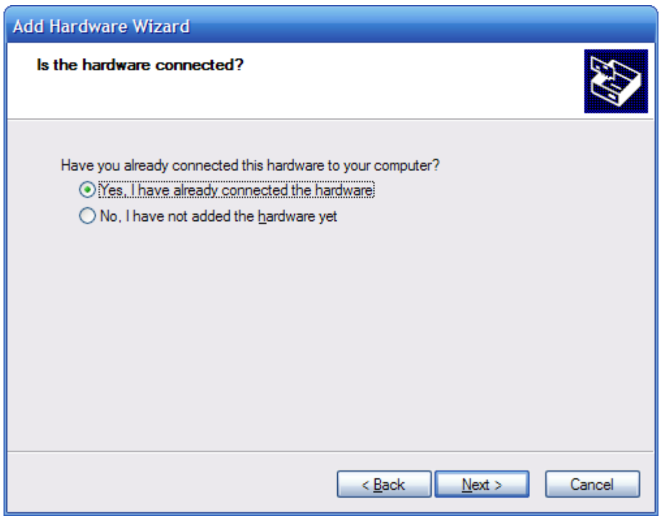
This wizard helps you:

- Install software to support the hardware you add to your computer.
- Troubleshoot problems you may be having with your hardware.

If your hardware came with an installation CD, it is recommended that you click Cancel to close this wizard and use the manufacturer's CD to install this hardware.

To continue, click Next.

Click *Next* and select the *Yes, I have already connected the hardware* option.

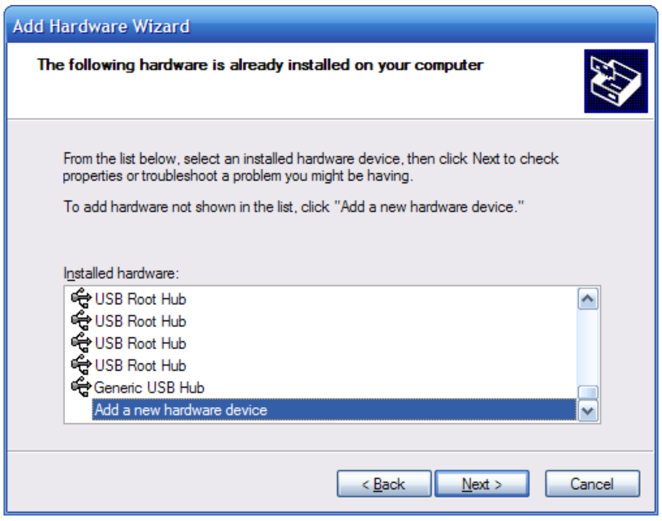


Is the hardware connected?

Have you already connected this hardware to your computer?

[✓ Yes, I have already connected the hardware]
No, I have not added the hardware yet

Click *Next* and then select *Add a new hardware device* from the very bottom of the list given to you.


Add Hardware Wizard

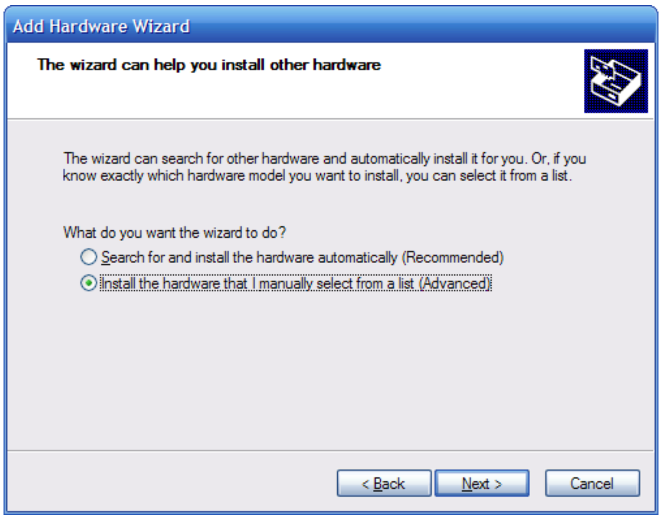
The following hardware is already installed on your computer

From the list below, select an installed hardware device, then click Next to check properties or troubleshoot a problem you might be having.

To add hardware not shown in the list, click "Add a new hardware device."

Installed hardware:
- USB Root Hub
- USB Root Hub
- USB Root Hub
- USB Root Hub
- Generic USB Hub
- Add a new hardware device

Click *Next* and then select the *Install the hardware that I manually select from a list* option.


Add Hardware Wizard

The wizard can help you install other hardware

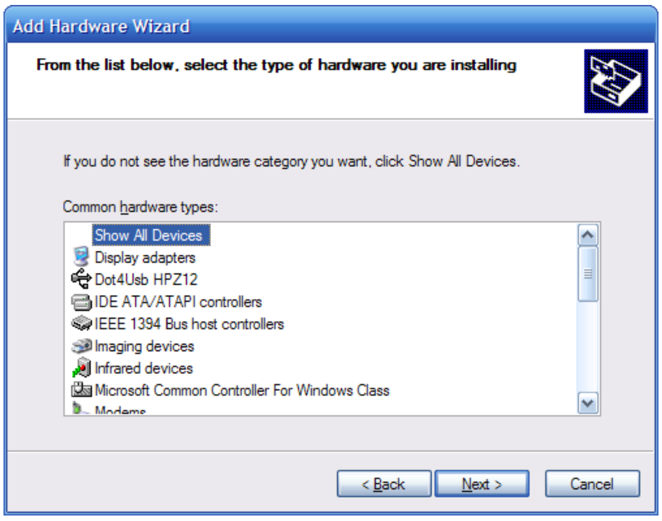
The wizard can search for other hardware and automatically install it for you. Or, if you know exactly which hardware model you want to install, you can select it from a list.

