Implementation and Analyses of the Mobile-IP Protocol

Under Windows™

SHANLUN JIN

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2008

COS/CCS 2008-17

Implementation and Analyses of the Mobile-IP Protocol

Under Windows™

Shaulun Jin

Abstract

This report is the result of a masters degree project conducted at the Department of Teleinformatics at the Royal Institute of Technology starting from the autumn 1996. The area investigated is the Mobile Internet Protocol, especially its implementation under Windows NT environment. Network driver writing under Windows NT was practised. Recent development in improving Mobile IP protocol to support micro-mobility have also been investigated.

Status: Final

List of Acronyms and abbreviations

ACM	Association for Computing Machinery
AMOS	Accumulated Mean Opinion Score
API	Application Programming Interface
ARP	Address Resolution Protocol
CCOA	Co-located Care of Address
CDFS	CD File System
CN	Correspondent Node
COA	Care of Address
DDK	Driver Development Kits
DHCP	Dynamic Host Configuration Protocol
DiffServ	Differentiated Services
DLL	Dynamically Loaded Library
DMSP	Designated Multicast Service Provider
DVMRP	Distance Vector Multicast Routing Protocol
FA	Foreign Agent
FCoA	Foreign Agent Care of Address
FCOA	
FSDs	File System Drivers
GFA	Gateway Foreign Agent
HA	Home Agent
HAL	Hardware Abstraction Layer
HMRSVP	Hierarchical Mobile Resource ReSerVation Protocol
HPFS	High Performance File System
ICMP	Internet Control Message Protocol
IDT	Interrupt Dispatch Tables
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IMHP	Internet Mobile Host Protocol
IntServ	Integrated Services
I/O	Input/Output
ioctl	Input/Output Control
IOCTL	
IP	Internet Protocol
IP V6	Internet Protocol Version 6
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IP-in-IP or IP-IN-IP	IP in IP encapsulation (a method of tunneling)
ipintr	a user space routine
IRP	I/O Request Packets

IRQL	Interrupt ReQuest Level
LAN	Local Area Network
LGHS	Low-latency Guarantee Handoff Scheme
MASR	Mobility Agent with SIP Registrar
MH	Mobile Host
MIP	Mobile-IP
MN	Mobile Network
MoM	Mobile Multicast
MOS	Mean Opinion Score
MOSPF	Multicast Open Shortest Path First protocol
MPLS	Multi Protocol Label Switching
MSDN	Microsoft Development Network
MS-DOS	Microsoft Disk Operating System
NAT	Network Address Translation
NDIS	Network Device Interface Specification
NetBEUI	NETBIOS Extended User Interface
NIC	Network Interface Card
NTFS	New Technology File System
OID	NDIS object identifiers
OS	Operating System
OSI	Open System Interconnection
PC	Personal Computer
PIM	Protocol Independent Multicast
QoS	Quality of Service
RSVP	Resource ReSerVation Protocol
RSVP-MP	Resource ReSerVation Protocol-Mobile Proxy
SDK	Software Development Kits
SIP	Session Initiation Protocol
SQPS	Scalable QoS Provisioning Scheme
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TOS	Type of Service
TTL	Time To Live
UDP	User Datagram Protocol
VDDs	Virtual DOS Drivers
VtoolsD	Development tool produced by Vireo Software, Inc.
Win32	Windows 32 bits system
Win95	Windows 95 operating system
WindowsNT	Windows NT operating system
WinMIP	Windows Mobile Internet Protocol

WWW	World Wide Web
XP	Windows XP operating system

1. Background

Today as powerful notebook computers and wireless communications proliferates, more and more data communication users require continuous connectivity to the Internet while they are moving around. However, present day internetworking protocols behave awkwardly when dealing with host migration from one network to another. These protocols makes an implicit assumption that the point at which a computer attaches to the Internet is fixed and its IP address identifies the network to which it is attached.

The development of Mobile IP [1, 2, 3, 4, 5] is to solve this problem. At the point when I started my thesis work, there had been several implementations of Mobile IP on SunOS or UNIX based systems. However, as we foresaw the proliferation of Windows operating systems based Personal Computers (PCs) both as servers and workstations, it made a lot of sense that Mobile IP should be implemented for systems running Microsoft's Windows operating systems.

Most of the work was done in the laboratory of and under the supervision of Prof. Dr. Gerald Q. Maguire Jr. . Ericsson Radio System AB provided financial support.

1.1 Mobile Internet Protocol (Mobile-IP)

Mobile-IP [1, 2, 3, 4, 5] developed by Mobile IP working group of the Internet Engineering Task Force (IETF) is an enhancement to IP which allows a computer to roam freely over different local networks while being reachable at the same IP address.

The Mobile-IP architecture, as proposed by the IETF, defines special entities called the Home Agent (HA) and Foreign Agent (FA) which co-operate to allow a Mobile Host (MH) to move without changing its IP address.

Each MH is associated with a unique home network as indicated by the network portion of its permanent IP address. IP routing always delivers packets meant for the MH to this (home) network. When a MH is away form this (home) network, a specially designated host (HA) on this network is responsible for intercepting and forwarding packets to the mobile. The MH uses a special registration protocol to keep its HA informed about its current network attachment point (i.e., the address that the HA should send packets to). Whenever a MH moves from its home network to a foreign network, or from one foreign network to another, it chooses a Foreign Agent (FA) on the new network and uses it to forward a registration message to its HA. After a successful registration, packets arriving for the MH on its home network are encapsulated by its HA and sent to its FA. Encapsulation refers to the process of enclosing the original datagram as data inside another datagram with a new IP header. The source and destination addresses fields in the outer header correspond to the HA and FA, respectively. This mechanism is also called tunneling since intermediate routers remain oblivious of the inner (original) IP header. Upon receiving the encapsulated datagram, the FA strips off the outer header and delivers the newly exposed datagram to the appropriate MH visiting on its local network.

Mobile-IP also allows a MH to do its own decapsulation. In this case, the MH must acquire a temporary IP address on the foreign network (e.g. using Dynamic Host Configuration Protocol (DHCP)) to be used for forwarding. Of course, it should still accept packets meant for its permanent IP address as this will be the destination address of packets following the decapsulation process.

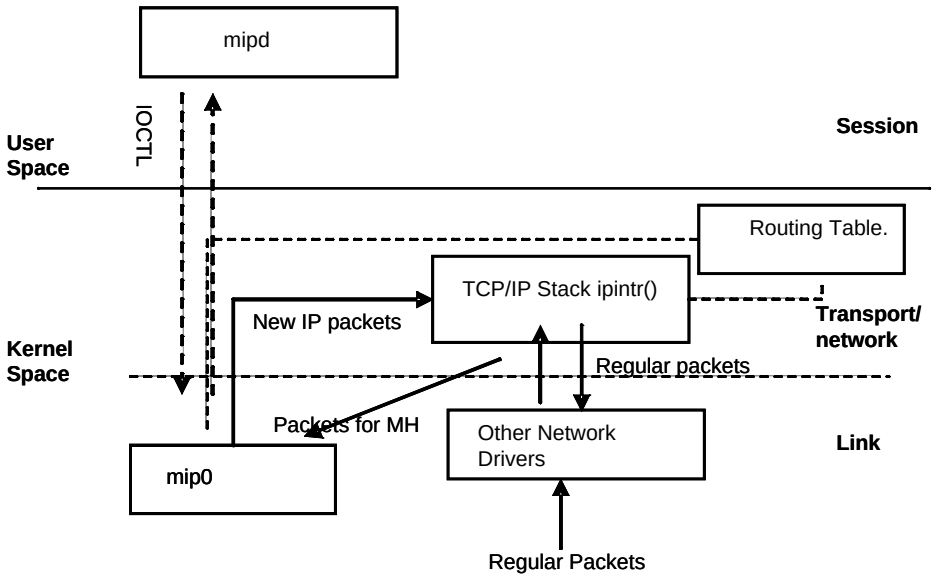
Overall Mobile IP supports mobile nodes moving between different networks without changing their IP address, thus allowing the upper level applications and users to be totally unaware of this roaming. However, while Mobile IP is well designed to support macro-level mobility, or so called “inter-domain” mobility, it is relatively poorly suited for micro-level (Intra-domain) mobility support, where applications can suffer due to packet loss and long latency during handoff. This will be discussed in detail in Chapter 9 of this thesis.

1.2 Previous Implementations

The simplest starting point seemed to be porting code from an existing implementations - even though this code might have been designed for another operating system and platform. At the time this thesis project began, there were several implementations of Mobile-IP on different operating systems, including: SunOS™, Solaris™, and Linux. The implementation that I studied was written by Anders Klemets for SunOS™. At that time, this was the only implementation whose source code was publicly available. This public availability was important, as one of the goals of this project was also to make the implementation publicly available - so that it could be used for research & development by others.

As Figure 1shows, Anders Klemet’s code for SunOS consists of two parts. A user space daemon called mipd that implements the Mobile Host, Foreign Agent, and Home Agent functions of the Mobile-IP protocol[1]. In addition, there is a kernel pseudo device driver which appears as network interface “mip0”. This driver is needed on computers that are going to act as a Foreign Agent or a Home Agent. The user application communicates with the driver through Input Output Control (ioctl) messages.

FIGURE 1. Anders Klemets’s implementaion of Mobile IP for SunOS



The driver is used to encapsulate or deencapsulate packets and it works as follows:

- When acting as a Home Agent, if mipd wants to establish a new tunnel, it will signal this to the device driver. Mipd will install a host route in the IP routing table for the mobile host whose packets will be tunneled. The host route will indicate that mip0 is the destination interface for datagrams addressed to this Mobile Host. Thus when a datagram destined to this MH is received, the ipintr () routine will pass it to the device driver. The driver will perform either IP-IN-IP [2] or Minimal encapsulation[3], as appropriate, and put the resulting datagram back into the IP input queue.
- Tunnel deencapsulation at a Foreign Agent is performed in a similar fashion. The device driver receives the incoming IP packets. The mipd program tells the driver that encapsulated datagrams for a particular MH should be deencapsulated, and indicates which interface the resulting datagrams should be forwarded to. As a result, the driver adds a host route in the IP routing table for this MH that points to the network interface on which it can be locally reached. After deencapsulating an incoming datagram, the driver puts the resulting datagram back in the IP input queue, where it will be taken care of by ipintr() and forwarded in accordance with the host route.

1.3 This Project

Despite several existing implementations, at the time of the start of this project there was no implementation for Microsoft's Windows™ operating system (which is the predominant operating system for PCs). This project was to design, implement, and evaluate Mobile-IP on Windows™ system.

1.3.1 General Plan

After studying Klemets' code, I made a plan of how to port it to Windows™. The ported code would also consist of two parts, an user application part corresponding to the mipd program, and a kernel mode driver part equivalent to the "mip0" driver.

Porting the user part of the code seemed quite straightforward. Although system calls such as those used for accessing IP routing table and communicating with the driver would be somewhat different, no other major changes should be made.

Porting the driver would be much trickier, since writing a Windows™ kernel mode driver is not something lots of people have experience with. However, I made an outline of how my driver should work. It would be quite similar to mip0 in terms of functionality, i.e. intercepting incoming IP packets, encapsulating or deencapsulating the packets of interests depending on the circumstances (whether the host acts as a HA or a FA) and forwarding the processed packets to the TCP/IP layer in the system.

However, I thought about the way the packets destined to MH are handled in a HA. As the driver is able to intercept all the incoming IP packets, why not grab these packets already at this stage, rather than let IP receive them first and forward them back to the driver later on? I thought my driver would probably be simpler: grab all the packets of interests (both those that should be encapsulated and those should be deencapsulated) and do the processing and put processed datagrams back into the IP input/output queue. Thus my driver would not need to receive datagrams from TCP/IP, which would be good.

1.3.2 Information Search

To begin with, I searched the WWW for information about the Windows™ operating system in general and in particular: device drivers and the architecture of

the network stack. However, the information I found was mostly very limited and not helpful for writing a functioning network driver.

At same time, I began to participate in different mailing lists concerning network programming and device driver development[1]. This provided access to expertise and experience in this area, that was unavailable elsewhere. These mailing lists functioned as a help desk throughout the project.

A tip I got from one of the mailing lists at that time was to look into Microsoft's Driver Development Kits (DDK) which is included in the professional level subscription to the Microsoft Development Network (MSDN). An alternative was a development tool called VtoolsD produced by Vireo Software, Inc. Since Windows™ is a Microsoft™ product, I opted for Microsoft's DDK, and it was the major development platform and information source for the whole project.

The Microsoft DDK provides some sample drivers so people don not have to write drivers from scratch. However, the samples were only available for Windows NT and developing drivers in NT was recommended by Microsoft™. Therefore I decided to do the implementation on NT. Another reason for doing that is PCs running as HA or FA would very likely be stationary servers using NT as their operating system. For more detail, see Appendix A.

1. I did not record the names of these mailing lists at that time; however, today there are many other similar information sources, such as BBS and Blogs. One of the blogs specializing in Windows driver writing that I go to a lot is:
   http://blogs.msdn.com/iliast/default.aspx

2. Understanding the Windows NT Drivers

Within the Windows NT operating system, there are two basic kinds of drivers: User-mode drivers, such as Win32 multimedia drivers, VDDs for MS-DOS® applications with application-dedicated devices, or another protected subsystem's drivers; Kernel-mode drivers for logical, virtual, or physical devices.

In order to process network packets quickly, my driver had to be implemented in the kernel. So in the rest of this section, we are going to discuss only Kernel-mode drivers.

Windows NT Kernel-mode drivers are part of the Windows NT executive, which is the underlying, "new technology" microkernel-based operating system. Like NT itself, NT drivers are implemented as discrete, modular components with a well defined set of required functionality. All NT drivers have a set of system-defined standard driver routines and some number of internal routines as determined by the driver writer.

2.1 Types of NT Drivers

There are three basic types of NT drivers. Each type has a slightly different structure and quite different functionality:

Device drivers, such as a keyboard or disk driver that directly controls a physical device. Device drivers are sometimes called lowest-level drivers, particularly when such a driver is the lowest driver in a chain of layered NT drivers.

Intermediate drivers, such as a virtual disk, mirror, or device-type-specific class driver, that depend on support from underlying device drivers

File system drivers (FSDs), such as the system-supplied FAT, HPFS, NTFS, or CDFS drivers, that also depend on support from underlying lower-level drivers.

While a particular NT file system driver might or might not get support from one or more intermediate drivers, every NT file system driver ultimately depends on support from one or more device drivers.

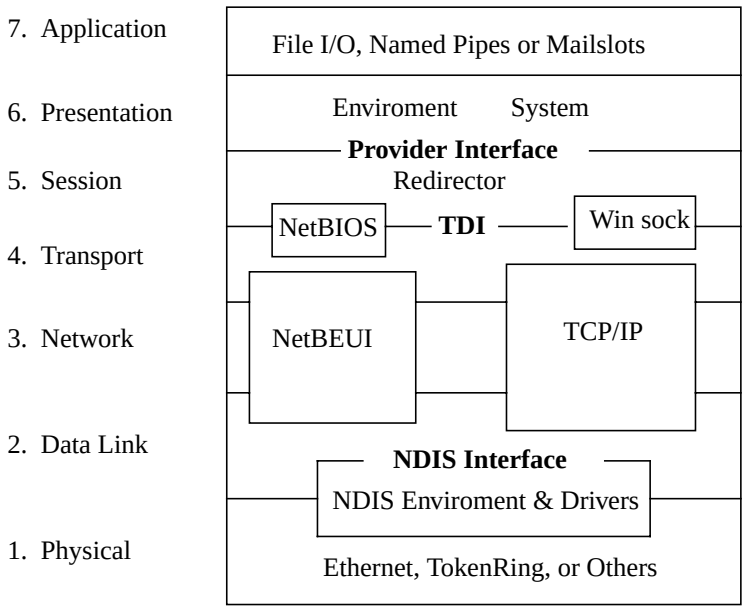
2.2 NT Network Driver

Theoretically any Windows NT network driver can be classified as one of these types of drivers. For example, an NT server or redirector is a specialized file system driver, a transport driver is a type of intermediate NT driver, and a physical netcard (sometimes called a media access controller) driver is an NT device driver. However, NT provides specialized interfaces and support for network drivers, such as NDIS 3.0 (Network Device Interface Specification, Version 3.0). In next section, we look at Windows NT network architecture, network drivers and NDIS in more detail.

3. Understanding Windows NT Network Architecture

Figure 2 below gives an overview of the Windows NT networking components. It also shows how they fit into the OSI reference model, and which protocols are used between layers. Components which are used in this implementation are described later in this section.

FIGURE 2. Windows NT networking components



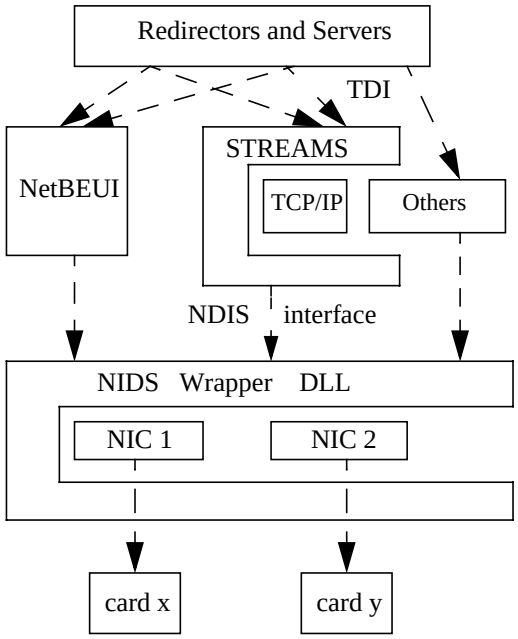
3.1 NDIS Environment and Network Drivers

Network Interface Cards (NICs) come with NIC drivers, which in the past often implemented a specific network protocol, such as IP. Because Windows NT allows many different protocol drivers to be loaded, each network card vendor using this approach would need to rewrite its drivers to support multiple protocols -- not an ideal strategy. To help vendors avoid this unnecessary work, Windows NT provides an interface and an environment called *Network Driver Interface Specification* (NDIS), which shields NIC drivers from the details of various protocols and vice versa. Figure 3 on the next page illustrates this.

Instead of writing a protocol-specific driver for Windows NT, network interface card vendors provide an NDIS interface as the uppermost layer of a single NIC driver. By doing so any protocol driver can direct its network requests to this network interface card by calling this interface. Thus, a user can communicate over a TCP/IP network and NetBEUI (or DECnet, NetWare, and so forth) network using one network interface card and a single NIC driver.

The NDIS interface was first available in LAN Manager, but was updated in Windows NT to NDIS version 3.0. Version 3.0. uses 32-bits addresses instead of 16-bit addresses, and is multiprocessor enabled. Like earlier versions, it can handle multiple independent network connections and multiple, simultaneously loaded network protocols.

FIGURE 3. Network Driver Interface Specification as a wrapper



3.2 Types of Network Drivers

Each NDIS network driver is responsible for sending and receiving packets over its network connection and managing the physical card directly or indirectly on behalf of the operating system. At its lowest boundary, the NDIS driver communicates directly with the card(s) it services, using NDIS routines to access them. The NDIS driver starts I/O on the card(s) and receive interrupts from them. At its upper edge, it calls upward to indicate that it has received data and to notify when it has completed an outbound data transfer.

Windows NT supports basically three types of network drivers:

3.2.1 Network Interface Card (NIC) drivers

These implement the Data Link Layer of the OSI model. They directly manage network interface cards at their lower edge and at their upper edge present an NDIS interface to allow upper layers to send packets via this interface to the network and to set the operational characteristic of the driver. This is also called a miniport driver interface.

3.2.2 Intermediate protocol drivers

These implement both the Network Layer and the Transport Layer of the OSI model. As the name suggests, they sit between the NIC drivers and upper level drivers. To the NIC drivers they look like protocol drivers, exporting a protocol driver interface. To the upper drivers they look like NIC drivers, exporting a miniport interface. One such driver can be layered on top of the another.

3.2.3 Upper level protocol drivers

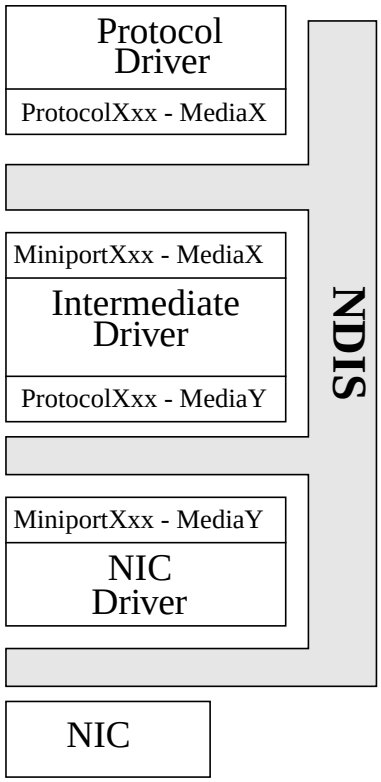
These implement an application-specific interface at its upper-edge to provide services to users of the network. Such a driver allocates buffers for packets, copies data from the sending application into the packet from at the upper edge of the

driver. At their lower edge, they provide a protocol interface to send or receive to or from the next lower level driver which is always an intermediate protocol driver.

Windows NT uses drivers at one or more network layers to pass data packets and make the protocol translation necessary for error-free communication.

The relationship between these three types of drivers is shown in Figure 4.

FIGURE 4. Relationship between the protocol, intermediate, and NIC drivers



3.3 NDIS Environment

NDIS defines three sets of functions: upper-edge functions, lower-edge functions and library functions, along with various objects and structures. Windows NT network drivers implement the former two sets of functions, call the library functions, and use the NDIS-defined objects and structures to accomplish successful transfer of data between user-applications and the physical link.

3.3.1 NDIS Upper-Edge Functions

NDIS upper_edge functions (MiniportXxx) are system-defined functions implemented in and exported by NIC drivers and intermediate drivers. NDIS NIC drivers should export all defined MiniportXxx functions while intermediate drivers export a subset of these functions at their upper edges, as shown in Figure 4. For example, an intermediate driver does not have to have functions such as MiniportISR or MiniportTimer, that a NIC driver must have to use to communicate with and control its NIC.

3.3.2 NDIS Lower-Edge Functions

NDIS lower-edge functions (ProtocolXxx) are system-defined functions implemented in and exported by NDIS drivers layered immediately above any driver that exports a set of NDIS upper-edge (MiniportXxx) functions. Both NDIS protocol drivers and intermediate drivers export a set of these ProtocolXxx functions at their lower edges.

Intermediate NDIS drivers appear to be NIC miniports to higher-level NDIS drivers that layered themselves above such an intermediate driver by establishing a binding. Intermediate NDIS drivers appear to be protocols to the NDIS drivers below them, including NIC drivers. To the NDIS library, such an intermediate driver appears to be both a protocol and a miniport driver. See Figure 4 on the previous page.

3.3.3 DriverEntry Function

NDIS-defined driver functions can have any name the driver writers chooses, expect for the DriverEntry function. The initial entry point of any Windows NT kernel-mode driver must have the explicit name DriverEntry in order to be loaded automatically by the system. The next section gives an introduction to Windows NT kernel-mode driver and in Section 6 there is more detailed information about the general requirements and functionality of the DriverEntry routine.

3.3.4 NDIS Library Functions

NDIS library is a Dynamically Loaded Library (DLL) containing a set of abstract functions that interface between protocol driver functions, intermediate driver functions, and NIC driver functions. It is used to submit a request to the operating system or cause a local action that does not require communication with other software functions. The main purpose of the NDIS library is to form a wrapper that allows network drivers to send and receive packets on a LAN or WAN in an operating system-independent manner. All the functions provided by the library have the names NdisXxx.

3.3.5 NDIS Object Identifiers

NDIS object identifiers (OID) are a set of system-defined constants of the form OID_XXX, that are used by higher level NDIS drivers in an NDIS_REQUEST-type structure for their calls to NdisRequest. Each request is classified as one of the following:

- A *query* is a call to retrieve information from or about the underlying NDIS driver, usually about the driver's or NIC's overall capabilities or current status. An NDIS protocol driver sets NdisRequestQueryInformation for the *RequestType* parameter to NdisRequest when it makes global queries. Such requests are handled by the MiniportQueryInformation functions of underlying NIC drivers.
- A *statistic query* is a call to retrieve information about network performance. Such a request always originates in a user-mode application. Protocol drivers never set NdisRequestQueryStatistics for the *RequestType* parameter to NdisRequest. Such requests are handled either by NDIS or by the MiniportQueryInformation functions of underlying NIC drivers.
- A *set* is a call with directions for the underlying NDIS driver, such as the header format the protocol wants to use for receive indications when the underlying drivers offers a choice or which optional features the miniport should enable on its NIC. An NDIS protocol driver sets NdisRequestSetInformation for the *RequestType* parameter to NdisRequest when it makes this type of request. Such requests are handled by the MiniportSetInformation functions of underlying NIC drivers.

Many system-defined OIDs are valid with more than one of the proceeding NdisRequestXxx values. Associated with each NDIS object identified by an OID_XXX is a data buffer, which varies in size and format depending on the given OID. The caller of NdisRequest supplies a pointer to this data buffer in the InformationBuffer member of the NDIS_REQUEST structure.

3.3.6 Structure Used by NDIS Drivers

The NDIS-defined structures are multifunction or alternative structure parameters. That is, either pointers to these structures are passed in calls to more than one NdisXxx, MiniportXxx, and/or ProtocolXxx function or a pointer to more than one of these structures can be passed in calls to the same function.

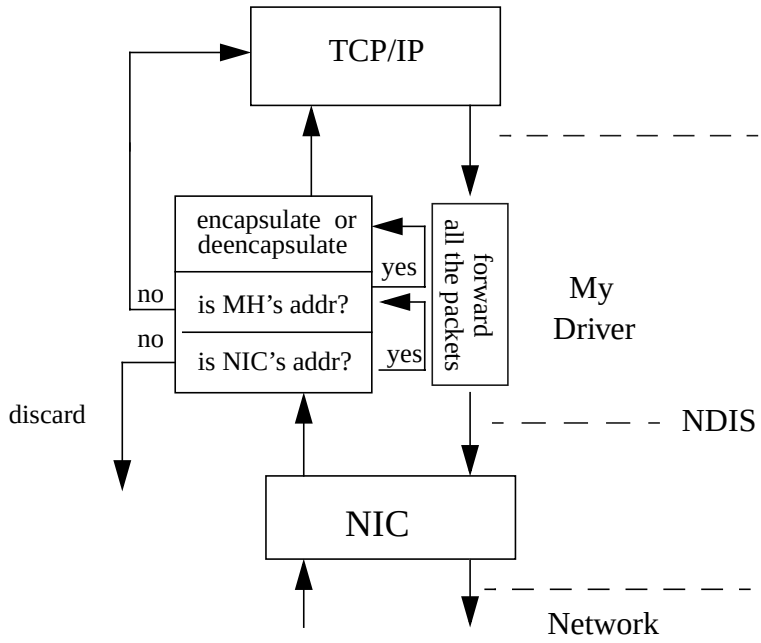
4. Defining Driver Type

As I mentioned earlier, my driver wants to have a peek at all incoming raw packets. So it could be either a NIC driver or an intermediate driver or a protocol driver, just layering on top of a NIC driver. Writing a NIC driver would involve knowledge of the hardware and make it hardware specific, which is an obvious disadvantage. Having excluded writing a NIC driver, there are still several means to achieve the result. Each approach has its advantages and disadvantages. I have examined three different options:

4.1 An Intermediate Driver

This driver sits between NIC driver(s) and the TCP/IP stack in the system. (see Figure 5) When acting as a HA, it grabs all the IP packets with the NIC(s)' hardware address(es) as its destination. Since the HA is doing a Proxy ARP for the mobile (which is away from the home network) packets destined for MH will have HA's link layer (hardware) address. The HA checks the IP destination address of each such packet. If it is the IP address of one of registered MHs, it encapsulates the packet and sends the resulting packet to the TCP/IP stack. After checking the routing table, TCP/IP will send the packet back to the driver, also indicating which network interface the packet should be sent to. The driver simply forwards the packet to the indicated interface. When acting as a FA, the situation is very similar.

FIGURE 5. An intermediate Driver



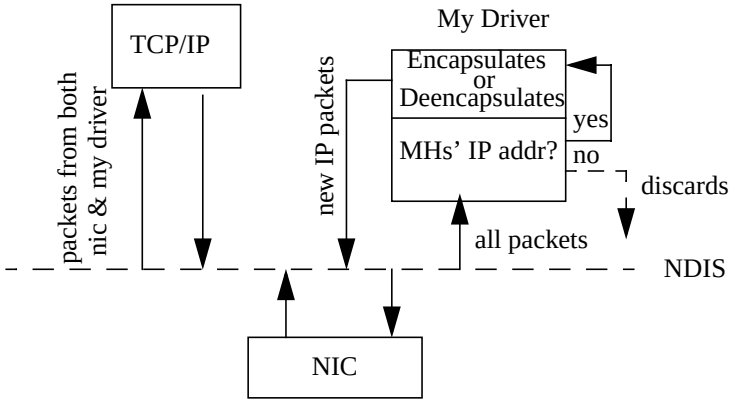
4.2 A Parallel Driver Interacting with TCP/IP

This driver sits parallel with the TCP/IP, as shown in Figure 6, which means that both the driver and TCP/IP are bound to the same NIC(s). It accepts all the incoming packets and processes each packets according to its IP destination. For IP packet belonging to one of registered MHs, the driver transmits it to NDIS after

having it encapsulated or deencapsulated. The TCP/IP stack will then detect from NDIS that there is a IP packet, and picks it up to eventually forward it to the correct network via a NIC. This assumes of course that the host is set up to do forwarding.

This approach requires however, that HA **must not** proxy ARP messages. If it did, then the IP stack would also receive the packets destined to MHs. Since the hardware address and IP address in each such packet does not match (i.e., it has the HA's hardware address, but the MH's IP address), then IP would send an ICMP error message to the sender, which would prevent the sender from sending additional packets!

FIGURE 6. A Parallel Driver Interacting with TCP/IP



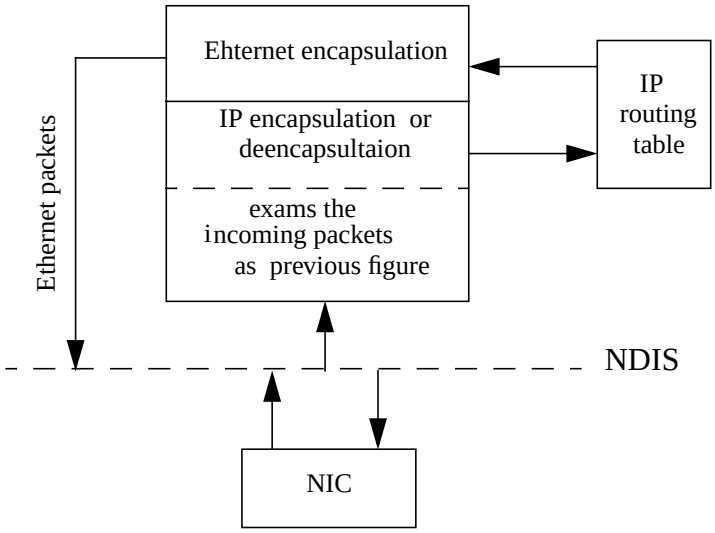
4.3 A Stand-alone Driver

This driver also sits in parallel with TCP/IP, but doesn't use any service from TCP/IP (see Figure 7). It differs from the previous driver described in Section 4.2 in the way it sends processed packets. Instead of forwarding these "bare" IP packets to TCP/IP via NDIS, it adds an Ethernet header to each packet and sends the resulting ethernet frame to the NIC(s), which in turn will transmits the packet directly to via the network interface.

This requires that the driver maintains an IP routing table from where the driver can get the hardware address and network interface for each IP destination. By using this information, the driver is able to add an Ethernet header to each packet correctly and send them to the appropriate NIC(s).

For the same reasons as explained in Section 4.2, the driver MUST NOT send any Proxy ARP messages when acting as a HA.

FIGURE 7. A Stand-alone Driver



4.4 Comparison of these three options

From an implementation complexity point of view, the second choice is the best. It only has lower edge interface towards NIC, while writing a driver as described in Section 4.1 would involve both sides of the interfaces. Option 3 also has only lower edge interface, but it has to maintain its own IP routing table, which adds to its complexity.

However from a performance point of view, option 3 should be the best. It doesn't involve the IP stack, and hence would certainly speed up the process of packets. However, one of the drawbacks for both options 2 and 3 are that both drivers have to accept almost all the incoming IP packets, while in option 1 the driver only accepts those with the host's hardware address. When acting as a HA, a little optimization could be done by discarding those packets destined to the local host, precisely opposite to what a bound TCP/IP stack does. This is possible, because in the absence of this host doing a Proxy ARP, these packets are only for the local host and hence are certainly not destined any MH.

Another advantage of adopting option 3 is that driver can be used in a system, such as Windows 95 where the normal TCP/IP stack doesn't support IP routing. Unlike Windows NT which does support routing. Actually by putting some additional functionality into this driver, it could turn the host into a router (see Section 5.2.5).

4.5 A Protocol Driver - WinMIP

After considering all the advantages and disadvantages, I selected option 3. Option 3 had a clear advantage compared to the first 2 options in terms of performance; as it does not involve the built-in TCP/IP stack. This proved to be a correct decision, as a later implementation done by three Nokia engineers on Windows 2000 system showed that TCP/IP related operations, such as changing routing table are very time consuming. (See Section 9.2).

Therefore, the driver was designed to be a minimal protocol driver that only has a lower edge interface implementing NDIS *protocol xxxx* functions. Also necessary were functions for loading, unloading, configuring the driver, some signaling

between the user application and the driver, etc. I named this driver WinMIP. In Chapter 5 I will describe the major phases of implementing this driver, and in Chapters 6 and 7 more technical details about writing a NT protocol driver are presented.

5. Implementing the Driver in Windows NT

5.1 Tools and platform

I used Microsoft Windows NT 4.0 for Workstation as the operating system to do my development and used the tools:

- DDK for Windows NT 4.0,
- Win32 SDK, and
- Virtual C++ 4.0 Beta

I based my driver on the “Packet” sample code from DDK for Windows NT 4.0. The sections below will describe this DDK environment and the sample code in more detail.

5.1.1 The DDK Environment

The DDK provides two versions of the Windows NT operating system, called the *free build* and the *checked build*. The *free build* of Windows NT is the retail version that end users buy. Free binaries are built with full optimization and contain a minimal set of debugging symbols, such as function entry points and global variables. The *checked build* is shipped only with the DDK; it is used in debugging drivers and other system code. Checked binaries provide error checking, argument verification, and system debugging code not present in the free binaries.

A device driver is a trusted part of the NT kernel. In between the kernel and user programs is the Win 32 subsystem which implements the Win 32 API and manages the screen, keyboard, mouse, parallel port, etc. Drivers are controlled by the I/O Manager and talk to the underlying electronics using NT's Hardware Abstraction Layer (HAL).

Device drivers are therefore not Win 32 programs so they can not use Win 32 API. An NT driver can’t link with the C runtime library, as linking would significantly bloat the driver’s load image with a set of functions specific to the Win32 programming environment. For the same reason, NT drivers can’t call user-mode C runtime routines such as printf() from kernel mode, since these routines are only designed to be called in user mode. Instead drivers would need to call various NT kernel routines in order to control, send I/O request to hardware devices.

Therefore, DDK supplies an alternative set of system routines and a set of kernel-mode C runtime routines declared in *ntddk.h* respective *ntdef.h*, which an NT driver can call. For example, DDK provides DbgPrint routine which is very similar to the regular printf routine, the difference being that it can only be called from kernel mode. The debug messages, instead of displaying on the screen, are sent to the kernel debugger (usually WinDbg), assuming that one is attached. For debugging purposes it is common for drivers to use the DbgPrint to emit debug messages. However it can be easily used for console display, like through the course of my work.

5.1.2 The Packet Sample

The “packet” sample driver is a typical protocol driver. After being loaded it opens and binds to an underlying NIC driver upon request from a user level application. It also sets the NIC driver into promiscuous mode (i.e., the NIC will then accept all packets) and returns the raw packets from the NIC if the user application so requires.

5.2 Major steps in writing the driver

5.2.1 Modifying Sample

I modified the “packet” sample from DDK, removing the upper level control part of the code. So after being loaded the driver was supposed to open and bind to the underlying NIC and then continuously receive network packets from the NIC without any intervention from an upper level application. I also configured the driver so that it should be able to look at all the network packets. The driver was successfully loaded (no error message or warning upon the start-up of the system), and nothing changed. The system and all the network applications worked as usual. This result is actually expected, since whatever packet the driver receives, it simply puts the packet back. Therefore other protocol stacks should not be affected.

5.2.2 Intercepting IP Packets

In order to continue the implementation, I made a test to let the driver intercept all the IP packets. First I changed the loading order of all the existing network drivers in the system, which are bound to the same NIC as my driver, so my driver is loaded first when the system starts. I thought that in this way, my driver would be able to receive the packets before the other drivers. If the driver, then altered the content of all the IP packet’s LookAheadBuffer (similar to the type field in Ethernet header) before putting them back, the TCP/IP stack wouldn’t recognize them and pick them up again. Consequently all the applications based on TCP/IP would not be able to work on this host. Unfortunately this was not the case. Initially, I though that there could be two reasons, either the underlying NIC driver doesn’t inform the incoming packets in a serial manner or the binding with the NIC driver does not depend on the loading order of protocols.

I asked the question to the same news groups in which I had been participating since the beginning of the project. One of the messages there pointed out that being loaded first doesn’t mean that a driver will be informed of receipt of a packet first, since NDIS informs **all** the binding drivers at almost the same time. Or more exactly, NDIS dynamically determines the order of informing drivers of a received packet based on the previous packet’s dispatch in order to optimize the performance of the network code. i.e., if my driver accepts the previous packet, it is most likely to be informed by NDIS of the current packet earlier than a protocol driver that did not accept the packet. This is largely because my driver has already created a memory cache in the NDIS before others.

So the conclusion is that my original thought of having my driver grab all the packets before others did not work. However, this might not actually be an issue for my implementation. When a mobile node is registered using the FA’s care-of address packets destined to mobile node are encapsulated using IP-in-IP encapsulation[2], Hence the TCP/IP stack in the FA would receive the encapsulated packets, but it would not recognize the value in the *protocol* field, ‘4’ in this case, in the IP header, since it is not told to handle IP-in-IP decapsulation. Hence it will most likely discard the packets silently. For the HA, these packets would not even be seen by TCP/IP since the network interface address does not match.

The result is that the failure to be able to create a driver which operates on received packets before other driver, was not a hindrance to my implementation. Even though it did cause some concerns at the time.

5.2.3 A Packet Dump Application on Windows NT

I decided to go head with implementing HA functionality first. Until this point, I had not seen any ‘real’ packets coming in or going out. How could I be sure that the driver really can receive all the network packets?

As suggested by Dr. Maguire, I wrote a console application which simply dumps all the packets the driver receives to the console window. The program involves some I/O control between a user application and a driver. Readers who are interested in the program can look into Appendix B for the source code and Section 7 for how I/O control works in the Windows NT environment. Please note that the driver code needs to be modified in order to be able to communicate with the console application. All I/O related routines for the driver are in the file Rawupper.c in Appendix A.

The dump screen looked like this:

```
==================================================================
Press Ctrl-C To Exit...
Packet No.: 0x0000000000000575   Time: 0x00056152   Length: 60
Destination: 00.C0.6C.33.69.41   Source: 08.00.89.A0.90.66
HexDump: 60 Bytes
000000:  00 C0 6C 33 69 41 08 00: 89 A0 90 66 00 1D AA AA    ..l3iA.....f....
000010:  03 08 00 07 80 9B 04 15 : 00 00 04 80 01 47 14 D8    .............G..
000020:  80 B5 03 C0 FF CA BC 09 : 03 00 02 00 00 00 00 00    ................
000030:  00 00 00 00 00 00 00 00 : 00 00 00 00                ................

Packet No.: 0x0000000000000576   Time: 0x00056159   Length: 60
Destination: 00.C0.6C.33.69.41   Source: 08.00.89.A0.90.66
HexDump: 60 Bytes
000000:  00 C0 6C 33 69 41 08 00 : 89 A0 90 66 00 1D AA AA    ..l3iA.....f....
000010:  03 08 00 07 80 9B 04 15 : 00 00 04 80 01 47 14 D8    .............G..
000020:  80 A5 03 60 FF CA BD 09 : 03 00 03 00 00 00 00 00    ...‘............
000030:  00 00 00 00 00 00 00 00 : 00 00 00 00                ................

==================================================================
```

This screen is similar to that of tcpdump! With some small modifications, i.e. by having the driver only pick up IP packets on the network, this is actually a tcpdump application on Windows NT.

5.2.4 Adding HA functionalities

As I mentioned in previous section, when a stand-alone driver is running in parallel with the TCP/IP stack acts as a HA, it must not send Proxy ARP. Hence the packets destined to the MN would not have HA’s hardware address. To save the processing capacity and enhance the performance, I have the driver only pick up IP packets with a hardware address that is different from its own NIC’s address. I hard-coded it in the driver, which means that if I want to install this driver onto another host, I have to first change the code and rebuild the driver. However, as suggested by Dr. Maguire, I later on found it largely meaningless - since if the link layer address is that of the HA, the driver can simply ignore it - while if it is for another host, then the driver will process it to see if it should be for a MH. In any case, my stand-alone driver should be the first one that responds to the MH’s hardware address in the home network when MH is not attached to the home network.

Since these received packets are raw network packets, the driver has to first remove their link layer (Ethernet in this case) header. The next step is check of the packet is for a mobile node which this home agent has an address for - if so then the driver does IP in IP encapsulation[2]. This places a new IP header before the original IP datagram. The new IP header should have the HA’s IP address as its source IP address, the FA’s IP as the destination IP address and with a newly computed total length and header checksum.The calculation of the total length of the new IP packet is straight-forward, i.e. new total length = header length (usually 20 bytes) +

old total length. To recompute the IP checksum for an outgoing datagram, the value of the checksum field is first set to 0. Then the 16-bit one's complement sum of the header is calculated (i.e. the entire header is considered as a sequence of 16-bit words). The 16-bit one's complement of this sum is stored in the checksum field. When a IP datagram is received, the 16-bit one's complement sum of the header is calculated. Since the receiver's calculated checksum contains the checksum stored by the sender, the receiver's checksum is all one bits if nothing in the header was modified[7]. The Time To Live (*TTL)* field should be decremented by 1 if the HA also functions as a router.

5.2.5 An NT Router

It seems that the driver is able to pick up all the network packets and modify them, but how about sending them back to the network?

In order to check this, I did a test, using two Windows NT machines in the same LAN, Bob and Spy. I installed the TCP Dump console application on Spy and configured it so that only IP packet with its own IP as destination address and with Bob's IP as source address will be dumped to the console. In Bob I loaded the driver which encapsulates all the incoming IP packets with Spy's IP address. Packets appeared on the dump screen of Spy, and they are encapsulated which could only happen if Bob was able to encapsulate and forward these packets.

As discussed in the previous section, a stand-alone driver could function as a router if it maintains its own routing table. This would require using two network cards connected to two separate segments of the network. Then the driver would bind to both NICs. Using a look-up table, the driver could map the destination IP address into an Ethernet address and network interface. It would then wrap the IP packet with an Ethernet header and send it to the correct interface. I suspect that the built-in IP routing feature in Windows NT is implemented in a similar way.

5.2.6 Implementing FA Functionality

Adding FA functionalists onto the driver should be pretty simple. After deencapsulating the datagram, the driver examines the destination IP address in the original IP packet. If the IP address is for one of the mobile clients it is acting for, then it would construct a Ethernet header with the correct network interface address, put it before the IP datagram, and send the resulting frame out the correct interface. Since most of the mobile hosts will be attached to the network via wireless link when visiting a foreign network, FA will mostly likely be dealing with more than one network interface. Probably with one interface for the wireless link and one for the fixed connection. In such a case, the FA should decrement the value in the *TTL* field in the original IP datagram by 1 before forwarding it.

Due to the time constraint and difficulties with setting up the test environment, I was not able to implement the FA functionality.

6. NT Protocol Driver

6.1 Basic structure

My WinMIP driver consists of three blocks of functions: Loading, Network, and Signalling. These blocks work relatively independently, which made it easy for me to concentrate on writing one block at a time.

6.2 Loading and Binding

As I mentioned in Section 3, a network driver's initial required entry point must be explicitly named DriverEntry so that the loader can identify it. This DriverEntry does the following:

6.2.1 Register the Protocol Driver

To set up communications with the NDIS library, a driver registers as a protocol driver by calling NdisRegisterProtocol in the DriverEntry routine.

Before making the call, the DriverEntry zero-initializes a structure of type NDIS_PROTOCOL_CHARACTERISTICS, and then stores the addresses of the mandatory Protocolxxx functions, as well as any optional Protocolxxx functions the driver exports, in the characteristics structure. Some of these functions will be discussed later, however, for a full list of these functions, readers are referred to the DDK documentations[6].

6.2.2 Opening and Binding an Adapter

Having been registered, the driver does two things:

First, the driver reads the registry information stored during setup to build a list of names of adapters to which it will bind. The registry contains binding information written when the network is configured. DriverEntry reads this information, including the names of the adapter or adapters to which it can bind, from the HKEY_LOCAL_MACHINES\CurrrentControlSet\Services*ProtocolComponentName*\ Linkage key in the registry.

The driver then calls NdisOpenProtocolConfigration to obtain a handle to the registry key at HKEY_LOCAL_MACHINES \CurrrentControlSet \Services*DeviceInstance* \Parameters*ProtocolName*, where the protocol driver can store adapter-specific information. Once this handle is obtained, the protocol driver uses NdisReadConfiguration functions to read adapter-specific information: the name of the NIC driver and the corresponding driver object.

Second, after the protocol driver has retrieved the information it requires from the registry and has registered by calling NdisRegisterProtocol, and before the protocol can send packets and receive incoming data, it must bind itself to one or more NICs managed by underlying NDIS driver(s). The protocol binds itself to an underlying NIC and the driver that controls it by calling NdisOpenAdapter. NDIS returns a handle to the protocol driver. The protocol driver passes this handle to NDIS in future calls relating to this binding, such as calls to send packets (NdisSend or NdisSendPackets).

If the protocol driver can't successfully bind to an underlying adapter, it deallocates any resources it previously allocated for the adapter. If the protocol driver can't successfully open any of the possible adapters, it deallocates any global resources

the protocol has previously allocated and return an appropriate failure status. In this case, the protocol driver will subsequently be unloaded.

6.2.3 Query and Set Operations

After a protocol driver has determined which adapters to open and has successfully bound to one or more, it can call NdisRequest to query the underlying NDIS driver for its characteristics and to set the protocol's own operating characteristics. The protocol can also negotiate certain parameters for the binding.

For example, a protocol driver can query the largest packet the underlying NIC driver can accommodate on the NIC it manages. The protocol driver must restrict the size of the packets it subsequently sends to be no larger that this size. It's an error for a protocol driver to submit a larger packet to the underlying NIC driver than the NIC driver has indicated it can support.

DriverEntry also allocates any resources the protocol needs to operate, such as a packet pool from which the protocol driver can allocate packet descriptors for receiving and transferring data. If DriverEntry is unable to allocate the requested packet pool, it releases any previously allocated resources, makes a call to NdisDeregisterProtocol if necessary, and returns an appropriate error status.

6.3 Network

This is the most important part of the driver. It should be fulfill the functions described in the following sections.

6.3.1 Receiving Incoming Raw Packets

When a packet arrives from the network, a NIC driver calls NdisMIndicateReceivePacket, passing a pointer(s) to one or more full packets and relinquishing ownership of the resources for these packets to the overlying drivers. NDIS then calls the ProtocolReceive function. This function does the following:

Peek at the LookAheadBuffer provided by NDIS to see if it is a packet of interest, in this case, to see if it is an IP packet. If it is, then it copies the LookAheadBuffer into an internal buffer. Then it allocates sufficient space for transferring the rest of the packet. After that it calls NdisTransferData with the packet descriptor, so the NIC driver copies the rest of the received data into the protocol‘s buffers (from the packet pool). When NdisTransferData returns STATUS_SUCCESS or the ProtocolTransferDataComplete function is called, the lookahead buffer is chained to the buffers containing transferred data. Thus it has a copy of the original packet which will be processed in the ProtocolReceiveComplete function.