What do you want the wizard to do?

○ Search for and install the hardware automatically (Recommended)

[✓ Install the hardware that I manually select from a list (Advanced)]

Click *Next* and then select *Show All Devices*.

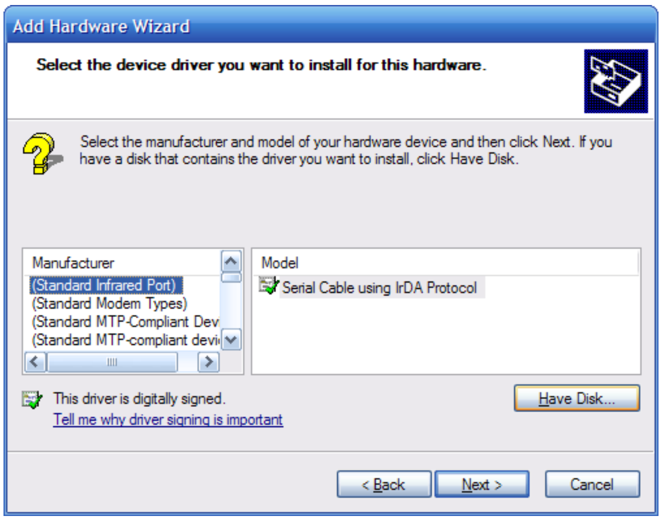

Add Hardware Wizard

From the list below, select the type of hardware you are installing

If you do not see the hardware category you want, click Show All Devices.

Common hardware types:
- Show All Devices
- Display adapters
- Dot4Usb HPZ12
- IDE ATA/ATAPI controllers
- IEEE 1394 Bus host controllers
- Imaging devices
- Infrared devices
- Microsoft Common Controller For Windows Class
- Modems

After a short pause, a list of hardware devices will be presented to you. Click on the *Have Disk…* button to direct the wizard to the place where you stored the mLAN Driver installation files.



Add Hardware Wizard

Select the device driver you want to install for this hardware.

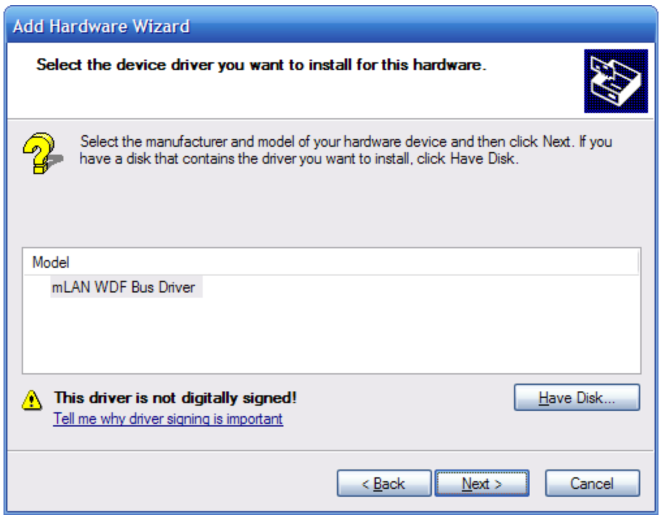
Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

Manufacturer
(Standard Infrared Port)
(Standard Modem Types)
(Standard MTP-Compliant Dev
(Standard MTP-compliant devi

Model
Serial Cable using IrDA Protocol

This driver is digitally signed.
Tell me why driver signing is important

You will now get a list with *mLAN WDF Bus Driver* as an option. Select this and click *Next*.



Add Hardware Wizard

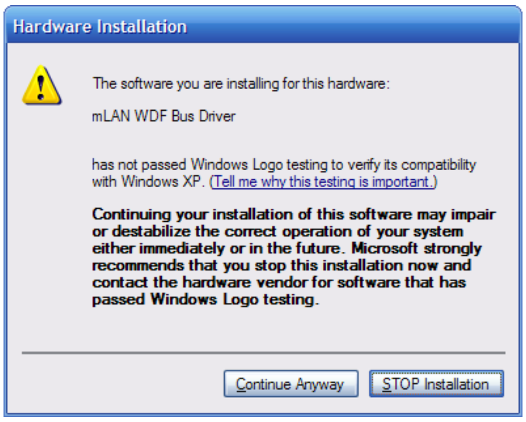
Select the device driver you want to install for this hardware.

Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

Model
mLAN WDF Bus Driver

This driver is not digitally signed!
Tell me why driver signing is important

You will be asked to confirm the installation of an un-signed driver.



Hardware Installation

The software you are installing for this hardware:

mLAN WDF Bus Driver

has not passed Windows Logo testing to verify its compatibility with Windows XP. (Tell me why this testing is important.)

Continuing your installation of this software may impair or destabilize the correct operation of your system either immediately or in the future. Microsoft strongly recommends that you stop this installation now and contact the hardware vendor for software that has passed Windows Logo testing.

The driver files will be copied and the device installed. Once this is complete a final screen will be displayed informing of the installation success. The driver will automatically load when Windows boots. To uninstall, simply Uninstall from the Device Manager.

iv. mLAN WDF Bus Driver Usage

To interact with the mLAN WDF Bus Driver via it's presented IOCTL's you use several Win32 functions to enumerate the specific device and then open a file handle attached to it.

These instructions can be found in the Win32 Help Documentation provided by Microsoft under the heading DeviceIO.

The procedure can be summarised as:

1. Enumerate all the drivers that present the desired interface GUID.
2. For each interface, retrieve the interface details which include it's device path.
3. Use the device path to open a file handle with the device.
4. Using the file handle, issue the IOCTL's to the device passing memory pointers for the input buffer and output buffer.