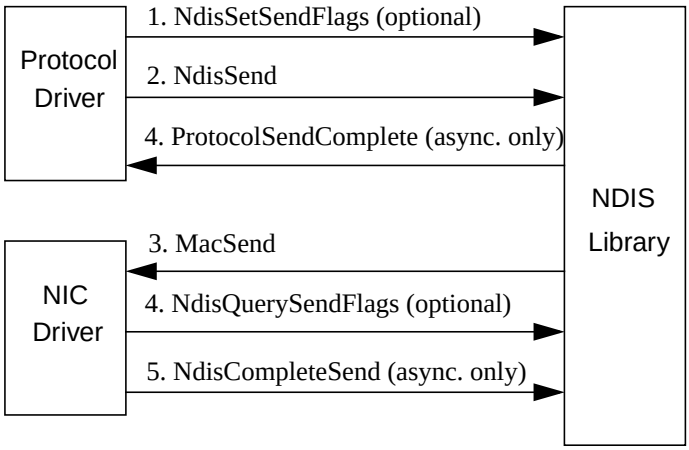
The ProtocolReceive function is designed to execute as quickly as possible, because it is holding the ownership of the indicated packet until it returns. Holding a packet for too long will certainly cause bad performance on the network and or even loss of coming packets. For this reason, I postpone processing the data until the ProtocolReceiveComplete function, which will be called after the ProtocolReceive function has returned.

6.3.2 Sending Packets

A protocol driver can transmit a single packet by calling NdisSend, passing in a pointer to a packet descriptor with chained buffer descriptors mapping the buffered data to be sent. Alternatively, a protocol driver can transmit several packets using NdisSendPackets, passing in a pointer to an array of pointers to one or more

packet descriptors. Which of the two is used, depends on the driver's own requirements and on the characteristics of the underlying NIC driver. By calling NdisRequest, with an OID_GEN_MAXIMUM_SEND_PACKETS query, the driver can determine the maximum number of packets to be sent that the underlying driver will accept in a packet array. If NIC driver only supports one at a time, then the protocol driver should use NdisSend to send packets. Otherwise both drivers' performance will be better if the protocol driver sends an array of packets with NdisSendPackets.

FIGURE 8. Using NdisSend



I put the sending function code in the ProtocolReceiveComplete, after the datagrams have been processed. NDIS always calls ProtocolReceiveComplete after one or more calls to the driver's ProtocolReceive function, regardless of whether any particular packet(s) are accepted by the bound protocol. Therefore I maintain a sending queue. Every time a packet comes, ProtocolReceive inserts a preallocated packet descriptor into the queue, and calls NDIS to copy the data of the incoming packet into the space mapped by the packet descriptor. When ProtocolReceiveComplete is called, the driver picks out packets one by one from the sending queue, processes them and sends them away individually by calling NdisSend.

The access to the sending queue is also synchronized by using a SpinLock, since both ProtocolReceive and ProtocolReceiveComplete may access the queue at the same time. (NDIS will call ProtocolReceive as soon as a packet arrives, a ProtocolReceive operation maybe in process at that time.)

6.4 Signalling

- I/O Request Packets (IRP) are the basis of all interactions between a device driver and its corresponding device. Each driver-specific I/O stack location in every IRP has a major function code (IRP_MJ_XXX) that tells the driver what operation it or the underlying device driver should carry out to satisfy the I/O request. Each Windows NT kernel-mode driver must set up one or more dispatch entry points for the required subset of system-defined major function codes that are set in the I/O stack location(s) of IRPs. The subset function codes that a

device driver must handle depends on the nature of it's device. (The subsets are defined in the *Kernel-Mode Driver Reference* - a Windows NT DDK document.) However, NT device and intermediate drivers usually handle the following set of basic requests:

- IRP_MJ_CREATE - open the target device object, indicating that it is present and available for I/O operations
- IPR_MJ_READ - transfer data from the device
- IPR_MJ_WRITE - transfer data to the device
- IPR_MJ_DEVICE_CONTROL - set up (or reset) the device, according to a system-defined, device-type-specific I/O control code
- IRP_MJ_CLOSE - close the target device object

For more information about the major function codes and device I/O control codes that NT drivers for particular kinds of devices are required to handle, see the *Kernel-Mode Driver Reference* from Windows NT DDK document.

6.4.1 IOCTL Interface

Windows NT (and also Win95) includes a device input and output control (IOCTL) interface that allows applications developed for the Microsoft Win32 application programming interface (API) to communicate directly with device drivers. Applications typically use this interface to carry out selected MS-DOS system functions, to obtain information about a device, or to carry out input and output (I/O) operations that are not available through standard Win32 functions.

Using the IOCTL interface in user applications will be discussed in the next section. We will also see how it is implemented in WinMIP driver.

6.5 IRQL (Interrupt ReQuest Level)

According to the definition of *Kernel-Mode Driver Reference* from Windows NT DDK, IRQL is the priority ranking of an interrupt. A processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation.

The particular IRQL at which a piece of kernel-mode code executes determines its hardware priority. Kernel-mode code is always interruptible: an interrupt with a higher IRQL value can occur at any time, thereby causing another piece of kernel-mode code with the system-assigned higher IRQL to be run immediately on that processor. In other words, when a piece of code runs at a given IRQL, the Kernel masks off all interrupt vectors with a lesser or equal IRQL value on the microprocessor.

IRQL is a very important issue when writing a network driver. Every driver function called by NDIS runs at a system-determined IRQL, one of PASSIVE_LEVEL < DISPATCH_LEVEL < DIRQL. For instance, the initialization function, halt function, reset function, and sometimes the shutdown function run at PASSIVE_LEVEL. Interrupt code runs at DIRQL. All other NDIS driver functions run at DISPATCH_LEVEL.

Running at IRQL DISPATCH_LEVEL or higher prevents threads (even those with the highest real-time priority level) from running on the same processor until the current kernel-mode routine lowers IRQL. However, running at raised IRQL on a given processor has no effect on the IRQL of any other processor in a symmetric multiprocessor machine.

The IRQL at which a driver runs affects which NDIS functions it can call. Certain functions can only be called at IRQL PASSIVE_LEVEL. Others can be called at DISPATCH_LEVEL or lower. However, at the beginning, I was not aware of this, and frequently caused system failure. So before you write a NDIS driver, check **every** NDIS function for IRQL restrictions.

When an interrupt occurs, it is handled (serviced) by a function called an Interrupt Service Routine (ISR). Data structures called Interrupt Dispatch Tables (IDT) track which interrupt service routine(s) will handle the interrupts occurring at each IRQL. A separate IDT is associated with each processor. Because of this, each processor can potentially associate different interrupt service routines with the same IRQL, or one processor can be asked to handle all interrupts.

Any driver function that shares resources with the driver's ISR must be able to raise its IRQL to DIRQL to prevent race conditions. NDIS provides such a mechanism. For example, obtaining a *Spin Lock* by calling NDISAcquireSpinLock function would raise the IRQL level of a function to DIRQL.

A device driver associates its interrupt service routine with an IRQL by constructing an *Interrupt Object* and passing it to the kernel. The kernel then connects the interrupt object to the appropriate interrupt dispatch table entry. When an interrupt occurs at the device's IRQL, the kernel locates the device driver's ISR using the interrupt object. More than one interrupt object can be associated with each IRQL, so multiple devices could potentially share the same IRQL. When a device driver is unloaded, it simply asks the kernel to disconnect its interrupt objects from the interrupt dispatch table. Interrupt objects increase device driver portability by providing a way to connect and disconnect ISR without needing direct access to the kernel's interrupt dispatch table.

7. Device I/O Control in Applications

As I mentioned before, people use the device IOCTL interface in an application to carry out “low-level” operations that are not supported by the Win32 API and that require direct communication with the driver. In section 6, I have described how it is implemented in the WinMIP driver. In this section, I am going to describe how to use this interface in user applications. In the packet dump application I wrote, IOCTL is heavily used. I will use some relevant code from this application as examples.

7.1 General

Windows NT implements the interface through the DeviceIoControl function, which sends commands and accompanying data directly to the given driver. To use this interface, you open the driver by using the CreateFile function, send commands and data by using DeviceIoControl, and finally close the driver by using the CloseHandle function.

7.2 Opening the Driver

You can open a static or dynamically loadable driver by specifying the module name, or registry entry identifying the driver in a call to the CreateFile function. If the driver exists and it supports the device IOCTL interface, the function returns a device handle that you can use in subsequent calls to the DeviceIoControl function. Otherwise, the function fails and sets the last error value to ERROR_NOT SUPPORTED or ERROR_FILE_NOT_FOUND. You can use the GetLastError function to retrieve the error value.

When you open a driver, you must specify a name having the following form:

```
\\.\DriverName
```

DriverName can be the module of the driver, the name of the driver file, or the name of a registry entry that specifies the filename. CreateFile checks for a filename extension to determine whether *DriverName* specifies a file. If a filename extension (such as *.SYS* in Win NT, *.VXD* in Win95) is present, the function looks for the file in the standard search path. If *DriverName* has no filename extension, CreateFile checks the registry. If the driver is loaded as a protocol driver such as my WinMIP driver, CreateFile will look into *HKLM\SYSTEM \CurrentControlSet \ Services* to see if the DriverName is one of the *DeviceInstance*s. If the *DriverName* is a value name, CreateFile uses the current value associated with the name as the full path of the driver file. This method is preferable, because sometimes driver files may not be in the standard search path. The packet dump application also uses this method to open the MIP driver.

If *DriverName* has no filename extension and is not in the registry, CreateFile assumes that the name is a driver module and searches the internally maintained device descriptor blocks for an already loaded driver having the given name.

You can open the same driver any number of times. CreateFile provides a unique handle each time you open a driver, but it makes sure that no more than one copy of the driver is loaded into memory. To ensure that the system removes the driver from memory when you close the last instance of the driver, use the FILE_FLAG_DELETE_ON_CLOSE value when opening dynamically loadable drivers. A static driver cannot be removed from memory.

Although CreateFile has several parameters, only the *lpName* and *fdwAttrsAndFlags* parameters are useful when opening a driver. *fdwAttrsAndFlags* can be zero, the FILE_FLAG_DELETE_ON_CLOSE value, or the FILE_FLAG_OVERLAPPED value. The FILE_FLAG_OVERLAPPED value is used for asynchronous operation and is described later in this section.

7.3 Sending Commands

You use DeviceIoControl to send commands to a driver. You must specify the previously opened device handle, control code, and input and output parameters for the call. The device handle identifies the driver, and the control code specifies an action for the driver to perform. In the following example, which is extracted from my packet dump application, the IOCTL_MIP_STOP_RECEPTION control code directs the given driver to stop receiving data packets.

Example:

```
HANDLE hDevice;
DWORD  nBytesReturned;

DeviceIoControl(
      hDevice,                      // the handle to the driver
      IOCTL_MIP_QUERY_LOWER_INFO,  // control code
      pSendData,       // buffer for input data to the driver
      64,                           // size of the buffer
      NULL, 0,         // no output data from the driver
      &nBytesReturned,
      &Overlapped                   //for asynchronous operation
      );
```

The input and output parameters of DeviceIoControl include the addresses and sizes of any buffers needed to pass data into or out of the driver, Whether you use these parameters depends on how the driver processes the control code. You supply an input buffer if the driver requires that you pass it data for processing, and you supply an output buffer if the driver returns the results of processing. In the previous example, only the input parameters are supplied.

Although the Win32 header files defines a set of standard control codes, Windows NT does not support these standard codes. Instead the meaning and value of control codes in Windows NT are specific to each driver. Different drivers may support different control codes. All the control codes for the WinMIP driver are defined in the file *IOCTL.h* in Appendix B.

If you opened a driver using the FILE_FLAG_OVERLAPPED value, you must also provide an OVERLAPPED structure when calling DeviceIoControl. This structure contains information that the driver uses to process the control code asynchronously.

7.4 Closing a Driver

When you have finished using a driver, you can close the associated device handle by using the CloseHandle function, or you can let the operating system close the handle when the application terminates.

Closing a driver does not necessarily remove the driver from memory. If you open a dynamically loadable driver, such as my WinMIP driver, using the FILE_FLAG_DELETE_ON_CLOSE value, CloseHandle also removes the driver if no other valid handles are present in the system. The system maintains a reference count for dynamically loadable drivers, incrementing the count each time the driver is opened and decrementing when the driver is closed. CloseHandle checks this count and removes the driver from memory when the count reaches zero. The system does not keep a reference count for static drivers; as it does not remove these drivers when their corresponding handles are closed.

In rare cases, you may need to use the DeleteFile function to remove a dynamically loadable driver from memory. For example, you use DeleteFile if another application has loaded the driver and you simply want to unload it. You also use DeleteFile if you successfully loaded a driver by using CreateFile, but the driver provides no handle to close and remove the driver. However, you should generally avoid using DeleteFile to remove a driver from memory; CloseHandle is a more appropriate approach.

7.5 Asynchronous Operations

You can direct a driver to process a control code asynchronously. In an asynchronous operation, the function returns immediately, regardless of whether the driver has finished processing the control code or not. Asynchronous operation allows an application to continue while the driver processes the control code in the background. As mentioned in the previous subsections, you request an asynchronous operation by specifying the address of the OVERLAPPED structure in the DeviceIoControl function. The *hDevice* member of OVERLAPPED specifies the handle of an event that the system sets to the signaled state when the driver has completed the operation

To perform an asynchronous operation, you must specify the FILE_FLAG_OVERLAPPED value when calling CreateFile to obtain a device handle. When calling DeviceIoControl, you must specify the address of an OVERLAPPED structure in *lpOverlapped* parameters. Additionally, you should also specify a handle of a manual reset event in the *hEvent* member of the structure. The system ignores all other members. This handle is obtained by calling function CreateEvent.

If DeviceIoControl completes the operation before returning, it returns TRUE, otherwise it returns FALSE. When the operation is finished, the system signals the manual reset event. You should call GetOverlappedResult when the thread that called DeviceIoControl needs to wait (that is, it wishes to stop executing) until the operation has finished.

Asynchronous operations are useful for lengthy operation, such as formatting a disk. In my packet dump program, I used asynchronous operation for both reading and writing data packets to the WinMIP driver. Just for fun, I also keep a count of how many packets received in the application are not in the same order as they are received in the driver, in variable *g_nSequenceErrorCount*.

8. Conclusions

8.1 Achieved Objectives

- Deeper understanding of Internet Protocol, particularly its encapsulation and routing mechanism. Through the project, I have realized how powerful, yet how simple the protocol IP is.
- Further understanding of and hands-on experience with the Mobile Internet Protocol.
- Thorough understanding of Windows NT network architecture and NDIS in particular.
- Practical knowledge of writing Windows NT kernel-mode driver

8.2 Unfulfilled Objectives

- Due to time constraints and some technical difficulties caused mainly by the limitation of NDIS architecture, none of the Mobile-IP nodes was fully implemented. Windows is still pretty much a closed system and programming in Windows OS level is very difficult despite tools, like DDK. NDIS does not have enough flexibility and programmer-friendly features to help a beginner like me to write a network driver quickly and easily.
- Due to time constraints and the fact that the none of the nodes was fully implemented, no testing was done to measure the performance of Mobile-IP in the Windows NT environment and to compare it with those of other environments, such as SunOS or Solaris. However, my qualified guess is that this implementation should work more efficiently than Anders Klemet's implementation for SunOS, when it comes to processing and routing of packets in the HA and the FA.The reason is simply that, in this implementation a dedicated protocol driver handles this solely, totally independent of other network protocols such as TCP/IP, thus eliminating the need for internal transmission of packets, thus reducing the number of times the packets must be copied.

8.3 Further Work

Due to the popularity of Linux (a PC version of Unix), many Mobile-IP implementations have been done using for Linux. Therefore, there is no imperative to implement Mobile IP on Windows. However, it would still be interesting to know how a Windows implementation would perform in comparison to one or more of these Linux implementation -- as you could run the systems on the same hardware platform. The possibilities of using a different network driver design for the Windows version could also be explored.

9. Development of Mobile IP Implementations

Almost 10 years have passed since I finished the main part of this thesis work. During the time, there have been many implementations of Mobile IP on the Microsoft Windows platform. As I am submitting the thesis now, I will give an overview of these implementations, highlight some key issues and proposed solutions. However, due to limited access to implementation information, particularly commercial implementation, the overview is far from comprehensive or conclusive. In Section 9.1, I will review the development of MIP in general. The structure of this section is adapted from a report by Chakchai So-In[35] as it provides a quite thorough overview of the area. I have selected a subset of the topics that this other report presented and will focus here on those that are most relevant to this project. Thus while the structure is similar, the sections included here reference additional research on each topic. In Section 9.2, I will focus on implementations on Microsoft Windows platform in particular.

9.1 Development of Mobile IP

Since the introduction of Mobile IP well over a decade ago, many developments have occurred concerning the implementation of this protocol, both from the research side and the commercial side. Abdul Sakib Mondal[19] described some of the current Mobile IP implementations and compared their features (as shown in Table 1).

TABLE 1. Comparison of several Mobile-IP implementations (based on [19])

Feature	Dynamic	MosquitoNet	Solaris MIP	Cellular IP	IMHP
Protocol Compability	High	High	Medium	Medium	Medium
Dependency on Network Support	High	Low	Medium	High	High
Support for Optimal Routing	Average	High	Above Average	Low	Above Average
Support for Security	High	Low	Medium	Low	High
Scalability	Highest	Above Average	Average	Lowest	Average
Speed of Handoff	Fast	Average	Average	Above Average	Slow
Overheads	High	Low	Below Average	High	Average

On the commercial side, most networking companies (such as Cisco, Nokia, Siemens, Hewlett-Packard, etc.) support Mobile IP. A number of vendors (such as BirdStep, Secgo, and ipUnplugged) provide Mobile IP clients. Despite all these successful commercializations of Mobile IP, as of today, we still do not see an adoption of Mobile IP on a large scale in the market. This probably has to attributed to the problems that Mobile IP is still facing, which I will explain more in detail in the following sections.

Overall Mobile IP is supports Macro Mobility well, when a MH roams between servicing wide-area wireless data systems (i.e., Inter-Domain mobility). However, when it comes to Micro Mobility support, when a MH moves from one base station's coverage area to another within the same network (i.e., Intra-Domain mobility), MIP has several limitations. Some of the key issues that have been addressed to overcome these limitations over the last decade include:

1. Smooth Handoff

When a MH roams from one local network to another, a smooth handoff between its old and new access point (i.e., with minimal packet loss and minimal increased latency) is critical to some applications running over Mobile IP, specifically those with tight real-time requirements such as VoIP. Thus while MIP provides a good solution for IP mobility at a global level, this may not be sufficient for mobility at the micro level. If a MH moves very frequently from one access point to another within a relatively small area, this will result in a lot of traffic to update the HA & Correspondent Node (CN) with the change of Care Of Address (COA), as the MH needs to notify both the HA and CN on each handoff. Since packets destined for the mobile node are not delivered until registration is completed at the HA, this interruption may cause a degradation in quality especially when real-time applications such as audio and video are used. The higher the link speed is, the more packets that might be lost during handoff - although this *may or may not* have an effect on the receiving application and the perceived quality of the communication. Further, when TCP is used for data downloading, changing the point of network attachment may degrade the throughput of the TCP connection due to TCP retransmission timeouts (i.e., the TCP connection may appear to be suffering from congestion - when in reality it is simply missing segments because the device did not have network connectivity for some period of time).

- Packet Loss

To avoid packet loss, the Foreign Agent Smooth Handoff proposal described by Perkins and Johnson in [8] provides a means for the MN's old foreign agent (oFA) to be notified of the MN's new mobility binding, allowing packets destined to the MN's oFA to be forwarded directly to its new COA. In this approach, the oFA allows the MN to execute a handoff **before** the new registration process is completed. During the handoff, the oFA maintains the binding for the former MH. After the oFA has received a binding update from the new FA (nFA) regarding the former MH, the OFA will tunnel all packets destined to the MH via the nFA (which will deliver them to the MH).

However, packets arriving at oFA *after* the MH has left **and** before the binding update is received from the nFA is received would be lost. To avoid this, each FA could utilize a circular buffer to store these packets. When a tunneled packet arrives at the oFA, it is decapsulated, and forwarded to the MH; additionally a copy is placed in the buffer. When the oFA receives a binding update these buffered packets are re-tunneled to the new FA. In order to avoid duplicate packets, the MH includes both the source address and datagram identification of the most recently received packets in the registration request that is sent to the oFA. The oFA can use this information to forward only the buffered packets that have **not** been received by the MH. There still remain (at least) two sources of packet loss in this solution: (a) Packets might be dropped by the oFA due to the limited size of the buffer; and (b) Packets that are tunneled from the oFA to nFA that arrive *before* the registration reply will be dropped.

- Handoff latency

There have been many proposals to reduce the handoff latency of MIP. I classify them into two categories: (a) aiming to reduce the network registration time by using a hierarchical network management structure; (b) attempting to reduce the address resolution time through address pre-configuration. Methods in category (a) are usually called *hierarchical handoff*, while methods in category (b) are referred to as *fast-handoff*.

Hierarchical handoff was introduced to hide local movements of MHs from HAs. Rather than provide the actual address of the MH, the address of a Gateway Foreign Agent (GFA) is sent to the HA. This address represents the address of a gateway - local to the visited network. This gateway is shared by a number of network access points. When a MN moves from one access point to another that is reachable through the same gateway, then the HA does not need to be informed, thus minimizing the signaling and routing traffic that needs to be sent through the network.

Many researcher have proposed fast-handoff techniques. Koodli and Perkins [9] and Koodli [10] proposed a Fast Hand Over technique to reduce handover delay. In their proposal the MH is allowed to send packets as soon as it detects a new network. Thus as soon as an FA detects a new MH, it would start delivering packets to it.

Another interesting proposal is from a group of students from the University of California at Berkeley.[11] They propose to achieve fast handoffs by using both the hierarchical structure of the network and location information provided by **GPS** (Global Positioning System) receiver at each router. Each network domain has a hierarchical tree of foreign agents with a GFA at the top of this tree, thus to the external world the GFA looks like the single FA for this domain. Using a GPS receiver, each router learns its own position. An advertisement scheme informs the other routers in the domain of this router's position. Routers use this position information to send the packets directed to both the (active) foreign agent and to adjacent foreign agents as well. In addition, the position information of GFAs are consulted to determine whether to send a registration request to the MH's home agent or to the MH's previous GFA. Note that the GFA will act as a HA, if a MH is geographically closer to the domain than to its own HA. While this approach may sound promising, geographical closeness may not necessary mean close in the network topology. Additionally, this scheme has some extra cost of implementation due to both the need for GPS receivers and to the extra messaging involved. Note that similar schemes based upon knowing or computing a set of adjacent access points have been proposed by others, for example Liu & Maguire[36] and neighbor graphs [37,38,39].

2. Mobile IP v6

IPv6 provides several additional features that enhance and simplify mobility support. Similar to MIP v4, two addresses are maintained (a Home address and COA), and a HA is used. However, FAs are not required, since Neighbor Discovery[15] and auto configuration of IP addresses (called Stateless Address Auto Configuration [14]) are mandatory parts of IPv6, a mobile node can obtain a local care-of address on its own (a co-located care of address (CCOA)). Consequently all routers must implement router advertisements. The MH still informs the HA of its current location. However, triangle routing no longer occurs and IP tunneling is no longer required as IPv6 source routing headers can be used. Additionally, the MH can inform any IPv6 CN of its new location and a route to reach this location. By using binding updates, binding acknowledgements, and the Return Routability mechanisms integrated in the MIP v6 packet, the MH can communicate directly with the CN. However, if a CN is not Mobile IPv6-capable, then packets sent between the CN and a MH that is away from its home network will be forwarded via the HA. Unfortunately, while many of the problems of Mobile IP are well address by IPv6 and Mobile IP v6, there are few internet service providers offering IPv6.

3. Multicast

IP multicast is an efficient way to send packets from one host to multiple hosts. However, traditional multicasting techniques have problems efficiently providing multicast to mobile hosts when faced with frequent membership or location changes. This is somewhat odd when one considers that a multicast address is not a network interface address, thus it would seem that it should support mobility.

IP multicasting is the transmission of an IP packet to interfaces that have joined a multicast. RFC 1112 [40] sets forth the recommended standard for IP multicasting on the Internet. In this scheme IP multicast packets are handled by “multicast routers” which may be co-resident with, or separate from, the typical unicast router. IP multicasting requires implementation of the Internet Group Management Protocol (IGMP), which a node uses to join a multicast group in order to receive packets addressed to a given multicast IP address.

There are two main Mobile IP multicast techniques: via a local multicast router and via bi-directional tunneling. In the first approach, the MH sends an IGMP message directly to a multicast router in its foreign network in order to join a specific multicast group, this IGMP request triggers multicast routing based on the interface which the request was received on - rather than an IP source address. This technique requires a multicast router in each visiting network and it requires a CCOA for the MH. When a CCOA can not be obtained, then MH origination multicast is not going to work. The reason is that multicast routers check if the upstream multicast packets arrive at the same interface as they should be sent back to the source. If not, the packets will be discarded. In the absence of CCOA, MH sends multicast packets using its home address as source address, whose interface is different from the link the MH is using. Hence the packets will be discarded by the router.

The second technique requires that the MH sends multicast packets through the unicast tunnel between the itself and its HA. Therefore, whenever the MH moves, there is no need to reconstruct the multicast distribution tree, but this may result in: (a) nonoptimal routing path (e.g., triangle routing), (b) tunnel convergence as the HA has to duplicate multicast packets for transmission to *each* mobile node that it is responsible for and send each of these packets via a tunnel to the respective MH’s FA. This replication of multicast packets by the HA is required, even if all of the mobile nodes are associated with a single HA and are visiting the same FA.

There are several alternatives to address these problems: Harrison, et al.[17] proposed a Mobile Multicast scheme, that establishes a Designated Multicast Service Provider (DMSP), most of the time this is the MH’s HA, which forwards multicast packets to its MHs, thus substantially reducing the required network traffic (as these packets can be forward as multicast packets).

Jiwoong Lee [30] proposed a Small Group Multicast (SGM) routing scheme. His basic idea is to include a list of destination addresses in a new SGM header and to avoid the use of the multicast routing protocols and multicast routing by the routers. SGM complements the existing multicast schemes in that it supports a very large number of small multicast groups, thus making multicast practical for new classes of applications such as IP telephony, various conferencing & collaborative applications, multiparty networked games, "replication" of Internet content, etc.

Jiang Wu and Gerald Q. Maguire Jr. proposed another scheme in which the MH sends messages to a multicast assistance agent in the network into which it believes that it will roam[29]. This agent joins the multicast tree on behalf of the MH. Thus

when the MH actually arrives in the network, much of the work to set up the multicast tree has hopefully already be done.

4. Voice over IP

VoIP is a typical real-time application that requires low handoff latency and low packet loss - even when users move from one network attachment point to another. Currently, there are two basic approaches to support macro mobility (i.e., inter-domain mobility) for VoIP services. MIP and related proposals seek to support mobility at the network layer while using existing VoIP protocols such as H.323 and the Session Initiation Protocol (SIP) seek to solve the mobility problem at the application layer. Note that when handling mobility at the application layer, the signaling might not only have to go between the communicating parties, but may also need to propagate between the various SIP proxies which were involved in setting up the call (as these may be *stateful*).

As mentioned earlier, the biggest problem in MIP for such an application is triangular routing which adds potentially considerable delay. Tunneling also adds additional overhead and consumes additional bandwidth.

Although SIP was not originally designed for node mobility, there has been ongoing research effort to develop SIP based mobility support. Wedlund and Schulzrinne have proposed [21] that when a MH moves from one cell to another during an active session, it obtains a new IP address from a Dynamic Host Configuration Protocol (DHCP) server or equivalent in the new network and sends a new session invitation to CN (i.e., a re-INVITE). The purpose of this new invitation is to update the CN with the MH's current IP address (so that the session can continue). Although this SIP-based solution offers some advantages over MIP, by eliminating triangular routing and tunneling overhead, it also has a number of drawbacks; the most significant is that a MH using SIP always needs to acquire a new IP address via DHCP which can cause a handoff delay of more than 2 seconds (however, there are techniques - such as proposed by Vatn and Maguire to reduce this delay[32]).The lack of an IP address and the resulting lack of the ability to send or receive packets can substantially disruption the communication, unless the new session is created while the MH is in an overlapping coverage area (i.e., in an area covered by both the old and new FA). Unfortunately, unless the MH has multiple receivers it is generally only able to communicate via a single interface (usually with a single unicast IP address) at any given time.

Jin-Woo Jung et al.[33] suggested a new mechanism that combines the best of Mobile IP and SIP based mobility. His scheme uses a SIP network server to handle call/session delivery **and** a Mobility Agent with SIP Registrar for handling location registration, location updates, and location queries. This mobility agent maintains two kinds of binding: one is the normal MIP "Permanent IP-COA" binding and the other is a SIP "User ID-Current IP binding". As each MH acquires both a COA and new IP address when roaming, thus the HAs do not need to tunnel data packets after completing a handoff for ongoing VoIP calls. This means that there is no session interruption during handoff (due to MIP) and no triangular routing during the active session (due to SIP).

5. Firewall Traversal

MIP assumes that MHs and FAs are uniquely identifiable by a globally routable IP address. Unfortunately, this assumption is not true when a mobile node attempts to communicate from behind a firewall. MIP relies on sending traffic from the home network to the MH or FA using IP-in-IP tunnelling, however, IP nodes behind a firewall are reachable only through the Network Address Translation (NAT)

device's public address(es). Unfortunately, IP-in-IP tunnelling does not generally contain enough information to allow a unique translation from a public address to the address of the COA of a MH or the address of a FA which resides behind the NAT. For this reason, IP-in-IP tunnels can not generally pass through a NAT, hence Mobile IP will not work across a NAT.

Several authors have proposed new data tunneling mechanisms to ensure a unique mapping of an internal address to an external address by the NAT; hence enabling MIP to traverse such a NAT. H. Levkowetz et al. [31] proposed a MIP UDP tunnelling mechanism which may be used to achieve NAT traversal. In MIP UDP tunnelling, the mobile node uses an extension in its Registration Request to indicate that it is able to use Mobile IP UDP tunnelling, rather than one of the standard Mobile IP tunnelling mechanisms. Note that this assumes that the registration request can pass through a NAT. Subsequently, the HA may then send a registration reply with an extension indicating acceptance (or denial). After receiving a reply from the HA, MIP UDP tunnelling will be used for both forward and reverse tunnelling (i.e., to and from the MH from the HA). UDP tunnelled packets sent by the mobile node use the same ports as the registration request message, which usually is UDP port 434. While, the source UDP port may vary between new registrations, it remains constant for all tunnelled packets and re-registrations. Note that UDP tunnelled packets *sent by* the HA uses the same pair of ports, but in the reverse order (i.e., what was the source port is used as the destination port and visa versa).

Even if the problem of NAT traversal is solved by using UDP tunneling, another problem remains. Usually a firewall does not allow packets whose source address is different from its own internal network address range to exit the firewall (this is called Ingress Filtering[24] - as addresses from another network are not topologically from "within" the firewall). To solve this problem, Montenegro recommends using Reverse Tunneling, rather than sending the packets directly to the CN, i.e., the MH tunnels all of the packets back to its HA, which then forwards these packets to the CN[25]. However, this results in non-optimal routing and increased signaling overhead.

6. Quality of Service (QoS)

Aside from the major problem of disruption during a Mobile IP handoff, the Quality of Service (QoS) of on-going communication is the next most significant issue, this is especially true for traffic which should receive a guaranteed traffic flow. The technologies to address this requirement (that have drawn the most attention) are: Integrated Services (IntServ), Differentiated Service (DiffServ), and Multiprotocol Label Switching (MPLS). IntServ uses a per-flow approach to provide guarantees to individual streams (based upon reserving resources in all of the routers along the path), while DiffServ provides aggregate assurances for a group of applications providing a forwarding equivalence class. While MPLS simulates circuit switching over an IP network. Instead of routing, MPLS does label based switching (for a good summery of MPLS and details of a generalization of MPLS see [41]).

However, all of the above mentioned techniques were designed for fixed networks, in order to adapt them to MIP, some modifications are needed. Zhang and Long [34] identified several major problems, challenges, and requirements in providing QoS-enabled mobile applications and their corresponding potential solutions. Additionally, the QoS survey regarding Mobile IP by Taha, et al. [28] is very precise and covers almost all the relevant QoS topics.

One of the topics in both of the above studies that is relevant here is the interruption in service during a handoff. This interruption in service and the time required to establish a new path may cause an abrupt change in the QoS experienced by packets arriving at an intermediate node in the new path. Additionally, this node may not yet know about the desired (previous) QoS requirements. Therefore, this node can only forward these packets using its default QoS. For example, after encapsulating a voice packet it might be marked (using DiffServ) as high priority, but the intermediate routers might modify this packet's priority level since thy do not (yet) know that this packet should be treated as a high priority packet nor that this user is allowed to send such priority packets. Raab and Chandra propose using a tunnel template to solve this problem [27]. Moreover, if the current QoS mechanism uses the source or destination IP address to determine the forwarding class, then some signaling may also be needed to notify the nodes along the unchanged part of the IP path that despite a MH's COA being changed that the new packets should receive the same treatment as the earlier packets. Examples of this approach are filter specs in IntServ nodes or packet classifiers at the edges of DiffServ networks.

9.2 Implementations on Windows

One of the more recent implementations that I studied was done by three Nokia engineers on Microsoft's Windows 2000 in year 2000. Haverinen et al.[18] adopted a similar approach as I discussed in section Section 4.1 by utilizing an intermediate driver between the TCP/IP stack and the device drivers. Even though they have only implemented MH functionality (which is much less complicated than either a HA or FA) in the driver, their approach still provides some useful insights. Specifically:

1. Although there are programming interfaces for manipulating the forwarding table and the ARP functionality in Microsoft's Windows 2000 system, there are still several things that are not supported, thus they have to be implemented in a kernel space driver. These include disabling the sending of ARP requests and an operation to control the reception of ICMP redirect messages.
2. For a MH, the forwarding table routines perform too strict sanity checks, thus an MH would not be allowed to use a host that resides on another IP subnetwork as its default gateway. While a sanity check is reasonable on fixed TCP/IP networks, in the case of Mobile IP the default router of a MH may very likely be a FA whose IP address does not belong to the same subnetwork as the MH. Therefore, there is a need to set the default router to some addresses that belongs to the home network and to create a false ARP table entry for this address for use with the link-layer address of packets to be sent to the HA.
3. Another practical problem was the time required to perform fowarding table updates. Unfortunately, it sometimes takes several seconds before changes in the fowarding table take effect. Since a MH needs to create temporary forwarding table entries for the mobility agents it is registering with, the handover performance due to this high delay will be very poor. As a workaround for this problem, this implementation sends the Mobile IP registration messages directly through the intermediate driver, without involving the TCP/IP stack.

10. Acknowledgments

The project was largely based on Anders Klemet's implementation of Mobile IP on SunOS. Special thanks to Prof. Dr. Gerald Q. Maguire Jr., for his encouragement, patience, and help. Also thanks to George Liu from Ericsson Radio System for giving me this opportunity to do this project. Good ideas have also been taken from previous thesis work on Mobile IP by Fredrik Broman and Fredrik Tarberg. The long delay in completing this work is solely my own fault.

11. Bibliography

[1] Charles E. Perkins (Editor), “IP Mobility Support”, Request for Comments 2002, Internet Engineering Task Force, October 1996. http://www.ietf.org/rfc/rfc2002.txt

[2] Charles E. Perkins, "IP Encapsulation within IP", Request for Comments 2003, Internet Engineering Task Force, October 1996. http://www.ietf.org/rfc/rfc2003.txt

[3] Charles E. Perkins, “Minimal Encapsulation within IP”, Request for Comments (Proposed Standard) 2004, Internet Engineering Task Force, October 1996. http://www.ietf.org/rfc/rfc2004.txt

[4] J. Solomon, “Applicability Statement for IP Mobility Support”, Request for Comments 2005, Internet Engineering Task Force, October 1996. http://www.ietf.org/rfc/rfc2005.txt

[5] D. Cong, M. Hamlen, and C. Perkins (Editors), “The Definitions of Managed Objects for IP Mobility”, Request for Comments 2006, Internet Engineering Task Force, October 1996. http://www.ietf.org/rfc/rfc2006.txt

[6] Microsoft, Windows Driver Development Kit (DDK), ~1996

[7] W. Richard Stevens, *TCP/IP Illustrated*, Volume 1 , Addison-Wesley Publishing Company, Inc., 1994

[8] C. Perkins and D. Johnson, Route Optimization in Mobile IP, IETF Internet Draft, draft-ietf-mobileip-optim-11.txt

[9] Rajeev Koodli and Charles E. Perkins, "Mobile IPv4 Fast Handovers," Request for Comments 4988, Internet Engineering Task Force, October 2007. http://www.ietf.org/rfc/rfc4988.txt

[10] Rajeev Koodli (Ed.), "Fast Handovers for Mobile IPv6," Request for Comments: 4068, Internet Engineering Task Force, July 2005. http://www.ietf.org/rfc/rfc4068.txt

[11] Mustafa Ergen, Sinem Coleri, Baris Dundar, Rahul Jain, Anuj Puri, Pravin Varaiya, "Application of GPS to Mobile IP and Routing in Wireless Networks," IEEE VTC, Vancouver, Canada, September, 2002.

[12] E. Gustafsson, A. Jonsson, and C. Perkins, "Mobile IPv4 Regional Registration," Internet Draft (work in progress), Internet Engineering Task Force, draft-ietf-mobileip-reg-tunnel-09.txt, 25 June 2004.

[13] D. Johnson, C. Perkins, and J. Arkko, "Mobility Support in IPv6," Request for Comments (Proposed Standard) 3775, Internet Engineering Task Force, June 2004. http://www.ietf.org/rfc/rfc3775.txt

[14] S. Thomson and T. Narten, “IPv6 Stateless Address Autoconfiguration”, Request for Comments 2462, Internet Engineering Task Force, December 1998. http://www.ietf.org/rfc/rfc2462.txt

[15] T. Narten, E. Nordmark, and W. Simpson, “Neighbor Discovery for IP Version 6 (IPv6)”, Request for Comments 2461, Internet Engineering Task Force, December 1998. http://www.ietf.org/rfc/rfc2461.txt

[16] G. Montenegro, “Bi-directional Tunneling for Mobile IP”, Internet Draft, Internet Engineering Task Force, draft-montenegro-tunneling-01.txt, September 1996. http://www.ietf.org/internet-drafts/draft-montenegro-tunneling-01.txt.

[17] Tim G. Harrison, Carey L. Williamson, Wayne L. Mackrell, and Richard B. Bunt, “Mobile Multicast (MoM) Protocol: Multicast Support for Mobile Hosts”, Proceedings of ACM/IEEE MOBICOM’97, September 1997, pages: 151-160.

[18] Henry Haverinen, Antti Kuikka, and Tuomas Maattanen, “A portable mobile IP implementation”, Proceeding of the 25th Annual IEEE Conference on Local Computer Network, November 2000.

[19] Abdul Sakib Mondal, *Mobile IP: Present State and Future*, Plenum Publishing Corporation, January 2003, ISBN-13: 9780306478741, 292 pages.

[20] Soonuk Seol, Myungchul Kim, Chansu Yu, and Jong-Hyun Lee, "Experiments and analysis of voice over Mobile IP," Personal, Indoor and Mobile Radio Communications, 2002. The 13th IEEE International Symposium, Vol. 2, 15-18 September 2002 Pages: 977 - 981.

[21] Elin Wedlund and Henning Schulzrinne, "Mobility support using SIP," In Proceedings of the 2nd ACM international workshop on Wireless mobile multimedia, 1999, Pages: 76 -82.

[22] Hanane Fathi, Shyam Chakraborty, and Ramjee Prasad, "Mobility management for VOIP: Evaluation of Mobile IP-based protocols," Communications, 2005. (ICC 2005) 2005 IEEE International Conference, Vol. 5, May 2005 Pages: 3230 - 3235.

[23] Min Wang and Geng-Sheng (G.S.) Kuo, "Enhancement of voice over mobile IP for infrastructure-mode wireless LANs," Global Telecommunications Conference, 2005. GLOBECOM’05. IEEE Vol. 5, December 2005 Pages: 2642 - 2646.

[24] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," Request for Comments (Proposed Standard) 2827, Internet Engineering Task Force, May 2000.

[25] G. Montenegro, "Reverse Tunneling for Mobile IP, revised," Request for Comments (Proposed Standard) 3024, Internet Engineering Task Force, January 2001.

[26] Charles E. Perkins and David B. Johnson, "Route Optimization in Mobile IP," Internet Draft, Internet Engineering Task Force, draft-ietf-mobileip-optim-11.txt, September 2001.

[27] Stefan Raab and Madhavi W. Chandra, "Mobile IP Technology and Applications," Cisco Press, 2005.

[28] Abd-Elhamid M. Taha, Hossam S. Hassanein, and Mouftah T. Mouftah, "Extensions for Internet QoS paradigms to mobile IP: a survey," Communications Magazine, IEEE Volume 43, Issue 5, May 2005 Pages: 132 - 139

[29] Jiang Wu and Gerald Q. Maguire Jr., Agent Based Seamless IP Multicast Receiver Handover, IFIP Conference on Personal Wireless Communications (PWC’2000), Gdansk, Poland, 14-15 September 2000.

[30] Jiwoong Lee, SGM Support in Mobile IP, Internet draft, IETF, draft-lee-sgm-support-mobileip-00.txt, October 2000

[31] H. Levkowetz and S. Vaarala, "Mobile IP Traversal of Network Address Translation (NAT) Devices", Request for Comments 3519, Internet Engineering Task Force, April 2003 http://www.ietf.org/rfc/rfc3519.txt

[32] Jon-Olov Vatn and Gerald Q. Maquire Jr., “The effect of using co-located care-of addresses on macro handover delay”, in 14th Nordic Teletraffic Seminar, Aug. 1998.

[33] Jin-Woo Jung, Hyun-Kook Kahng , Ranganathan Mudumbai, and Doug Montgomery, “Performance Evaluation of Two Layered Mobility Management using Mobile IP and Session Initiation Protocol”, Global Telecommunications Conference, 2003. GLOBECOM ’03. IEEE, Volume: 3, 1-5 Dec. 2003, pages 1190- 1194 http://w3.antd.nist.gov/pubs/sip-mip-jwjung-globecom2003.pdf

[34] Runtong, Zhang and Long, Keping (2002) QoS issues in mobile IP: challenges, requirements and solutions. In: Proceedings of the International Conference on Computer Communications (ICCC 2002): Vol. 2, 11-14 Aug 2002, Mumbai, India, pages 802 - 812.

[35] Chakchai So-In, ”Mobile IP Survey”, A survey paper for the course CSE574S: Advanced Topics in Networking: Wireless and Mobile Networking, Department of Computer Science & Engineering ,Washington University in St. Louis, May, 2006. http://www.cse.wustl.edu/~jain/cse574-06/ftp/mobile_ip.pdf

[36] Juntong Liu and Gerald Q. Maguire Jr. “GMRM: An Efficient Routing Model for an Integrated Wireless Mobile Packet Switch Network”, The 3rd Workshop on Personal Wireless Communication (PWC98), Tokyo, Japan, 1998.

[37] Arunesh Mishra, Min-ho Shin, and William A. Arbaugh, “An Analysis of the Layer 2 Handoff costs in Wireless Local Area Networks”, ACM Computer Communications Review , vol. 33, no. 2, pp. 93 - 102, 2003.

[38] Min-ho Shin, Arunesh Mishra, and William A. Arbaugh, “An Efficient Handoff Scheme in IEEE 802.11 using Neighbor Graphs”, in Proceedings of the Second International Conference on Mobile Systems, Applications, and Services , 2004.

[39] Arunesh Mishra, Min-ho Shin, and William A. Arbaugh, “Context Caching using Neighbor Graphs for Fast Handoffs in a Wireless Network”, in Proceedings of the 23rd Conference on Computer Communications (INFOCOM) , March 2004.

[40] S. Deering, Host Extensions for IP Multicasting, Request for Comments (RFC) 1112, Internet Engineering Task Force, August 1989. http://www.ietf.org/rfc/rfc1112.txt

[41] Pontus Sköldström, “Control and Integration of GMPLS networks”. Masters Thesis, Department of Communication Systems, School of Information and Communication Technology, Royal Institute of Technology, August 2008.

Appendix A A Modified Version of ‘Packet’ Sample

A.1 Rawupper.h

```
#ifndef __RAWUPPER_H__
#define __RAWUPPER_H__

#include“WinMIP.H”
#include“IOCTL.H”


//////////////////////////////////////////////////////////////////////
//// PROMISCUOUS PROTOCOL DRIVER UPPER-EDGE FUNCTION PROTOTYPES

VOID UPPERPacketReceived( PUSER_PACKET_DATA pRAWUserPacketData );

BOOL UPPEROnOpenHandle( void );
BOOL UPPERStartReception( PIRP pIrp );
DWORD UPPERPacketRead( PDEVICE_OBJECT pDeviceObject, PIRP pIrp );
VOID UPPERFreeUserPacketData( PUSER_PACKET_DATA pRAWUserPacketData );
BOOL UPPERStopReception( PIRP pIrp );
BOOL UPPEROnCloseHandle( void );
DWORD UPPERBufferSend( PIRP pIrp );

#endif // __RAWUPPER_H__
```

A.2 Rawupper.c

```
#include “ntddk.h”
#include “ntiologc.h”
#include “ndis.h”
#include<WinDef.h>
#include “debug.h”
#include “WinMIP.h”
#include“RAWUpper.h”
#include“RAWLower.h”

//////////////////////////////////////////////////////////////////////////
//// GLOBAL DATA

typedef
enum _UPPERState
{
     UPPERSTATE_IDLE = 0,
     UPPERSTATE_STARTED,
     UPPERSTATE_STOPPING,
     UPPERSTATE_STOPPED
}
     UPPERState, *PUPPERState;

DWORD     g_nQueuedPacketCount = 0;
UPPERState g_nUpperState = UPPERSTATE_IDLE;

//////////////////////////////////////////////////////////////////////////
//// PROTOCOL DRIVER UPPER-EDGE FUNCTION STUBS


//////////////////////////////////////////////////////////////////////////
//// UPPERSendCompleteHandler

VOID
UPPERSendCompleteHandler( DWORD nUPPERCallerData )
{
     PIRP                                        pIrp;
     PIO_STACK_LOCATIONpIrpSp;

     ASSERT( nUPPERCallerData );

     if( !nUPPERCallerData )
     {
          return;
     }

     pIrp = (PIRP )nUPPERCallerData;

     pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

     pIrp->IoStatus.Status = STATUS_SUCCESS;
     pIrp->IoStatus.Information = 0;// Nothing Returned

     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
}
```

```
////////////////////////////////////////////////////////////////////////////
//// UPPERBufferSend
// Remarks
// Modeled loosely after the packet sending mechanism used by the NT DDK
// PACKET protocol sample.
//

DWORD UPPERBufferSend( PIRP pIrp )
{
     PIO_STACK_LOCATIONpIrpSp;
     DWORD                                          nResult;

#if DBG
     DbgPrint( “UPPERBufferSend: Entry...\n” );
#endif

     /* Sanity Checks
     ---------------- */
     ASSERT( pIrp );

     if( !pIrp )
     {
          return( STATUS_UNSUCCESSFUL );
     }

     pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

     IoMarkIrpPending( pIrp );
     pIrp->IoStatus.Status = STATUS_PENDING;

     /* IO Method Comments
     ----------------------
      * Because this function is dispatched from DeviceIoControl, the method
      * used to transfer user data into the device is specified by the
      * method field of the IOCTL code - NOT by the flags specified when the
      * device was created.
      */

     nResult = LOWERBufferSend(
                                    (PBYTE )pIrp->AssociatedIrp.SystemBuffer,

pIrpSp->Parameters.DeviceIoControl.InputBufferLength,
                                    (DWORD )pIrp
                                    );

     return( nResult );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// UPPERPacketReceived
//

VOID
UPPERPacketReceived( PUSER_PACKET_DATA pRAWUserPacketData )
{
     PDEVICE_OBJECT       pDeviceObject;
     PDEVICE_EXTENSIONpDeviceExtension;
     BOOL                                          bResult;
     PIRP                                          pIrp;
     PIO_STACK_LOCATIONpIrpSp;
     KIRQL                                         OldIrql;
     PLIST_ENTRY                 pLinkage;

#if DBG
//   DbgPrint( “UPPERPacketReceived Entry...\n” );
#endif

     /* Sanity Check On Arguments
     --------------------------- */
     ASSERT( pRAWUserPacketData );

     if( !pRAWUserPacketData )
     {
          return;
     }

     ASSERT( g_pTheDriverObject );

     if( !g_pTheDriverObject )
     {
          return;
     }

     if( g_nUpperState != UPPERSTATE_STARTED )
     {
          /* Free Packet Here
          ------------------- */
          LOWERFreeUserPacketData( pRAWUserPacketData );

          return;
     }

     /* Report Packet Reception To Queued Callers
     ------------------------------------------ */
     pDeviceObject = g_pTheDriverObject->DeviceObject;

     while( pDeviceObject )
     {
          pDeviceExtension = pDeviceObject->DeviceExtension;

          KeAcquireSpinLock(
                  &pDeviceExtension->ReadListSpinLock,
                  &OldIrql
                  );

          if( !IsListEmpty( &pDeviceExtension->ReadList ) )
          {
                  pLinkage = RemoveHeadList(
```

```
                                                &pDeviceExtension->ReadList
                                                );

                        KeReleaseSpinLock(
                                &pDeviceExtension->ReadListSpinLock,
                                OldIrql
                                );

                        ASSERT( pLinkage );

                        if( pLinkage )
                        {
                                pIrp = CONTAINING_RECORD(
                                                                pLinkage,
                                                                IRP,

    Tail.Overlay.ListEntry
                                                                );

                                IoSetCancelRoutine( pIrp, NULL );

                                pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

                                NdisMoveMappedMemory(
                                        MmGetSystemAddressForMdl( pIrp->MdlAddress ),
                                        pRAWUserPacketData,
                                        sizeof( USER_PACKET_DATA )// ATTENTION!!! Could
    copy less!!!
                                        );

                                pIrp->IoStatus.Status = STATUS_SUCCESS;
                                pIrp->IoStatus.Information = sizeof( USER_PACKET_DATA );

                                IoCompleteRequest( pIrp, IO_NO_INCREMENT );
                        }
                }

                KeReleaseSpinLock(
                        &pDeviceExtension->ReadListSpinLock,
                        OldIrql
                        );

                pDeviceObject = pDeviceObject->NextDevice;
        }

        /* No User. Free Packet Here
        --------------------------- */
        LOWERFreeUserPacketData( pRAWUserPacketData );
    }
```

```
////////////////////////////////////////////////////////////////////////////
//// UPPEROnOpenHandle

BOOL UPPEROnOpenHandle( void )
{
     g_nQueuedPacketCount = 0;
     g_nUpperState = UPPERSTATE_IDLE;

     return( TRUE );
}


////////////////////////////////////////////////////////////////////////////
//// UPPERStartReception
//

BOOL UPPERStartReception( PIRP pIrp )
{
     PIO_STACK_LOCATIONpIrpSp;

     /* Sanity Check
     --------------- */
     ASSERT( pIrp );

     if( !pIrp )
     {
          return( STATUS_UNSUCCESSFUL );
     }

     pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

     g_nQueuedPacketCount = 0;
     g_nUpperState = UPPERSTATE_STARTED;

     pIrp->IoStatus.Status = STATUS_SUCCESS;
     IoCompleteRequest( pIrp, IO_NO_INCREMENT );

     return( STATUS_SUCCESS );
}


////////////////////////////////////////////////////////////////////////////
//// UPPERCancelPacketRead
// Remarks
// This routine is called with the CancelSpinLock held and is responsible
// for releasing it.
//

VOID UPPERCancelPacketRead( PDEVICE_OBJECT pDeviceObject, PIRP pIrp )
{
     PDEVICE_EXTENSIONpDeviceExtension;
     PIO_STACK_LOCATIONpIrpSp;
     KIRQL                                          OldIrql;

#if DBG
     DbgPrint( “UPPERCancelPacketRead: Entry...\n” );
#endif

     ASSERT( g_pTheDriverObject );
```

```
     if( !g_pTheDriverObject )
     {
          return;
     }

     pDeviceObject = g_pTheDriverObject->DeviceObject;

     pDeviceExtension = pDeviceObject->DeviceExtension;

     KeAcquireSpinLock(
          &pDeviceExtension->ReadListSpinLock,
          &OldIrql
          );

     if( !IsListEmpty( &pDeviceExtension->ReadList ) )
     {
          RemoveEntryList( &pIrp->Tail.Overlay.ListEntry );
     }

     KeReleaseSpinLock(
          &pDeviceExtension->ReadListSpinLock,
          OldIrql
          );

     IoSetCancelRoutine( pIrp, NULL );

     pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

     pIrp->IoStatus.Status = STATUS_CANCELLED;
     pIrp->IoStatus.Information = 0;

     IoReleaseCancelSpinLock( pIrp->CancelIrql );

     IoCompleteRequest( pIrp, IO_NO_INCREMENT );
}


////////////////////////////////////////////////////////////////////////
//// UPPERPacketRead
//

DWORD UPPERPacketRead( PDEVICE_OBJECT pDeviceObject, PIRP pIrp )
{
     PDEVICE_EXTENSION            pDeviceExtension;
     PIO_STACK_LOCATION   pIrpSp;
     DWORD                                                          nResult;
     KIRQL                                                          OldIrql;

#if DBG
//   DbgPrint( "UPPERPacketRead: Entry...\n" );
#endif

     /* Sanity Checks
     ---------------- */
     ASSERT( pDeviceObject );

     if( !pDeviceObject )
     {
          return( STATUS_UNSUCCESSFUL );
     }
```

```
     /* Sanity Checks
     ---------------- */
     ASSERT( pIrp );

     if( !pIrp )
     {
          return( STATUS_UNSUCCESSFUL );
     }

     pDeviceExtension = pDeviceObject->DeviceExtension;

     IoSetCancelRoutine( pIrp, UPPERCancelPacketRead );

     IoMarkIrpPending( pIrp );
     pIrp->IoStatus.Status = STATUS_PENDING;

     ExInterlockedInsertTailList(
          &pDeviceExtension->ReadList,
          &pIrp->Tail.Overlay.ListEntry,
          &pDeviceExtension->ReadListSpinLock
          );

     return( STATUS_PENDING );
}


VOID UPPERFreeUserPacketData( PUSER_PACKET_DATA pRAWUserPacketData )
{
     BOOL bResult;

     --g_nQueuedPacketCount;

     if( g_nUpperState == UPPERSTATE_STOPPING )
     {
#if DBG
          DbgPrint( "UPPERFreeUserPacketData: %d Packets Queued\n",
g_nQueuedPacketCount );
#endif

          if( g_nQueuedPacketCount == 0 )
          {
                  g_nUpperState = UPPERSTATE_STOPPED;

                  /* Wakeup Thread Blocked In UPPEROnStopReception
                  ---------------------------------------------- */
//                SignalID( (DWORD )&g_nQueuedPacketCount );
          }
     }

     LOWERFreeUserPacketData( pRAWUserPacketData );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// UPPERStopReception
//

BOOL UPPERStopReception( PIRP pIrp )
{
     PIO_STACK_LOCATIONpIrpSp;

#if DBG
     DbgPrint( “UPPEROnStopReception: %d Packets Queued\n”, g_nQueuedPacketCount
);
#endif

     /* Sanity Check
     --------------- */
     ASSERT( pIrp );

     if( !pIrp )
     {
          return( FALSE );
     }

     pIrpSp = IoGetCurrentIrpStackLocation( pIrp );

     g_nUpperState = UPPERSTATE_STOPPING;

     /* Block Until Outstanding Queued Packets Are Processed
     ------------------------------------------------------ */

     g_nUpperState = UPPERSTATE_STOPPED;

     pIrp->IoStatus.Status = STATUS_SUCCESS;
     IoCompleteRequest( pIrp, IO_NO_INCREMENT );

     return( TRUE );
}


//////////////////////////////////////////////////////////////////////////
//// UPPEROnCloseHandle
//

BOOL UPPEROnCloseHandle( void )
{
     return( TRUE );
}
```

A.3 Rawlower.h

```
#ifndef __RAWLOWER_H__
#define __RAWLOWER_H__

VOID
LOWEROpenAdapterCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status,
    IN NDIS_STATUS  OpenErrorStatus
    );

VOID
LOWERCloseAdapterCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status
    );

VOID
LOWERSendCompleteHandler(
    IN NDIS_HANDLE   ProtocolBindingContext,
    IN PNDIS_PACKET  pNdisPacket,
    IN NDIS_STATUS   Status
    );

VOID
LOWERTransferDataCompleteHandler(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_PACKET pNdisPacket,
    IN NDIS_STATUS Status,
    IN UINT BytesTransferred
    );

VOID
LOWERResetCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status
    );

VOID
LOWERRequestCompleteHandler(
    IN NDIS_HANDLE   ProtocolBindingContext,
    IN PNDIS_REQUEST pNdisRequest,
    IN NDIS_STATUS   Status
    );

NDIS_STATUS
LOWERReceiveHandler(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID HeaderBuffer,
    IN UINT HeaderBufferSize,
    IN PVOID LookAheadBuffer,
    IN UINT LookaheadBufferSize,
    IN UINT PacketSize
    );

VOID
LOWERReceiveCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext
    );
```

```
VOID
LOWERStatusHandler(
    IN NDIS_HANDLE   ProtocolBindingContext,
    IN NDIS_STATUS   GeneralStatus,
    IN PVOID         StatusBuffer,
    IN UINT          StatusBufferSize
    );

VOID
LOWERStatusCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext
    );

VOID
LOWERBindAdapterHandler(
    OUT PNDIS_STATUS    pStatus,
    IN  NDIS_HANDLE     hBindAdapterContext,
    IN  PNDIS_STRING    AdapterName,
    IN  PVOID           SystemSpecific1,
    IN  PVOID           SystemSpecific2
    );

VOID
LOWERUnbindAdapterHandler(
    OUT PNDIS_STATUS    pStatus,
    IN  NDIS_HANDLE     ProtocolBindingContext,
    IN  NDIS_HANDLE     hUnbindAdapterContext
    );

PLOWER_CONTEXT LOWERAllocContext( void );
VOID LOWERFreeContext( PLOWER_CONTEXT pLOWERContext );

VOID LOWERFreeUserPacketData( PUSER_PACKET_DATA pRAWUserPacketData );

BOOL LOWERQueryInfo( PLOWER_INFO pLOWERInfo );

DWORD LOWERBufferSend( PBYTE pBuffer, DWORD nLength, DWORD nUPPERCallerData );

#endif // __RAWLOWER_H__
```

```
//////////////////////////////////////////////////////////////////////////
//// INCLUDE FILES

#include “ntddk.h”
#include “ntiologc.h”
#include “ndis.h”
#include<WinDef.h>// ATTENTION!!! For NT def of DWORD, etc. Need better way...

#include “debug.h”
#include “WinMIP.h”
#include“RAWLOWER.h”
#include “PCAEnet.H”
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERFreePacketAndBuffers
//
// Purpose
// Free the NDIS buffers associated with a RAW_PACKET and then free
// the NDIS packet.
//
VOID
LOWERFreePacketAndBuffers( PRAW_PACKET pRAWPacket )
{
     ULONG                         nDataSize, nBufferCount;
     PNDIS_BUFFERpBuffer;

     /* Sanity Check On Arguments
     ---------------------------- */
     ASSERT( pRAWPacket );

     if( !pRAWPacket )
     {
          return;
     }

     /* Verify The Packet Signature
     ------------------------------ */
     ASSERT( pRAWPacket->Reserved.Signature == RAW_PACKET_SIGN );

     if( pRAWPacket->Reserved.Signature != RAW_PACKET_SIGN )
     {
          IF_LOUD(DbgPrint("LOWERFreePacketAndBuffers: Invalid Packet
Signature\n");)

          return;
     }

     NdisQueryPacket(
          (PNDIS_PACKET )pRAWPacket,
          (PULONG )NULL,
          (PULONG )&nBufferCount,
          &pBuffer,
          &nDataSize
          );

     /* Free All Of The Packet's Buffers
     ---------------------------------- */
     while( nBufferCount-- > 0L )
     {
          NdisUnchainBufferAtFront( (PNDIS_PACKET )pRAWPacket, &pBuffer );
          NdisFreeBuffer( pBuffer );
     }

     /* Recycle The Packet
     --------------------- */
     pRAWPacket->Reserved.Signature = 0;// Zap The Signature

     NdisReinitializePacket( (PNDIS_PACKET )pRAWPacket );

     /* And Free The Packet
     ---------------------- */
     NdisFreePacket( (PNDIS_PACKET )pRAWPacket );
}
```

```
////////////////////////////////////////////////////////////////////////
//// LOWERFreeUserPacketData
//

VOID
LOWERFreeUserPacketData( PUSER_PACKET_DATA pRAWUserPacketData )
{
     PRAW_PACKET pRAWPacket;

     /* Sanity Check On Arguments
     ---------------------------- */
     ASSERT( pRAWUserPacketData );

     if( !pRAWUserPacketData )
     {
          return;
     }

     pRAWPacket = CONTAINING_RECORD(
                                                  pRAWUserPacketData,
                                                  RAW_PACKET,
                                                  Reserved.UserPacketData
                                                  );

     /* Verify The Packet Signature
     ------------------------------ */
     ASSERT( pRAWPacket->Reserved.Signature == RAW_PACKET_SIGN );

     if( pRAWPacket->Reserved.Signature != RAW_PACKET_SIGN )
     {
          IF_LOUD(DbgPrint("LOWERFreePacketAndBuffers: Invalid Packet
Signature\n");)

          return;
     }

     LOWERFreePacketAndBuffers( pRAWPacket );
}


////////////////////////////////////////////////////////////////////////
//// LOWERQueryInfo
//
// Purpose
// To respond to a query requesting information about the adapter on the
// binding.
//

BOOL LOWERQueryInfo( PLOWER_INFO pLOWERInfo )
{
     PLOWER_CONTEXTpLOWERContext;
     ANSI_STRING  szansiAdapterName;

     /* Sanity Checks
     ---------------- */
     ASSERT( pLOWERInfo );

     if( !pLOWERInfo )
     {
          return( FALSE );
```

```
        }

        /* Return Information About The First Binding
        ------------------------------------------- */
        ASSERT( !IsListEmpty( &g_LowerBindingList ) );

        if( IsListEmpty( &g_LowerBindingList ) )
        {
             return( FALSE );       // No Information To Report...
        }

        pLOWERContext = (PLOWER_CONTEXT )g_LowerBindingList.Flink;

        /* Return Adapter Address
        ------------------------ */
        NdisMoveMemory(
             pLOWERInfo->szAdapterAddress,
             pLOWERContext->szAdapterAddress,
             ETHER_ADDR_LENGTH
             );

        /* Return Adapter Name
        --------------------- */
        szansiAdapterName.Buffer = pLOWERInfo->szAdapterName;
        szansiAdapterName.MaximumLength = MAX_ADAPTER_NAME;

        RtlUnicodeStringToAnsiString(
             &szansiAdapterName,
             (PNDIS_STRING )&pLOWERContext->szUniAdapterName,// Cheat!
             FALSE
             );

        szansiAdapterName.Buffer[ pLOWERContext->szUniAdapterName.Length ] = 0;

        /* Return The Medium
        ------------------- */
        pLOWERInfo->nSelectedMedium = (UINT )pLOWERContext->nSelectedMedium;

        /* Driver Options
        ---------------- */
        pLOWERInfo->bNoLoopback = pLOWERContext->bNoLoopback;

        /* Packet Statistics
        ------------------- */
        pLOWERInfo->nLastPacketNumber.LowPart =
    pLOWERContext->nLastPacketNumber.LowPart;
        pLOWERInfo->nLastPacketNumber.HighPart =
    pLOWERContext->nLastPacketNumber.HighPart;

        pLOWERInfo->nTossedPacketCount.LowPart =
    pLOWERContext->nTossedPacketCount.LowPart;
        pLOWERInfo->nTossedPacketCount.HighPart =
    pLOWERContext->nTossedPacketCount.HighPart;

        return( TRUE );
    }
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERBufferSend
//
// Purpose
// To send a buffer on the MAC binding.
//

DWORD LOWERBufferSend( PBYTE pBuffer, DWORD nLength, DWORD nUPPERCallerData )
{
     PLOWER_CONTEXTpLOWERContext;
     PRAW_PACKET  pRAWPacket;
     PNDIS_BUFFERpNdisBuffer;
     NDIS_STATUS  nNdisStatus;

     /* Sanity Checks
     ---------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          UPPERSendCompleteHandler( nUPPERCallerData );

          return( STATUS_INVALID_PARAMETER );
     }

     ASSERT( pBuffer );

     if( !pBuffer )
     {
          UPPERSendCompleteHandler( nUPPERCallerData );

          return( STATUS_INVALID_PARAMETER );
     }

     ASSERT( nLength <= MAX_ETHER_SIZE );

     if( nLength > MAX_ETHER_SIZE )
     {
          UPPERSendCompleteHandler( nUPPERCallerData );

          return( STATUS_INVALID_PARAMETER );
     }

     if( nLength == 0 )
     {
          UPPERSendCompleteHandler( nUPPERCallerData );

          return( STATUS_SUCCESS );// Do Nothing Gracefully...
     }

     /* Send On The First Binding
     --------------------------- */
     ASSERT( !IsListEmpty( &g_LowerBindingList ) );

     if( IsListEmpty( &g_LowerBindingList ) )
     {
          UPPERSendCompleteHandler( nUPPERCallerData );

          return( STATUS_UNEXPECTED_NETWORK_ERROR );// No Binding
     }
```

```
        pLOWERContext = (PLOWER_CONTEXT )g_LowerBindingList.Flink;

        /* Allocate The Transmit Packet Descriptor
        ---------------------------------------- */
        NdisAllocatePacket(
             &nNdisStatus,
             &(PNDIS_PACKET )pRAWPacket,
             pLOWERContext->hPacketPool
             );

        if( nNdisStatus != NDIS_STATUS_SUCCESS || !pRAWPacket )
        {
             // ATTENTION!!! Use separate counter for TX/RX tossed counts...
    //       RAWLargeIntegerIncrement( &pLOWERContext->nTossedPacketCount );

             IF_LOUD(DbgPrint("LOWERBufferSend Could Not Allocate Packet\n");)

             UPPERSendCompleteHandler( nUPPERCallerData );

             return( STATUS_NO_MEMORY );     // Must Drop It
        }

        /* Initialize The Packet Signature
        --------------------------------- */
        pRAWPacket->Reserved.Signature = RAW_PACKET_SIGN;

        /* Save Data To Be Returned To Caller On Completion
        ------------------------------------------------- */
        pRAWPacket->Reserved.nUPPERCallerData = nUPPERCallerData;

        /* Add Our Source Address To The Buffer
        ------------------------------------ */
        switch( pLOWERContext->nSelectedMedium )
        {
             case NdisMediumLocalTalk:
                    pBuffer[ 1 ] = pLOWERContext->szAdapterAddress[ 5 ];
                    break;

             case NdisMedium802_5:
             case NdisMedium802_3:
             default:
                    NdisMoveMemory(
                           &pBuffer[ MSrcAddr ],
                           pLOWERContext->szAdapterAddress,
              ETHER_ADDR_LENGTH
                           );
                    break;
        }

        /* Allocate An NDIS Buffer Descriptor For The Transmit Data Buffer
        ---------------------------------------------------------------- */
        NdisAllocateBuffer(
             &nNdisStatus,
             &pNdisBuffer,
             pLOWERContext->hBufferPool,
             pBuffer,
             nLength
             );
```

```
     if( nNdisStatus != NDIS_STATUS_SUCCESS || !pNdisBuffer )
     {
          // ATTENTION!!! Use separate counter for TX/RX tossed counts...
//        RAWLargeIntegerIncrement( &pLOWERContext->nTossedPacketCount );

          NdisFreePacket( (PNDIS_PACKET )pRAWPacket );

#if DBG
          DbgPrint( "LOWERBufferSend Could Not Allocate Buffer\n" );
#endif

          return( STATUS_NO_MEMORY );     // Must Drop It
     }

     /* Chain The Caller's Buffer Into The NDIS Packet
     ---------------------------------------------- */
     NdisChainBufferAtFront( (PNDIS_PACKET )pRAWPacket, pNdisBuffer );

     /* Send The Frame
     ---------------- */
     NdisSetSendFlags( (PNDIS_PACKET )pRAWPacket, 0 );

     NdisSend(
          &nNdisStatus,
          pLOWERContext->hNdisAdapterHandle,
          (PNDIS_PACKET )pRAWPacket
          );

     if( nNdisStatus != NDIS_STATUS_PENDING )
     {
          LOWERSendCompleteHandler(
                  (NDIS_HANDLE )pLOWERContext,
                  (PNDIS_PACKET )pRAWPacket,
                  nNdisStatus
                  );
     }

     return( STATUS_SUCCESS );
}


////////////////////////////////////////////////////////////////////////
//// LOWERAllocContext
//

PLOWER_CONTEXT LOWERAllocContext( void )
{
     PLOWER_CONTEXTpLOWERContext;
     NDIS_STATUS  nNdisStatus;
     int                         i;

     /* Allocate Memory For The Lower Binding Context
     ---------------------------------------------- */
     nNdisStatus = NdisAllocateMemory(
                                        &pLOWERContext,
                                        sizeof(LOWER_CONTEXT ),
                                        0,                              //
Allocate non-paged system-space memory
                                        HighestAcceptableMax
                                        );
```

```
ASSERT( nNdisStatus == NDIS_STATUS_SUCCESS );

if( nNdisStatus != NDIS_STATUS_SUCCESS )
{
     //
     // no memory
     //
     return( (PLOWER_CONTEXT )NULL );
}

NdisZeroMemory( pLOWERContext, sizeof(LOWER_CONTEXT) );

/* Allocate The Packet Pool
--------------------------- */
NdisAllocatePacketPool(
     &nNdisStatus,
     &pLOWERContext->hPacketPool,
     RAW_PACKET_POOL_SIZE,
     sizeof(RAW_PACKET_RESERVED)
     );

if (nNdisStatus != NDIS_STATUS_SUCCESS)
{
     IF_LOUD(DbgPrint("WinMIP: Failed to allocate packet pool\n");)

     LOWERFreeContext(pLOWERContext);

     return( (PLOWER_CONTEXT )NULL );
}

/* Allocate The Buffer Pool
--------------------------- */
NdisAllocateBufferPool(
     &nNdisStatus,
     &pLOWERContext->hBufferPool,
     RAW_BUFFER_POOL_SIZE
     );

if( nNdisStatus != NDIS_STATUS_SUCCESS )
{
     IF_LOUD(DbgPrint("WinMIP: Failed to allocate buffer pool\n");)

     NdisFreePacketPool( pLOWERContext->hPacketPool );

     LOWERFreeContext(pLOWERContext);

     return( (PLOWER_CONTEXT )NULL );
}

/* Initialize Packet Receive List
--------------------------------- */
NdisAllocateSpinLock(&pLOWERContext->ReceiveListSpinLock);
InitializeListHead( &pLOWERContext->PacketReceiveList );

/* Initialize Request List
-------------------------- */
NdisAllocateSpinLock(&pLOWERContext->RequestSpinLock);
NdisInitializeListHead(&pLOWERContext->RequestList);
```

```
	/* Link Up The Request Pool
	--------------------------- */
	for (i=0;i<MAX_REQUESTS;i++)
	{
		NdisInterlockedInsertTailList(
			&pLOWERContext->RequestList,
			&pLOWERContext->Requests[i].ListElement,
			&pLOWERContext->RequestSpinLock
			);
	}

	/* Default To No Loopback
	------------------------ */
//	pLOWERContext->bNoLoopback = TRUE;
	pLOWERContext->bNoLoopback = FALSE;

	pLOWERContext->bLowerContextInitDone = TRUE;

	return( pLOWERContext );
}


////////////////////////////////////////////////////////////////////////
//// LOWERFreeContext
//
// Purpose
//
// Parameters
//
// Return Value
//

VOID LOWERFreeContext( PLOWER_CONTEXT pLOWERContext )
{
	/* Free Spin Locks
	----------------- */
	NdisFreeSpinLock(&pLOWERContext->RequestSpinLock);
	NdisFreeSpinLock(&pLOWERContext->ReceiveListSpinLock);

	/* Free The NDIS Buffer And Packet Pools
	--------------------------------------- */
	if( pLOWERContext->hBufferPool )
	{
		NdisFreeBufferPool( pLOWERContext->hBufferPool );

		pLOWERContext->hBufferPool = NULL;
	}

	if( pLOWERContext->hPacketPool )
	{
		NdisFreePacketPool( pLOWERContext->hPacketPool );

		pLOWERContext->hPacketPool = NULL;
	}

	/* Free The Adapter Name Space
	----------------------------- */
	if( pLOWERContext->szUniAdapterName.Buffer )
	{
		NdisFreeMemory(
```

```
                    pLOWERContext->szUniAdapterName.Buffer,
                    pLOWERContext->szUniAdapterName.MaximumLength,
                    0
                    );
     }

     /* Finally, Free The Context Memory
     ---------------------------------- */
     NdisFreeMemory(
          pLOWERContext,
          sizeof(LOWER_CONTEXT ),
          0
          );
}


////////////////////////////////////////////////////////////////////////
//// NDIS 3.1 PROTOCOL LOWER-EDGE FUNCTION STUBS


////////////////////////////////////////////////////////////////////////
//// LOWERGetAdapterAddress
//

VOID LOWERGetAdapterAddress(
     PLOWER_CONTEXT pLOWERContext
     )
{
     PLIST_ENTRY          pRequestListEntry;
     PINTERNAL_REQUESTpRequest;

     /* Sanity Checks
     --------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     ASSERT( pLOWERContext );

     if( !pLOWERContext )
     {
          return;
     }

     /* Ask For Current Address
     ------------------------- */
     pLOWERContext->nAdapterAddressStatus = NDIS_STATUS_FAILURE;

     pRequestListEntry = NULL;

     if( !IsListEmpty( &pLOWERContext->RequestList ) )
     {
          pRequestListEntry = NdisInterlockedRemoveHeadList(

&pLOWERContext->RequestList,

&pLOWERContext->RequestSpinLock
```

```
                                                                        );
     }

     if( pRequestListEntry == NULL)
     {
          pLOWERContext->nAdapterAddressStatus = NDIS_STATUS_RESOURCES;

          IF_LOUD(DbgPrint("LOWERGetAdapterAddress: Could Not Allocate Request
Entry\n");)

          return;
     }

     pRequest = (PINTERNAL_REQUEST )pRequestListEntry;

     pRequest->Request.RequestType = NdisRequestQueryInformation;

     switch( pLOWERContext->nSelectedMedium )
     {
          case NdisMedium802_3:
                  pRequest->Request.DATA.QUERY_INFORMATION.Oid =
OID_802_3_CURRENT_ADDRESS;

                  pRequest->Request.DATA.QUERY_INFORMATION.InformationBuffer =
pLOWERContext->szAdapterAddress;

pRequest->Request.DATA.QUERY_INFORMATION.InformationBufferLength =
ETHER_ADDR_LENGTH;
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesWritten = 0;
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesNeeded =
ETHER_ADDR_LENGTH;
                  break;

          case NdisMedium802_5:
                  pRequest->Request.DATA.QUERY_INFORMATION.Oid =
OID_802_5_CURRENT_ADDRESS;

                  pRequest->Request.DATA.QUERY_INFORMATION.InformationBuffer =
pLOWERContext->szAdapterAddress;

pRequest->Request.DATA.QUERY_INFORMATION.InformationBufferLength =
ETHER_ADDR_LENGTH;
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesWritten = 0;
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesNeeded =
ETHER_ADDR_LENGTH;
                  break;

          case NdisMediumLocalTalk:
                  pRequest->Request.DATA.QUERY_INFORMATION.Oid =
OID_LTALK_CURRENT_NODE_ID;

                  pRequest->Request.DATA.QUERY_INFORMATION.InformationBuffer =
&pLOWERContext->szAdapterAddress[ 4 ];

pRequest->Request.DATA.QUERY_INFORMATION.InformationBufferLength = sizeof( USHORT
);
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesWritten = 0;
                  pRequest->Request.DATA.QUERY_INFORMATION.BytesNeeded = sizeof(
USHORT );
                  break;
```

```
            default:
                    ASSERT( FALSE );
                    return;
      }

      NdisRequest(
            &pLOWERContext->nAdapterAddressStatus,
            pLOWERContext->hNdisAdapterHandle,
            &pRequest->Request
            );

      if( pLOWERContext->nAdapterAddressStatus != NDIS_STATUS_PENDING )
      {
            IF_LOUD(DbgPrint("LOWERGetAdapterAddress: Calling
LOWERRequestCompleteHandler\n");)

            LOWERRequestCompleteHandler(
                    (NDIS_HANDLE )pLOWERContext,
                    &pRequest->Request,
                    pLOWERContext->nAdapterAddressStatus
                    );
      }
}


////////////////////////////////////////////////////////////////////////
//// LOWEROpenAdapterCompleteHandler
//

VOID LOWEROpenAdapterCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status,
    IN NDIS_STATUS  OpenErrorStatus
    )

{
      PLOWER_CONTEXT                      pLOWERContext;

      IF_LOUD(DbgPrint("WinMIP: OpenAdapterCompleteHandler\n");)

      /* Sanity Checks
      ---------------- */
      ASSERT( g_NdisProtocolHandle );

      if( !g_NdisProtocolHandle )
      {
            return;
      }

      pLOWERContext= (PLOWER_CONTEXT)ProtocolBindingContext;

      if (Status != NDIS_STATUS_SUCCESS)
      {
            IF_LOUD(DbgPrint("WinMIP: OpenAdapterComplete-FAILURE\n");)

#if defined(NDIS40) || defined(NDIS41)
            if( pLOWERContext->hBindAdapterContext )
            {
                    NdisCompleteBindAdapter(
```

```
                        pLOWERContext->m_hBindAdapterContext,
                        Status,                                    // Final
Status Of Bind Operation
                        OpenErrorStatus                    // Status From
NdisOpenAdapter
                        );
          }
#endif

          LOWERFreeContext( pLOWERContext );

          return;
     }

     /* Determine Selected Medium
     ---------------------------- */
     pLOWERContext->nSelectedMedium =
          pLOWERContext->AdapterMediumArray[ pLOWERContext->nSelectedMediumIndex
];

#if DBG
     if( pLOWERContext->nSelectedMediumIndex == 0 )
     {
          DbgPrint( “SelectedMedium: 802.3\n” );
     }
     else if( pLOWERContext->nSelectedMediumIndex == 1 )
     {
          DbgPrint( “SelectedMedium: 802.3\n” );
     }
     else if( pLOWERContext->nSelectedMediumIndex == 2 )
     {
          DbgPrint( “SelectedMedium: LocalTalk\n” );
     }
     else
     {
          DbgPrint( “SelectedMedium: %d Not Supported By WinMIP!\n”,
                  pLOWERContext->nSelectedMediumIndex );
     }
#endif // DBG

     /* Link The Binding Context Into The Binding List
     ---------------------------------------------- */
     InsertHeadList(
          (PLIST_ENTRY )&g_LowerBindingList,
          (PLIST_ENTRY )pLOWERContext
          );

     /* Indicate That The Adapter Has Been Opened
     ----------------------------------------- */
     pLOWERContext->bAdapterOpen = TRUE;

#if defined(NDIS40) || defined(NDIS41)
          if( pLOWERContext->hBindAdapterContext )
          {
                  NdisCompleteBindAdapter(
                          pLOWERContext->m_hBindAdapterContext,
                          NDIS_STATUS_SUCCESS,      // Final Status Of Bind
Operation
                          OpenErrorStatus                    // Status From
NdisOpenAdapter
```

```
                              );
          }
#endif

     /* Get The Adapter Datalink (Physical) Address
     ------------------------------------------- */
     LOWERGetAdapterAddress( pLOWERContext );

     return;
}


////////////////////////////////////////////////////////////////////////
//// LOWERCloseAdapterCompleteHandler
//

VOID LOWERCloseAdapterCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status
    )
{
     PLOWER_CONTEXT       pLOWERContext;

     IF_LOUD(DbgPrint("WinMIP: CloseAdapterCompleteHandler Entry...\n");)

     /* Sanity Checks
     ---------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     pLOWERContext = (PLOWER_CONTEXT)ProtocolBindingContext;

     pLOWERContext->NdisRequestStatus = Status;

     KeSetEvent( &pLOWERContext->NdisRequestEvent, 0L, FALSE );

     return;
}


////////////////////////////////////////////////////////////////////////
//// LOWERSendCompleteHandler
//

VOID LOWERSendCompleteHandler(
    IN NDIS_HANDLE   ProtocolBindingContext,
    IN PNDIS_PACKET  pNdisPacket,
    IN NDIS_STATUS   Status
    )
{
     PLOWER_CONTEXT pLOWERContext;
     PRAW_PACKET  pRAWPacket;
     DWORD                        nUPPERCallerData;

#if DBG
     DbgPrint( "SendCompleteHandler Entry..." );
```

```
#endif

     /* Sanity Checks On Arguments
     ---------------------------- */
     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return;
     }

     ASSERT( pNdisPacket );

     if( !pNdisPacket )
     {
          return;
     }

     /* Get Driver Context
     -------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     pRAWPacket = (PRAW_PACKET )pNdisPacket;

     /* Verify The Packet Signature
     ---------------------------- */
     ASSERT( pRAWPacket->Reserved.Signature == RAW_PACKET_SIGN );

     if( pRAWPacket->Reserved.Signature != RAW_PACKET_SIGN )
     {
          return;
     }

     nUPPERCallerData = pRAWPacket->Reserved.nUPPERCallerData;

     /* Free The Packet
     ----------------- */
     LOWERFreePacketAndBuffers( pRAWPacket );

     /* Call The Upper Edge Send Completion Routine
     ------------------------------------------- */
     UPPERSendCompleteHandler( nUPPERCallerData );
}


////////////////////////////////////////////////////////////////////////
//// LOWERTransferDataCompleteHandler
//

VOID LOWERTransferDataCompleteHandler (
    IN NDIS_HANDLE   ProtocolBindingContext,
    IN PNDIS_PACKET  pNdisPacket,
    IN NDIS_STATUS   Status,
    IN UINT          BytesTransfered
    )
{
     PLOWER_CONTEXT pLOWERContext;
     PRAW_PACKET  pRAWPacket;

#if DBG
```

```
//   DbgPrint( “WinMIP: TransferDataCompleteHandler; Status: 0x%X; Bytes
Transfered; %d\n”,
//        Status, BytesTransfered );
#endif

     /* Sanity Checks On Arguments
     ---------------------------- */
     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return;
     }

     ASSERT( pNdisPacket );

     if( !pNdisPacket )
     {
          return;
     }

     /* Get Driver Context
     -------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     pRAWPacket = (PRAW_PACKET )pNdisPacket;

     /* Verify The Packet Signature
     ---------------------------- */
     ASSERT( pRAWPacket->Reserved.Signature == RAW_PACKET_SIGN );

     if( pRAWPacket->Reserved.Signature != RAW_PACKET_SIGN )
     {
          return;
     }

     pRAWPacket->Reserved.UserPacketData.nPacketDataLength +=
          BytesTransfered;
}


////////////////////////////////////////////////////////////////////////
//// LOWERResetCompleteHandler
//

VOID LOWERResetCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext,
    IN NDIS_STATUS  Status
    )
{
    PLOWER_CONTEXT      pLOWERContext;
    PIRP                pIrp;

    PLIST_ENTRY         ResetListEntry;

    IF_LOUD(DbgPrint( “WinMIP: ResetCompleteHandler Entry...\n”);)

     /* Sanity Checks
     --------------- */
     ASSERT( g_NdisProtocolHandle );
```

```
     if( !g_NdisProtocolHandle )
     {
          return;
     }

    pLOWERContext= (PLOWER_CONTEXT )ProtocolBindingContext;


    //
    //  remove the reset IRP from the list
    //
    ResetListEntry=NdisInterlockedRemoveHeadList(
                      &pLOWERContext->ResetIrpList,
                      &pLOWERContext->RequestSpinLock
                      );

#if DBG
    if (ResetListEntry == NULL) {
        DbgBreakPoint();
        return;
    }
#endif

    pIrp=CONTAINING_RECORD(ResetListEntry,IRP,Tail.Overlay.ListEntry);

    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);

    IF_LOUD(DbgPrint("WinMIP: ResetCompleteHandler Normal Exit\n");)

    return;
}


////////////////////////////////////////////////////////////////////////
//// LOWERSetPacketFilter
//
VOID LOWERSetPacketFilter(
          PLOWER_CONTEXT pLOWERContext,
          ULONG nPacketFilter
          )
{
     PLIST_ENTRY          pRequestListEntry;
     PINTERNAL_REQUESTpRequest;

     /* Sanity Check On Arguments
     ---------------------------- */
     ASSERT( pLOWERContext );

     if( !pLOWERContext )
     {
          return;
     }

     /* Set The Packet Filter
     ------------------------ */
     pLOWERContext->nSetPacketFilterStatus = NDIS_STATUS_FAILURE;

     pLOWERContext->nPacketFilter = nPacketFilter;
```

```
     pRequestListEntry = NULL;

     if( !IsListEmpty( &pLOWERContext->RequestList ) )
     {
          pRequestListEntry = NdisInterlockedRemoveHeadList(

&pLOWERContext->RequestList,

&pLOWERContext->RequestSpinLock
                                                                        );
     }

     if( pRequestListEntry == NULL)
     {
          pLOWERContext->nSetPacketFilterStatus = NDIS_STATUS_RESOURCES;

#if DBG
          DbgPrint( “LOWERSetPacketFilter: Could Not Allocate Request Entry\n” );
#endif

          return;
     }

     pRequest = (PINTERNAL_REQUEST )pRequestListEntry;

     pRequest->Request.RequestType = NdisRequestSetInformation;
     pRequest->Request.DATA.SET_INFORMATION.Oid = OID_GEN_CURRENT_PACKET_FILTER;

     pRequest->Request.DATA.SET_INFORMATION.InformationBuffer =
&pLOWERContext->nPacketFilter;
     pRequest->Request.DATA.SET_INFORMATION.InformationBufferLength = sizeof(
ULONG );
     pRequest->Request.DATA.SET_INFORMATION.BytesRead = 0;
     pRequest->Request.DATA.SET_INFORMATION.BytesNeeded = 0;

     NdisRequest(
          &pLOWERContext->nSetPacketFilterStatus,
          pLOWERContext->hNdisAdapterHandle,
          &pRequest->Request
          );

     if( pLOWERContext->nSetPacketFilterStatus != NDIS_STATUS_PENDING )
     {
#if DBG
          DbgPrint( “LOWERSetPacketFilter: Calling LOWERRequestCompleteHandler\n”
);
#endif

          LOWERRequestCompleteHandler(
                  (NDIS_HANDLE )pLOWERContext,
                  &pRequest->Request,
                  pLOWERContext->nSetPacketFilterStatus
                  );
     }
}
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERRequestCompleteHandler
//

VOID LOWERRequestCompleteHandler(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN PNDIS_REQUEST pNdisRequest,
    IN NDIS_STATUS Status
    )
{
     PLOWER_CONTEXT       pLOWERContext;
     PINTERNAL_REQUESTpRequest;

#if DBG
     DbgPrint( “WinMIP: LOWERRequestCompleteHandler Status 0x%x\n”, Status );
#endif

     /* Sanity Checks
     ---------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return;
     }

     ASSERT( pNdisRequest );

     if( !pNdisRequest )
     {
          return;
     }

     /* Get Driver Context
     --------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     /* Get Driver Context
     --------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     pRequest = CONTAINING_RECORD( pNdisRequest, INTERNAL_REQUEST, Request );

     /* Handle Completion Based On RequestType
     ---------------------------------------- */
     switch( pRequest->Request.RequestType )
     {
          case NdisRequestSetInformation:
                  /* Handle Based On Object ID
                  -------------------------- */
                  switch( pRequest->Request.DATA.SET_INFORMATION.Oid )
                  {
                          case OID_GEN_CURRENT_PACKET_FILTER:
```

```
                                        if( Status == NDIS_STATUS_SUCCESS )
                                        {
                                                IF_LOUD(DbgPrint("SetPacketFilter
Succeeded\n");)
                                                pLOWERContext->bFilterSet = TRUE;
                                        }
                                        else
                                        {
                                                IF_LOUD(DbgPrint("SetPacketFilter
FAILED\n");)
                                                pLOWERContext->bFilterSet = FALSE;
                                        }
                                        break;

                                case OID_802_3_MULTICAST_LIST:
                                        if( Status == NDIS_STATUS_SUCCESS )
                                        {
                                                IF_LOUD(DbgPrint("SetMulticastAddress
Succeeded\n");)

#if DBG
                                                DbgPrint( "BytesRead: %d\n",

pRequest->Request.DATA.SET_INFORMATION.BytesRead );
#endif
                                        }
                                        else
                                        {
                                                IF_LOUD(DbgPrint("SetMulticastAddress
FAILED\n");)
                                        }
                                        break;

                                default:
                                        IF_LOUD(DbgPrint("LOWERRequestCompleteHandler
Unsupported OID\n");)

                                        break;
                        }
                        break;

                case NdisRequestQueryInformation:
                        /* Handle Based On Object ID
                        --------------------------- */
                        switch( pRequest->Request.DATA.QUERY_INFORMATION.Oid )
                        {
                                case OID_802_3_CURRENT_ADDRESS:
                                        if( Status == NDIS_STATUS_SUCCESS )
                                        {
                                                IF_LOUD(DbgPrint("Got Adapter 802.3
Current Address\n");)

                                                LOWERSetPacketFilter(
                                                          pLOWERContext,
                                                          NDIS_PACKET_TYPE_DIRECTED
                                                          |
NDIS_PACKET_TYPE_PROMISCUOUS
                                                          );
                                        }
                                        else
```

```
                                        {
                                                IF_LOUD(DbgPrint("Get Adapter 802.3
Address FAILED\n");)
                                        }
                                        break;

                                case OID_802_5_CURRENT_ADDRESS:
                                        if( Status == NDIS_STATUS_SUCCESS )
                                        {
                                                IF_LOUD(DbgPrint("Got Adapter 802.5
Current Address\n");)

                                                LOWERSetPacketFilter(
                                                        pLOWERContext,
                                                        NDIS_PACKET_TYPE_DIRECTED
                                                        |
NDIS_PACKET_TYPE_PROMISCUOUS
                                                        );
                                        }
                                        else
                                        {
                                                IF_LOUD(DbgPrint("Get Adapter 802.5
Address FAILED\n");)
                                        }
                                        break;

                                case OID_LTALK_CURRENT_NODE_ID:
                                        if( Status == NDIS_STATUS_SUCCESS )
                                        {
                                                IF_LOUD(DbgPrint("Got Adapter LocalTalk
Node Address\n");)

                                                /* Put Node Address In Least Significant
Byte

------------------------------------------- */
                                                pLOWERContext->szAdapterAddress[ 5 ] =
pLOWERContext->szAdapterAddress[ 4 ];
                                                pLOWERContext->szAdapterAddress[ 4 ] =
0;

                                                LOWERSetPacketFilter(
                                                        pLOWERContext,
                                                        NDIS_PACKET_TYPE_DIRECTED
                                                        | NDIS_PACKET_TYPE_BROADCAST
                                                        |
//
NDIS_PACKET_TYPE_PROMISCUOUS
                                                        );
                                        }
                                        else
                                        {
                                                IF_LOUD(DbgPrint("Get Adapter LocalTalk
Node Address FAILED\n");)
                                        }
                                        break;

                                default:
                                        IF_LOUD(DbgPrint("LOWERRequestCompleteHandler
Unsupported OID\n");)
```

```
                                        break;
                        }
                        break;

                default:
                        IF_LOUD(DbgPrint("LOWERRequestCompleteHandler Unsupported
RequestType\n");)
                        break;
        }

        /* Put The Request Parameter Block Back Into The Free Request List
        ------------------------------------------------------------------ */
        NdisInterlockedInsertTailList(
                &pLOWERContext->RequestList,
                &pRequest->ListElement,
                &pLOWERContext->RequestSpinLock
                );
}


VOID RAWLargeIntegerIncrement( PLARGE_INTEGER pLargeInteger )
{
        if( ++pLargeInteger->LowPart == 0 )
        {
                ++pLargeInteger->HighPart;
        }
}


////////////////////////////////////////////////////////////////////////////
//// RAWShortGet
//

USHORT RAWShortGet( PUCHAR pBuffer, BOOL bDoSwap )
{
        USHORTnResult;

        nResult = 0;

        if( !pBuffer )
        {
                return( 0 );
        }

        if( bDoSwap )
        {
                /* Perform Byte Swap
                -------------------- */
                nResult = ( (pBuffer[ 0 ] << 8) & 0xFF00 )
                        | ( pBuffer[ 1 ] & 0x00FF );
        }
        else
        {
                /* Do Not Perform Byte Swap
                --------------------------- */
                nResult = ( (pBuffer[ 1 ] << 8) & 0xFF00 )
                        | ( pBuffer[ 0 ] & 0x00FF );
        }

        return( nResult );
```

```
}


//////////////////////////////////////////////////////////////////////////
//// LOWERAcceptPacket
//
// Purpose
//   This function examines the packet header and lookahead buffer to
// determine if the packet should be accepted at the lower edge of
// the protocol driver.
//
// Parameters
//
// Return Value
//   Returns TRUE if packet is to be accepted.
//
// Remarks
// For this sample, the function simply filters based upon the value
// of pLOWERContext->bNoLoopback.
//

BOOL LOWERAcceptPacket(
          PLOWER_CONTEXT pLOWERContext,
          IN PVOID HeaderBuffer,
          IN UINT HeaderBufferSize,// Should Be MHdrSize (i.e., 14)
          IN PVOID LookAheadBuffer,
          IN UINT LookaheadBufferSize
          )
{
     /* Handle No Loopback Option
     --------------------------- */
     if( pLOWERContext->bNoLoopback )
     {
          /* Compare Source Address With Adapter Address
          ------------------------------------------- */
          // ATTENTION!!! Wrapper function to use instead of memcmp???
          if( memcmp(
                         &((PBYTE )HeaderBuffer)[ MSrcAddr ],
                         pLOWERContext->szAdapterAddress,
                         ETHER_ADDR_LENGTH
                         ) == 0
                 )
          {
                 /* Source Is This Adapter!!!
                 --------------------------- */
                 return( FALSE );
          }
     }

     return( TRUE );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERReceiveHandler
//

NDIS_STATUS
LOWERReceiveHandler(
    IN NDIS_HANDLE ProtocolBindingContext,
    IN NDIS_HANDLE MacReceiveContext,
    IN PVOID       HeaderBuffer,
    IN UINT        HeaderBufferSize,
    IN PVOID       LookAheadBuffer,
    IN UINT        LookaheadBufferSize,
    IN UINT        PacketSize
    )
{
     PLOWER_CONTEXT pLOWERContext;
     PNDIS_BUFFERpBuffer;
     PRAW_PACKET  pRAWPacket;
     NDIS_STATUS  nNdisStatus;
     BOOL                        bResult;
     LARGE_INTEGERnTickCount;

#if DBG
//   DbgPrint( "WinMIP: ReceiveHandler Entry; Packet Size: %d - %d\n",
//        PacketSize, RAWShortGet( &((PBYTE )HeaderBuffer)[ MLength ], TRUE ) );
#endif

     /* Sanity Checks On Arguments
     ---------------------------- */
     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return( NDIS_STATUS_SUCCESS );
     }

     /* Get Driver Context
     -------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     /* Filter Received Packet
     -------------------------
      * Return NDIS_STATUS_NOT_ACCEPTED if the filter decision is not to
      * accept the packet.
      */

     bResult = LOWERAcceptPacket(
                                 pLOWERContext,
                        HeaderBuffer,
                        HeaderBufferSize,
                        LookAheadBuffer,
                        LookaheadBufferSize
                                 );

     if( !bResult )
     {
          return( NDIS_STATUS_NOT_ACCEPTED );
     }

     /* Increment The Packet Number
```

```
      ------------------------------ */
      RAWLargeIntegerIncrement( &pLOWERContext->nLastPacketNumber );

      /* Allocate The Receive Packet Descriptor
      ---------------------------------------- */
      NdisAllocatePacket(
           &nNdisStatus,
           &(PNDIS_PACKET )pRAWPacket,
           pLOWERContext->hPacketPool
           );

      if( nNdisStatus != NDIS_STATUS_SUCCESS || !pRAWPacket )
      {
           RAWLargeIntegerIncrement( &pLOWERContext->nTossedPacketCount );

#if DBG
           DbgPrint( "LOWERReceiveHandler Could Not Allocate Packet\n" );
#endif

           return( NDIS_STATUS_RESOURCES );// Must Drop It
      }

      /* Initialize The Packet Signature
      --------------------------------- */
      pRAWPacket->Reserved.Signature = RAW_PACKET_SIGN;

      /* Copy Header Buffer
      -------------------- */
      pRAWPacket->Reserved.UserPacketData.nPacketDataLength = HeaderBufferSize;

      NdisMoveMemory(
           pRAWPacket->Reserved.UserPacketData.PacketBuffer,
           (PBYTE )HeaderBuffer,
           HeaderBufferSize
           );

      /* Allocate An NDIS Buffer Descriptor For The Receive Data Buffer
      ---------------------------------------------------------------- */
      NdisAllocateBuffer(
           &nNdisStatus,
           &pBuffer,
           pLOWERContext->hBufferPool,
           &pRAWPacket->Reserved.UserPacketData.PacketBuffer[ HeaderBufferSize ],
           (MAX_ETHER_SIZE - HeaderBufferSize)
           );

      if( nNdisStatus != NDIS_STATUS_SUCCESS || !pBuffer )
      {
           RAWLargeIntegerIncrement( &pLOWERContext->nTossedPacketCount );

           NdisFreePacket( (PNDIS_PACKET )pRAWPacket );

#if DBG
           DbgPrint( "LOWERReceiveHandler Could Not Allocate Buffer\n" );
#endif

           return( NDIS_STATUS_RESOURCES );// Must Drop It
      }
      else
      {
```

```
            NdisChainBufferAtFront( (PNDIS_PACKET )pRAWPacket, pBuffer );
      }

      /* Queue The Packet For Reception
      ------------------------------- */
      pRAWPacket->Reserved.UserPacketData.nPacketNumber.LowPart =
            pLOWERContext->nLastPacketNumber.LowPart;

      pRAWPacket->Reserved.UserPacketData.nPacketNumber.HighPart =
            pLOWERContext->nLastPacketNumber.HighPart;

      KeQueryTickCount( &nTickCount );

      pRAWPacket->Reserved.UserPacketData.nPacketTime = nTickCount.LowPart;

      pRAWPacket->Reserved.UserPacketData.nSelectedMedium =
            (UINT )pLOWERContext->nSelectedMedium;

      pRAWPacket->Reserved.nTransferDataStatus = NDIS_STATUS_PENDING;
      pRAWPacket->Reserved.nBytesTransferred = 0;

      // BUGBUG!!! Save - or at least check - MAC length...

      NdisInterlockedInsertTailList(
            &pLOWERContext->PacketReceiveList,
            &pRAWPacket->Reserved.ListElement,
            &pLOWERContext->ReceiveListSpinLock
            );

      /* Initiate Transfer Data
      ------------------------
       * In NDIS terms, the “data” to be transfered is the 802.3 data,
       * which begins after the 14-byte 802.3 header.
       */
      NdisTransferData(
            &pRAWPacket->Reserved.nTransferDataStatus,
            pLOWERContext->hNdisAdapterHandle,// From NdisOpenAdapter
            MacReceiveContext,                        // Argument To this
            0,                                                              //
ByteOffset From First Byte After Header
            PacketSize,                                                     //
BytesToTransfer
            (PNDIS_PACKET )pRAWPacket,// Destination Packet
            &pRAWPacket->Reserved.nBytesTransferred
            );

      if( pRAWPacket->Reserved.nTransferDataStatus != NDIS_STATUS_PENDING )
      {
            /* The Transfer Didn’t Pend
            -------------------------- */
            LOWERTransferDataCompleteHandler (
                    ProtocolBindingContext,
                    (PNDIS_PACKET )pRAWPacket,
                    pRAWPacket->Reserved.nTransferDataStatus,
                    pRAWPacket->Reserved.nBytesTransferred
                    );
      }

      return( NDIS_STATUS_SUCCESS );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERReceiveCompleteHandler
//


VOID LOWERReceiveCompleteHandler(
    IN NDIS_HANDLE  ProtocolBindingContext
    )
{
     PLOWER_CONTEXT pLOWERContext;
     PRAW_PACKET          pRAWPacket;
     PLIST_ENTRY          linkage;

#if DBG
//   DbgPrint( “ReceiveCompleteHandler Entry...\n” );
#endif

     /* Sanity Checks On Arguments
     ---------------------------- */
     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return;
     }

     /* Get Driver Context
     --------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     /* Check For Packets Waiting In The PacketReceiveList
     ---------------------------------------------------- */
     NdisAcquireSpinLock( &pLOWERContext->ReceiveListSpinLock );

     while( !IsListEmpty( &pLOWERContext->PacketReceiveList ) )
     {
          /* Find The Oldest Packet
          ------------------------- */
          linkage = (PLIST_ENTRY )pLOWERContext->PacketReceiveList.Flink;

          pRAWPacket = CONTAINING_RECORD(
                                                              linkage,
                                                              RAW_PACKET,

Reserved.ListElement
                                                              );

          if( pRAWPacket->Reserved.nTransferDataStatus == NDIS_STATUS_PENDING )
          {
                  NdisReleaseSpinLock( &pLOWERContext->ReceiveListSpinLock );

                  return;
          }

#ifdef ZNEVER
          linkage = NdisInterlockedRemoveHeadList(

&pLOWERContext->PacketReceiveList,

&pLOWERContext->ReceiveListSpinLock
```

```
                                                        );
#else
            // SpinLock Already Held...
            linkage = RemoveHeadList(
                                            &pLOWERContext->PacketReceiveList,
                                            );
#endif

            pRAWPacket = CONTAINING_RECORD(
                                                        linkage,
                                                        RAW_PACKET,
                                                        Reserved.ListElement
                                                        );

            /* Verify The Packet Pointer
            ---------------------------- */
            ASSERT( pRAWPacket );

            if( !pRAWPacket )
            {
                    break;
            }

            /* Verify The Packet Signature
            ------------------------------ */
            ASSERT( pRAWPacket->Reserved.Signature == RAW_PACKET_SIGN );

            if( pRAWPacket->Reserved.Signature != RAW_PACKET_SIGN )
            {
#if DBG
                    DbgPrint( "LOWERReceiveCompleteHandler: Invalid Packet
Signature\n" );
#endif

                    break;
            }

            NdisReleaseSpinLock( &pLOWERContext->ReceiveListSpinLock );

            /* Handle Packet Based On TransferData Status
            ------------------------------------------ */
            if( pRAWPacket->Reserved.nTransferDataStatus == NDIS_STATUS_SUCCESS )
            {
                    UPPERPacketReceived( &pRAWPacket->Reserved.UserPacketData );
            }
            else
            {
                    // BUGBUG!!! Update Statistics...

#if DBG
                    DbgPrint( "LOWERReceiveCompleteHandler: Transfer Data Failed\n"
);
#endif

#if DBG
                    DbgPrint( "TransferCompleteStatus: 0x%x\n",
                                    pRAWPacket->Reserved.nTransferDataStatus );
#endif // DBG

                    /* Free The Packet And It's Buffers
```

```
                    ---------------------------------- */
                    LOWERFreePacketAndBuffers( pRAWPacket );
               }

               NdisAcquireSpinLock( &pLOWERContext->ReceiveListSpinLock );
          }

          NdisReleaseSpinLock( &pLOWERContext->ReceiveListSpinLock );
     }


     ////////////////////////////////////////////////////////////////////////
     //// LOWERStatusHandler
     //

     VOID LOWERStatusHandler(
         IN NDIS_HANDLE   ProtocolBindingContext,
         IN NDIS_STATUS   GeneralStatus,
         IN PVOID         StatusBuffer,
         IN UINT          StatusBufferSize
         )
     {
         IF_LOUD(DbgPrint("WinMIP: StatusHandler Entry...\n");)

          /* Sanity Checks
          ---------------- */
          ASSERT( g_NdisProtocolHandle );

          if( !g_NdisProtocolHandle )
          {
               return;
          }

         return;
     }


     ////////////////////////////////////////////////////////////////////////
     //// LOWERStatusCompleteHandler
     //

     VOID LOWERStatusCompleteHandler(
         IN NDIS_HANDLE  ProtocolBindingContext)
     {
         IF_LOUD(DbgPrint("WinMIP: StatusCompleteHandler Entry...\n");)

          /* Sanity Checks
          ---------------- */
          ASSERT( g_NdisProtocolHandle );

          if( !g_NdisProtocolHandle )
          {
               return;
          }

         return;
     }
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERBindAdapterHandler
//
//
// Remarks
// Windows NT supports the NDIS 3.0 specification for protocol drivers. This
// means that the BindAdapterHandler function isn't automatically called
// under Windows NT.
//

VOID LOWERBindAdapterHandler(
     OUT PNDIS_STATUS    pProtocolBindStatus,
     IN  NDIS_HANDLE     BindAdapterContext,
     IN  PNDIS_STRING    AdapterName,
     IN  PVOID           SystemSpecific1,
     IN  PVOID           SystemSpecific2
     )
{
     PLOWER_CONTEXTpLOWERContext;
     NDIS_STATUS          nNdisStatus;

     IF_LOUD(DbgPrint("BindAdapterHandler Entry...");)

     *pProtocolBindStatus = NDIS_STATUS_FAILURE;// Assume Failure

     /* Sanity Checks
     ---------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     /* Display The "AdapterName"
     ----------------------------
      * In the Windows NT environment, the "AdapterName" is really the exported
      * DosDevice name of the MAC adapter. A typical "AdapterName", as displayed
      * by the following DbgPrintf, would be:
      *
      *           30, 32, "\Device\NE20001"
      */
#if DBG
     DbgPrint( "AdapterName : %d, %d, \042%*.*ws\042\n",
          AdapterName->Length,
          AdapterName->MaximumLength,
          AdapterName->Length/sizeof( wchar_t ),
          AdapterName->Length/sizeof( wchar_t ),
          AdapterName->Buffer
          );
#endif

#ifdef SINGLE_ADAPTER_BINDING
     /* Only Allow Binding To One Adapter
     -----------------------------------
      * The general architecture of the WinMIP driver could be extended
      * to support concurrent binding to multiple Ethernet adapters. However,
      * it is not done in this example because of the additional complexity
      * which would be introduced in the upper-edge API.
      *
```

```
         * At this point the lower-edge binding list is examined. If a binding
         * already exists, return without calling NdisOpenAdapter.
         */
        ASSERT( IsListEmpty( &g_LowerBindingList ) );

        if( !IsListEmpty( &g_LowerBindingList ) )
        {
             return;
        }
#endif

        pLOWERContext = LOWERAllocContext();

        ASSERT( pLOWERContext );

        if( !pLOWERContext )
        {
             return;
        }

        /* Save AdapterName
        ------------------- */
        pLOWERContext->szUniAdapterName.Length = AdapterName->Length;
        pLOWERContext->szUniAdapterName.MaximumLength = AdapterName->MaximumLength;

        nNdisStatus = NdisAllocateMemory(

&pLOWERContext->szUniAdapterName.Buffer,

pLOWERContext->szUniAdapterName.MaximumLength,
                                          0,                                //
Allocate non-paged system-space memory
                                          HighestAcceptableMax
                                          );

        ASSERT( nNdisStatus == NDIS_STATUS_SUCCESS );

        if( nNdisStatus != NDIS_STATUS_SUCCESS )
        {
             return;
        }

        NdisMoveMemory(
             pLOWERContext->szUniAdapterName.Buffer,
             AdapterName->Buffer,
             pLOWERContext->szUniAdapterName.Length
             );

        pLOWERContext->szUniAdapterName.Buffer[
pLOWERContext->szUniAdapterName.Length/sizeof( wchar_t ) ] = 0;

        /* Assume Failure
        ----------------- */
        pLOWERContext->nOpenAdapterStatus = NDIS_STATUS_FAILURE;
        pLOWERContext->nOpenAdapterErrorStatus = NDIS_STATUS_FAILURE;

        /* Specify Desired Mediums
        ------------------------- */
        pLOWERContext->AdapterMediumArray[ 0 ] = NdisMedium802_3;
        pLOWERContext->AdapterMediumArray[ 1 ] = NdisMedium802_5;
```

```
	pLOWERContext->AdapterMediumArray[ 2 ] = NdisMediumLocalTalk;

	/* Save The BindAdapterContext
	----------------------------- */
	pLOWERContext->hBindAdapterContext = BindAdapterContext;

	pLOWERContext->nSelectedMedium = -1;// Medium Not Yet Known

	//
	//  Try to open the MAC
	//

	IF_LOUD(DbgPrint("Calling NdisOpenAdapter...");)

	NdisOpenAdapter(
		&pLOWERContext->nOpenAdapterStatus,
		&pLOWERContext->nOpenAdapterErrorStatus,
		&pLOWERContext->hNdisAdapterHandle,
		&pLOWERContext->nSelectedMediumIndex,
		&pLOWERContext->AdapterMediumArray[ 0 ],
		3,                                                              //
MediumArraySize
		g_NdisProtocolHandle,
		(NDIS_HANDLE )pLOWERContext,// Specify ProtocolBindingContext
		AdapterName,
		0,
		NULL
		);

	/* ATTENTION!!!
	---------------
	 * This function always reports NDIS_STATUS_SUCCESS - even
	 * in the case that the NdisOpenAdapter call returns NDIS_STATUS_PENDING.
	 * A better (possibly) approach would be for this function to return
	 * NDIS_STATUS_PENDING instead of NDIS_STATUS_SUCCESS if NdisOpenAdapter
	 * returned NDIS_STATUS_PENDING; if this change was made, then
	 * NdisBindAdapterComplete would need to be called from
	 * LOWEROpenAdapterCompleteHandler.
	 */

	if( pLOWERContext->nOpenAdapterStatus != NDIS_STATUS_PENDING )
	{
		LOWEROpenAdapterCompleteHandler(
			(NDIS_HANDLE )pLOWERContext,// ProtocolBindingContext
			pLOWERContext->nOpenAdapterStatus,
			pLOWERContext->nOpenAdapterErrorStatus
			);
	}

	*pProtocolBindStatus = NDIS_STATUS_SUCCESS;// Report Success

	IF_LOUD(DbgPrint("BindAdapterHandler Exit\n");)
}
```

```
//////////////////////////////////////////////////////////////////////////
//// LOWERUnbindAdapterHandler
//
// Remarks
// Windows NT supports the NDIS 3.0 specification for protocol drivers. This
// means that the UnbindAdapterHandler function isn't automatically called
// under Windows NT.
//
// The WinMIP uses a UnbindAdapterHandler, which is called by
// the driver itself when unbinding is necessary.
//

VOID LOWERUnbindAdapterHandler(
     OUT PNDIS_STATUS    pStatus,
     IN  NDIS_HANDLE     ProtocolBindingContext,
     IN  NDIS_HANDLE     UnbindAdapterContext
     )
{
     PLOWER_CONTEXTpLOWERContext;

     IF_LOUD(DbgPrint("UnbindAdapterHandler Entry...");)

     /* Sanity Checks
     ---------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     ASSERT( ProtocolBindingContext );

     if( !ProtocolBindingContext )
     {
          return;
     }

     /* Get Driver Context
     --------------------- */
     pLOWERContext = (PLOWER_CONTEXT )ProtocolBindingContext;

     // ATTENTION!!! Unload Upper-Edge Bindings!!!

     //
     // This event is used in case any of the NDIS requests
     // pend; we wait until it is set by the completion
     // routine, which also sets NdisRequestStatus.
     //
     KeInitializeEvent(
          &pLOWERContext->NdisRequestEvent,
          NotificationEvent,
          FALSE
          );

     NdisCloseAdapter(
          pStatus,
          pLOWERContext->hNdisAdapterHandle// Handle From NdisOpenAdapter
          );
```

```
        if( *pStatus == NDIS_STATUS_PENDING )
        {
             //
             // The completion routine will set NdisRequestStatus.
             //
             KeWaitForSingleObject(
                     &pLOWERContext->NdisRequestEvent,
                     Executive,
                     KernelMode,
                     TRUE,
                     (PLARGE_INTEGER)NULL
                     );

             *pStatus = pLOWERContext->NdisRequestStatus;

             KeResetEvent( &pLOWERContext->NdisRequestEvent );
        }
        else
        {
             LOWERCloseAdapterCompleteHandler(
                     ProtocolBindingContext,
                     *pStatus
                     );
        }

        LOWERFreeContext( pLOWERContext );
    }
```

A.4 WinMIP.h

```
#include “IOCTL.h”

//////////////////////////////////////////////////////////////////////////
//// PROTOCOL LOWER-EDGE BINDING CONTEXT

#define RAW_PACKET_SIGN     (UINT)0x52415745

/*
-------------------------------------- */
typedef
struct _RAW_PACKET_RESERVED
{
     UINT                                          Signature;           //
RAW_PACKET_SIGN

     LIST_ENTRY     ListElement;
//   PIRP                                          pIrp;                //
ATTENTION!!! Get Rid Of This!!!
//   PMDL                                          pMdl;                //
ATTENTION!!! Get Rid Of This!!!
     DWORD                                         nUPPERCallerData;

     NDIS_STATUS                 nTransferDataStatus;
     UINT                                          nBytesTransferred;

     USER_PACKET_DATAUserPacketData;

     DWORD                                         nPackingInsurance;// Spare.
Do Not Use.
}
     RAW_PACKET_RESERVED, *PRAW_PACKET_RESERVED;

#define RESERVED(_p) ((PRAW_PACKET_RESERVED)((_p)->ProtocolReserved))


/*
-------------------------------------- */
typedef
struct _RAW_PACKET
{
     NDIS_PACKET    Packet;
     RAW_PACKET_RESERVEDReserved;
}
     RAW_PACKET, *PRAW_PACKET;


/* Structure For Making An NDIS Request
-------------------------------------- */
typedef
struct _INTERNAL_REQUEST
{
     LIST_ENTRY     ListElement;
     PIRP           pIrp;
     NDIS_REQUEST   Request;
}
     INTERNAL_REQUEST, *PINTERNAL_REQUEST;

#define  MAX_REQUESTS   4
```

```
typedef
enum
{
     RAW_UNDEFINED_EXTENSION,
     RAW_ADAPTER_EXTENSION,
     RAW_PROTOCOL_EXTENSION
}
     RAW_EXTENSION_TYPE;


typedef
struct _DEVICE_ADAPTER_EXTENSION
{
     NDIS_STRING                 AdapterName;

     PWSTR                                          BindString;
     PWSTR                                          ExportString;

     PIRP                                           pOpenCloseIrp;
}
     DEVICE_ADAPTER_EXTENSION, *PDEVICE_ADAPTER_EXTENSION;


typedef
struct _DEVICE_PROTOCOL_EXTENSION
{
     PWSTR                               ExportString;

     PIRP                                pOpenCloseIrp;
}
     DEVICE_PROTOCOL_EXTENSION, *PDEVICE_PROTOCOL_EXTENSION;

//
// Driver device extension.
//

typedef
struct _DEVICE_EXTENSION
{
     RAW_EXTENSION_TYPEnExtensionType;

     PDEVICE_OBJECT       pDeviceObject;

     /* Queue For Pending Reads On The Device
     ---------------------------------------- */
     KSPIN_LOCK           ReadListSpinLock;
     LIST_ENTRY           ReadList;

     union
     {
          DEVICE_ADAPTER_EXTENSIONadapterExt;
          DEVICE_PROTOCOL_EXTENSIONprotocolExt;
     }
          u;
}
     DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
/* Lower-Edge Binding Context
-----------------------------
 * There is one of these for each adapter which the protocol is bound to.
 * The global variable g_LowerBindingList contains the list of lower
 * bindings.
 *
 * This sample does not support concurrent binding to multiple adapters
 * because it would make the upper-edge API too complex for a simple sample.
 * See the additional comment with the BindAdapterHandler function.
 */
typedef
struct _LOWER_CONTEXT
{
     LIST_ENTRY           qLink;

     UINT                                 Signature;

     BOOL                                 bLowerContextInitDone;

     PDEVICE_EXTENSION   DeviceExtension;

     /* Adapter Open Fields
     ---------------------- */
     BOOL                                 bAdapterOpen;                  // TRUE
If Adapter Has Been Opened
     NDIS_HANDLE  hNdisAdapterHandle;// returned from NdisOpenAdapter
     NDIS_HANDLE          hBindAdapterContext;// To Call NdisCompleteBindAdapter
     NDIS_STATUS  nOpenAdapterStatus;
     NDIS_STATUS  nOpenAdapterErrorStatus;

     UINT         nSelectedMediumIndex;
     NDIS_MEDIUM          nSelectedMedium;
     NDIS_MEDIUM  AdapterMediumArray[ 4 ];

     NDIS_HANDLE          hUnbindAdapterContext;// To Call
NdisCompleteUnbindAdapter
     NDIS_STATUS  nCloseAdapterStatus;

     /* Packet Filter Fields
     ---------------------- */
     BOOL                                 bFilterSet;                    // TRUE
If Packet Filter Has Been Set
     NDIS_STATUS          nSetPacketFilterStatus;
     ULONG                                nPacketFilter;       // Ndis Packet
Filter Bits For EtherTalk

     /* Adapter Address Fields
     ------------------------- */
     NDIS_STATUS  nAdapterAddressStatus;
     BYTE                         szAdapterAddress[ ETHER_ADDR_LENGTH ];

     /* Adapter Name
     -------------- */
     NDIS_STRING  szUniAdapterName;

     /* Driver Options
     ----------------- */
     BOOL                         bNoLoopback;

     /* Packet Statistics
```

```
      -------------------- */
      LARGE_INTEGERnLastPacketNumber;
      LARGE_INTEGERnTossedPacketCount;// Because Of Lower-Edge Resources...

      NDIS_SPIN_LOCKReceiveListSpinLock;
      LIST_ENTRY           PacketReceiveList;


      NDIS_HANDLE                 hPacketPool;
      NDIS_HANDLE                 hBufferPool;

      KSPIN_LOCK          RcvQSpinLock;
      LIST_ENTRY          RcvList;

//    PIRP                OpenCloseIrp;

      KEVENT              NdisRequestEvent;// used for pended requests.
      NDIS_STATUS         NdisRequestStatus;// records request status.

      NDIS_SPIN_LOCK      RequestSpinLock;
      LIST_ENTRY          RequestList;

      LIST_ENTRY          ResetIrpList;

      INTERNAL_REQUEST    Requests[MAX_REQUESTS];
}
      LOWER_CONTEXT, *PLOWER_CONTEXT;

#defineRAW_PACKET_POOL_SIZE64
#defineRAW_BUFFER_POOL_SIZE64

extern PDRIVER_OBJECTg_pTheDriverObject;

extern NDIS_HANDLEg_NdisProtocolHandle;// returned from NdisRegisterProtocol

extern LIST_ENTRYg_LowerBindingList;
extern LIST_ENTRYg_PendingSendListList;

VOID RAWETHERDUnloadProtocol( void );
VOID RAWETHERDriverUnload( IN PDRIVER_OBJECT DriverObject );

//
// This constant is used for places where NdisAllocateMemory
// needs to be called and the HighestAcceptableAddress does
// not matter.
//
NDIS_PHYSICAL_ADDRESS HighestAcceptableMax =
    NDIS_PHYSICAL_ADDRESS_CONST(-1,-1);

#endif // __WinMIP_H__



////////////////////////////////////////////////////////////////////////
//// INCLUDE FILES

#include “stdarg.h”
#include “ntddk.h”
#include “ntiologc.h”
#include “ndis.h”
```

```
#include “debug.h”
#include “WinMIP.h”
#include“RAWLOWER.h”
#include“ADAPTDEV.H”
#include“PROTODEV.H”


/* NDIS Binding Context
---------------------- */
PDRIVER_OBJECT                          g_pTheDriverObject = NULL;

NDIS_HANDLE                                         g_NdisProtocolHandle = 0;//
returned from NdisRegisterProtocol
NDIS_PROTOCOL_CHARACTERISTICSProtocolChar;
LIST_ENTRY                                          g_LowerBindingList;

int
g_nUseCount = 0;

////////////////////////////////////////////////////////////////////////
//// RAWETHERReadRegistry
//
// Purpose
//
// Parameters
//
// Return Value
// Status is returned.
//

NTSTATUS
RAWETHERReadRegistry(
    IN  PWSTR             *MacDriverName,
    IN  PWSTR             *RAWETHERDriverName,
    IN  PUNICODE_STRING    RegistryPath
    );


#if DBG
//
// Declare the global debug flag for this driver.
//

ULONG RAWETHERDebugFlag = RAWETHER_DEBUG_LOUD;

#endif
```

```
//////////////////////////////////////////////////////////////////////////
//// _DeviceOpen
//
// Purpose
// This is the dispatch routine for device create/open requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//
//

NTSTATUS
_DeviceOpen(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceOpen\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceOpen( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceOpen( pDeviceObject, pIrp ) );

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// _DeviceClose
//
// Purpose
// This is the dispatch routine for device close requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//

NTSTATUS
_DeviceClose(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceClose\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceClose( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceClose( pDeviceObject, pIrp ) );

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// _DeviceRead
//
// Purpose
// This is the dispatch routine for device read requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//

NTSTATUS
_DeviceRead(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceRead\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceRead( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceRead( pDeviceObject, pIrp ) );

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// _DeviceWrite
//
// Purpose
// This is the dispatch routine for device write requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//

NTSTATUS
_DeviceWrite(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceWrite\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceWrite( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceWrite( pDeviceObject, pIrp ) );

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// _DeviceCleanup
//
// Purpose
// This is the dispatch routine for device cleanup requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//

NTSTATUS
_DeviceCleanup(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceCleanup\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceCleanup( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceCleanup( pDeviceObject, pIrp ) );

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
////////////////////////////////////////////////////////////////////////
//// _DeviceIoControl
//
// Purpose
// This is the dispatch routine for device IOCTL requests.
//
// Parameters
//    pDeviceObject - Pointer to the device object.
//    pIrp - Pointer to the request packet.
//
// Return Value
// Status is returned.
//
// Remarks
// The driver supports two kinds of device drivers: a PROTOCOL
// device and ADAPTER devices. This routine dispatches to a more specific
// routine depending on kind of device.
//

NTSTATUS
_DeviceIoControl(
    IN PDEVICE_OBJECT pDeviceObject,
    IN PIRP pIrp
    )
{
     PDEVICE_EXTENSION pDeviceExtension;

     IF_LOUD(DbgPrint("WinMIP: _DeviceIoControl\n");)

     pDeviceExtension = pDeviceObject->DeviceExtension;

     switch( pDeviceExtension->nExtensionType )
     {
          case RAW_ADAPTER_EXTENSION:
                  return( RAWETHERAdapterDeviceIoControl( pDeviceObject, pIrp ) );

          case RAW_PROTOCOL_EXTENSION:
                  return( RAWETHERProtocolDeviceIoControl( pDeviceObject, pIrp )
);

          default:
                  break;
     }

     pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
     return( STATUS_UNSUCCESSFUL );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// DriverEntry
//
// Purpose
// This routine initializes the driver.
//
// Parameters
//   pDriverObject - Pointer to driver object created by system.
//   RegistryPath - Pointer to the Unicode name of the registry path
//                  for this driver.
//
// Return Value
// The function return value is the final status from the initialization
// operation.
//

NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT pDriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
     NDIS_PROTOCOL_CHARACTERISTICS  ProtocolChar;

     UNICODE_STRING MacDriverName;
     UNICODE_STRING UnicodeDeviceName;

     PDEVICE_OBJECT pDeviceObject = NULL;
     PDEVICE_EXTENSION pDeviceExtension = NULL;

     staticNTSTATUS RegisterStatus = STATUS_SUCCESS;
     NTSTATUS Status = STATUS_SUCCESS;
     NTSTATUS ErrorCode = STATUS_SUCCESS;

     ULONG          DevicesCreated=0;

     PWSTR          BindString;
     PWSTR          ExportString;

     PWSTR          BindStringSave;
     PWSTR          ExportStringSave;

     IF_LOUD(DbgPrint("\n\nWinMIP: DriverEntry\n");)

     g_pTheDriverObject = pDriverObject;

     InitializeListHead( &g_LowerBindingList );

     NdisZeroMemory(&ProtocolChar,sizeof(NDIS_PROTOCOL_CHARACTERISTICS));

     ProtocolChar.MajorNdisVersion            = 3;
     ProtocolChar.MinorNdisVersion            = 0;
     ProtocolChar.Reserved                    = 0;
     ProtocolChar.OpenAdapterCompleteHandler  = LOWEROpenAdapterCompleteHandler;
     ProtocolChar.CloseAdapterCompleteHandler = LOWERCloseAdapterCompleteHandler;
     ProtocolChar.SendCompleteHandler         = LOWERSendCompleteHandler;
     ProtocolChar.TransferDataCompleteHandler = LOWERTransferDataCompleteHandler;
     ProtocolChar.ResetCompleteHandler        = LOWERResetCompleteHandler;
     ProtocolChar.RequestCompleteHandler      = LOWERRequestCompleteHandler;
     ProtocolChar.ReceiveHandler              = LOWERReceiveHandler;
```

```
      ProtocolChar.ReceiveCompleteHandler      = LOWERReceiveCompleteHandler;
      ProtocolChar.StatusHandler               = LOWERStatusHandler;
      ProtocolChar.StatusCompleteHandler       = LOWERStatusCompleteHandler;

      // ATTENTION!!! Change To NdisxxxInitString
      RtlInitString( &ProtocolChar.Name, RAWETHER_TRANSPORT_NAME );

// The NdisRegisterProtocol request is called when a PROTOCOL module
// initializes. It provides the NDIS interface with information about the
// PROTOCOL driver, including the addresses of its request handlers.

// The PROTOCOL driver passes in a characteristics table. This table is
// copied by the NdisRegisterProtocol request to its own internal storage.
// Thus, once registered, the PROTOCOL driver cannot alter its handler
// routines.

      NdisRegisterProtocol(
           &RegisterStatus,
           &g_NdisProtocolHandle,
           &ProtocolChar,
           sizeof(NDIS_PROTOCOL_CHARACTERISTICS)
           );

      if (RegisterStatus != NDIS_STATUS_SUCCESS)
      {
           IF_LOUD(DbgPrint("WinMIP: Failed to register protocol with NDIS\n");)

           g_pTheDriverObject = NULL;

           return RegisterStatus;
      }

      //
      // Set up the adapter device entry points.
      //

      pDriverObject->MajorFunction[IRP_MJ_CREATE] = _DeviceOpen;
      pDriverObject->MajorFunction[IRP_MJ_CLOSE]  = _DeviceClose;
      pDriverObject->MajorFunction[IRP_MJ_READ]   = _DeviceRead;
      pDriverObject->MajorFunction[IRP_MJ_WRITE]  = _DeviceWrite;
      pDriverObject->MajorFunction[IRP_MJ_CLEANUP]  = _DeviceCleanup;
      pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  = _DeviceIoControl;

      pDriverObject->DriverUnload = RAWETHERDriverUnload;


      //
      //  Get the name of the driver and the name of the MAC driver
      //  to bind to from the registry
      //

      Status=RAWETHERReadRegistry(
                          &BindString,
                          &ExportString,
                          RegistryPath
                          );

      if (Status != STATUS_SUCCESS)
      {
           IF_LOUD(DbgPrint("WinMIP: Failed to read registry\n");)
```

```
            goto RegistryError;
      }

      BindStringSave   = BindString;
      ExportStringSave = ExportString;


      //
      //  create a device object for each entry
      //
      while (*BindString!= UNICODE_NULL && *ExportString!= UNICODE_NULL)
      {
            //
            //  Create a counted unicode string for both null terminated strings
            //

            // ATTENTION!!! Change To NdisxxxInitString
            RtlInitUnicodeString(
                    &MacDriverName,
                    BindString
                    );

            // ATTENTION!!! Change To NdisxxxInitString
            RtlInitUnicodeString(
                    &UnicodeDeviceName,
                    ExportString
                    );

            //
            //  Advance to the next string of the MULTI_SZ string
            //
            BindString   +=
(MacDriverName.Length+sizeof(UNICODE_NULL))/sizeof(WCHAR);

            ExportString +=
(UnicodeDeviceName.Length+sizeof(UNICODE_NULL))/sizeof(WCHAR);

            IF_LOUD(DbgPrint("WinMIP: DeviceName=%ws
MacName=%ws\n",UnicodeDeviceName.Buffer,MacDriverName.Buffer);)

                        Status = IoCreateDevice(
                                    pDriverObject,
                                    sizeof(DEVICE_EXTENSION),
                                    &UnicodeDeviceName,
                                    FILE_DEVICE_TRANSPORT,
                                    0,                              //
Standard device characteristics
                                    FALSE,              // This isn't an
exclusive device
                                    &pDeviceObject
                                    );

            if (Status != STATUS_SUCCESS)
            {
                    IF_LOUD(DbgPrint("WinMIP: IoCreateDevice() failed:\n");)

                    break;
            }
```

```
            /* Simulate NDIS 3.1 Binding Mechanism
            -------------------------------------
             * The NDIS 3.1 binding mechanism is emulated.
             * At this point in DriverEntry the information necessary to fake the
             * BindAdapterHandler call is available and it is called.
             *
             * This means that, unlike the Packet Driver example, WinMIP bindings
             * are made when the driver is loaded - not when the device associated
             * with the binding is opened.
             *
             * Alternative mechanisms could also be used. For example, instead of
             * binding at this point, binding could be deferred until a device
             * OPEN is made which references the adapter. Using this approach, a
             * reference count should be maintained. The binding would remain
             * open as long as the binding reference count was non-zero.
             */
            LOWERBindAdapterHandler(
                    &RegisterStatus,// pProtocolBindStatus
                    NULL,                                           //
    BindAdapterContext
                    &MacDriverName,// AdapterName
                    NULL,                                           // SystemSpecific1
                    NULL                                            // SystemSpecific2
                    );

            DevicesCreated++;

            pDeviceObject->Flags |= DO_DIRECT_IO;
            pDeviceExtension  =  (PDEVICE_EXTENSION)
    pDeviceObject->DeviceExtension;
            pDeviceExtension->pDeviceObject = pDeviceObject;

            pDeviceExtension->nExtensionType = RAW_ADAPTER_EXTENSION;

            /* Initialize Packet Receive List
            --------------------------------- */
            KeInitializeSpinLock(&pDeviceExtension->ReadListSpinLock);
            InitializeListHead( &pDeviceExtension->ReadList );

            //
            //  Save the the name of the MAC driver to open in the Device Extension
            //

            pDeviceExtension->u.adapterExt.AdapterName=MacDriverName;

            if (DevicesCreated == 1)
            {
                    pDeviceExtension->u.adapterExt.BindString   = BindStringSave;
                    pDeviceExtension->u.adapterExt.ExportString = ExportStringSave;
            }
      }

      if (DevicesCreated > 0)
      {
            // ATTENTION!!! Change To NdisxxxInitString
            RtlInitUnicodeString(
                    &UnicodeDeviceName,
                    L"\\Device\\WinMIP");

            /* Create The Protocol Device
```

```
            ----------------------------- */
            Status = IoCreateDevice(
                                        pDriverObject,
                                        sizeof(DEVICE_EXTENSION),
                                        &UnicodeDeviceName,
                                        FILE_DEVICE_TRANSPORT,
                                        0,                              //
Standard device characteristics
                                        FALSE,              // This isn't an
exclusive device
                                        &pDeviceObject
                                        );

            if (Status != STATUS_SUCCESS)
            {
                    IF_LOUD(DbgPrint("WinMIP: IoCreateDevice() failed:\n");)
            }
            else
            {
                    pDeviceObject->Flags |= DO_DIRECT_IO;
                    pDeviceExtension  =  (PDEVICE_EXTENSION)
pDeviceObject->DeviceExtension;
                    pDeviceExtension->pDeviceObject = pDeviceObject;

                    pDeviceExtension->nExtensionType = RAW_PROTOCOL_EXTENSION;

                    /* Initialize Packet Receive List
                    ------------------------------- */
                    KeInitializeSpinLock(&pDeviceExtension->ReadListSpinLock);
                    InitializeListHead( &pDeviceExtension->ReadList );
            }

            //
            //  Managed to create at least on device.
            //
            return STATUS_SUCCESS;
      }

      ExFreePool(BindStringSave);
      ExFreePool(ExportStringSave);

RegistryError:

      NdisDeregisterProtocol(
            &Status,
            g_NdisProtocolHandle
            );

      g_NdisProtocolHandle = 0;

      g_pTheDriverObject = NULL;

      Status=STATUS_UNSUCCESSFUL;

      return(Status);
}


//////////////////////////////////////////////////////////////////////////
//// RAWETHERUnloadProtocol
```

```
//
// Purpose
// Unload the NDIS 3.1 protocol.
//
// Parameters
//
// Return Value
//
// Remarks
// This function emulates the NDIS 3.1 UnloadHandler function, which is not
// implemented in the NT NDIS 3.0 and NDIS 4.0 NDIS wrapper.
//

VOID
RAWETHERUnloadProtocol( void )
{
     PLOWER_CONTEXTpLOWERContext;
     NDIS_STATUS nNdisStatus = NDIS_STATUS_FAILURE;

     IF_LOUD(DbgPrint("UnloadProtocolHandler Entry..." );)

     /* Sanity Checks
     --------------- */
     ASSERT( g_NdisProtocolHandle );

     if( !g_NdisProtocolHandle )
     {
          return;
     }

     /* The Lower Binding List Should Be Empty Already!
     ------------------------------------------------- */
     ASSERT( IsListEmpty( &g_LowerBindingList ) );

     while( !IsListEmpty( &g_LowerBindingList ) )
     {
          pLOWERContext = (PLOWER_CONTEXT )RemoveHeadList( &g_LowerBindingList );

          // ATTENTION!!! Unload Upper-Edge Bindings!!!

          /* Note
          -------
           * LOWERUnbindAdapterHandler() will call LOWERFreeContext(), so don't
           * touch pLOWERContext after this call returns.
           */
          LOWERUnbindAdapterHandler(
                  &nNdisStatus,
                  (NDIS_HANDLE )pLOWERContext,
                  NULL
                  );
     }

     NdisDeregisterProtocol(
          &nNdisStatus,
          g_NdisProtocolHandle// Handle From NdisRegisterProtocol
          );

     g_NdisProtocolHandle = 0;
}
```

```
//////////////////////////////////////////////////////////////////////////
//// RAWETHERDriverUnload
//
// Purpose
//
// Parameters
//   pDriverObject - Pointer to driver object created by system.
//
// Return Value
//

VOID
RAWETHERDriverUnload(
    IN PDRIVER_OBJECT pDriverObject
    )
{
     PDEVICE_OBJECT     DeviceObject;
     PDEVICE_OBJECT     OldDeviceObject;
     PDEVICE_EXTENSION  pDeviceExtension;
     NDIS_STATUS        nNdisStatus;

     IF_LOUD(DbgPrint("WinMIP: DriverUnload\n");)

     /* Delete The Driver's Devices
     ------------------------------ */
     DeviceObject = pDriverObject->DeviceObject;

     while( DeviceObject != NULL )
     {
          pDeviceExtension = DeviceObject->DeviceExtension;

          if( pDeviceExtension->u.adapterExt.BindString != NULL )
          {
                  ExFreePool( pDeviceExtension->u.adapterExt.BindString );
          }

          if( pDeviceExtension->u.adapterExt.ExportString != NULL )
          {
                  ExFreePool( pDeviceExtension->u.adapterExt.ExportString );
          }

          OldDeviceObject=DeviceObject;

          DeviceObject=DeviceObject->NextDevice;

          IoDeleteDevice(OldDeviceObject);
     }

     /* Unload The Protocol
     ---------------------- */
     RAWETHERUnloadProtocol();
}
```

```
//////////////////////////////////////////////////////////////////////////
//// RAWETHERQueryRegistryRoutine
//

NTSTATUS
RAWETHERQueryRegistryRoutine(
    IN PWSTR     ValueName,
    IN ULONG     ValueType,
    IN PVOID     ValueData,
    IN ULONG     ValueLength,
    IN PVOID     Context,
    IN PVOID     EntryContext
    )

{
     PUCHAR       Buffer;

     IF_LOUD(DbgPrint("WinMIP: QueryRegistryRoutine\n");)

     if (ValueType != REG_MULTI_SZ)
     {
          return STATUS_OBJECT_NAME_NOT_FOUND;
     }

     Buffer=ExAllocatePool(NonPagedPool,ValueLength);

     if (Buffer==NULL)
     {
          return STATUS_INSUFFICIENT_RESOURCES;
     }

     RtlCopyMemory( Buffer, ValueData, ValueLength );

     *((PUCHAR *)EntryContext)=Buffer;

     return STATUS_SUCCESS;
}
```

```
//////////////////////////////////////////////////////////////////////////
//// RAWETHERReadRegistry
//

NTSTATUS
RAWETHERReadRegistry(
    IN  PWSTR              *MacDriverName,
    IN  PWSTR              *RAWETHERDriverName,
    IN  PUNICODE_STRING     RegistryPath
    )

{
     NTSTATUS   Status;

     RTL_QUERY_REGISTRY_TABLE ParamTable[5];

     PWSTR      Bind       = L"Bind";
     PWSTR      Export     = L"Export";
     PWSTR      Parameters = L"Parameters";
     PWSTR      Linkage    = L"Linkage";

     PWCHAR     Path;

     Path=ExAllocatePool( PagedPool, RegistryPath->Length+sizeof(WCHAR) );

     if( Path == NULL )
     {
          return STATUS_INSUFFICIENT_RESOURCES;
     }

     RtlZeroMemory( Path, RegistryPath->Length+sizeof(WCHAR) );

     RtlCopyMemory(
          Path,
          RegistryPath->Buffer,
          RegistryPath->Length
          );

     IF_LOUD(DbgPrint("WinMIP: Reg path is %ws\n",RegistryPath->Buffer);)

     RtlZeroMemory( ParamTable, sizeof(ParamTable) );

     //
     //  change to the parmeters key
     //

     ParamTable[0].QueryRoutine = NULL;
     ParamTable[0].Flags = RTL_QUERY_REGISTRY_SUBKEY;
     ParamTable[0].Name = Parameters;

     //
     //  change to the linkage key
     //

     ParamTable[1].QueryRoutine = NULL;
     ParamTable[1].Flags = RTL_QUERY_REGISTRY_SUBKEY;
     ParamTable[1].Name = Linkage;


     //
```

```
      //  Get the name of the mac driver we should bind to
      //

      ParamTable[2].QueryRoutine = RAWETHERQueryRegistryRoutine;
      ParamTable[2].Flags = RTL_QUERY_REGISTRY_REQUIRED |

RTL_QUERY_REGISTRY_NOEXPAND;

      ParamTable[2].Name = Bind;
      ParamTable[2].EntryContext = (PVOID)MacDriverName;
      ParamTable[2].DefaultType = REG_MULTI_SZ;

      //
      //  Get the name that we should use for the driver object
      //

      ParamTable[3].QueryRoutine = RAWETHERQueryRegistryRoutine;
      ParamTable[3].Flags = RTL_QUERY_REGISTRY_REQUIRED |

RTL_QUERY_REGISTRY_NOEXPAND;

      ParamTable[3].Name = Export;
      ParamTable[3].EntryContext = (PVOID)RAWETHERDriverName;
      ParamTable[3].DefaultType = REG_MULTI_SZ;

      Status=RtlQueryRegistryValues(
                           RTL_REGISTRY_ABSOLUTE,
                           Path,
                           ParamTable,
                           NULL,
                           NULL
                           );

      ExFreePool(Path);

      return Status;
```

Appendix B Win32 Packet Dump Application

B.1 IOCTL.h

```
#include <ntddndis.h>
#include <devioctl.h>

#include <WinDef.h>
#include “PCAEnet.h”

/*  Transport Driver Name
-------------------------------- */
#define RAWETHER_TRANSPORT_NAME “WinMIP”

/* Function Codes For WinMIP Protocol (Transport) Driver
------------------------------------------------------------
 * Passed in AL register on DeviceioControl call.
 * Macros and public IOCTL device codes are defined in WINIOCTL.H.
 */
#define IOCTL_RAWETHER_BASEFILE_DEVICE_TRANSPORT

#define IOCTL_PROTOCOL_SET_OID\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0 , METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROTOCOL_QUERY_OID\
     CTL_CODE(IOCTL_RAWETHER_BASE, 1 , METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROTOCOL_RESET\
     CTL_CODE(IOCTL_RAWETHER_BASE, 2 , METHOD_BUFFERED, FILE_ANY_ACCESS)

#define IOCTL_RAWETHER_START_RECEPTION\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_RAWETHER_STOP_RECEPTION\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_RAWETHER_RELEASE_PACKET\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0802, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_RAWETHER_QUERY_LOWER_INFO\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0803, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_RAWETHER_BUFFER_SEND\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0804, METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_RAWETHER_PACKET_READ\
     CTL_CODE(IOCTL_RAWETHER_BASE, 0x0805, METHOD_OUT_DIRECT, FILE_ANY_ACCESS)

#pragma pack(push,2)

typedef
struct _PACKET_OID_DATA
{
     ULONG           Oid;
     ULONG           Length;
     UCHAR           Data[1];
}
     PACKET_OID_DATA, *PPACKET_OID_DATA;


typedef
struct _USER_PACKET_DATA
{
     LARGE_INTEGERnPacketNumber;
     DWORD                        nPacketTime;                          //
System Time (milliseconds)
```

```
     UINT                          nSelectedMedium;            // NDIS_MEDIUM
Value
     UINT                          nPacketDataLength;
     UCHAR                         PacketBuffer[ MAX_ETHER_SIZE + 16 ];// + Safety
Pad
}
     USER_PACKET_DATA, *PUSER_PACKET_DATA;

#define MAX_ADAPTER_NAME64

typedef
struct _LOWER_INFO
{
     /* Adapter Address Fields
     ------------------------- */
     BYTE                                szAdapterAddress[ ETHER_ADDR_LENGTH ];
     CHAR                                szAdapterName[ MAX_ADAPTER_NAME ];
     UINT                                nSelectedMedium;     // NDIS_MEDIUM
Value

     /* Driver Options
     ----------------- */
     BOOL                                bNoLoopback;

     /* Packet Statistics
     -------------------- */
     LARGE_INTEGERnLastPacketNumber;
     LARGE_INTEGERnTossedPacketCount;// Because Of Lower-Edge Resources...
}
     LOWER_INFO, *PLOWER_INFO;

#pragma pack(pop)

#endif // __IOCTL_H__
```

B.2 WinDump.c

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <WindowsX.h>
#include<WinReg.H>
#include<RegStr.H>

#include <Assert.h>

#include<WinIOCTL.h>
#include “IOCTL.h”


// To Build: nmake /A

/* Static Data
-------------- */
HANDLE    hDevice = INVALID_HANDLE_VALUE;// File handle of the VxD
BOOL              bShutdown = FALSE;                          // Set by Ctrl-C
Handler

DWORD             g_nReceivedPacketCountCount = 0;
DWORD             g_nSequenceErrorCount = 0;
BOOL              g_bQuietDisplay = FALSE;

DWORD             nLastSendTime = 0;
#define   SEND_TEST_INTERVAL15                    // Seconds

/* Extract From NDIS.H
---------------------- */
typedef
enum _NDIS_MEDIUM
{
    NdisMedium802_3,
    NdisMedium802_5,
    NdisMediumFddi,
    NdisMediumWan,
    NdisMediumLocalTalk,
    NdisMediumDix,              // defined for convenience, not a real medium
    NdisMediumArcnetRaw,
    NdisMediumArcnet878_2
}
     NDIS_MEDIUM, *PNDIS_MEDIUM;

VS_FIXEDFILEINFOg_AppVersionInfo;
char                              g_szAppVersionString[ 16 ];

#define MAX_LINK_NAME_LENGTH   64
```

```
//////////////////////////////////////////////////////////////////////////
//// HexDump
//
// Purpose
// Dump a HEX/ASCII representation of a buffer to the console display.
int HexDump( long start, PUCHAR buf, int len )
{
     int i;

     /* Sanity Check
     --------------- */
     assert( buf );

     if( !buf )
     {
          return( 0 );
     }

     assert( len );

     if( len == 0L )
     {
          return( 0 );
     }

     while( len > 0L )
     {
          printf( "%6.6X: ", start );

          /* Print The HEX Representations
          ------------------------------- */
          for( i = 0; i < 8; ++i )
          {
                  if( len - i > 0 )
                  {
                          printf( " %2.2X", buf[ i ] & 0xFF );
                  }
                  else
                  {
                          printf( "   " );
                  }
          }

          printf( " :" );

          for( i = 8; i < 16; ++i )
          {
                  if( len - i > 0 )
                  {
                          printf( " %2.2X", buf[ i ] & 0xFF );
                  }
                  else
                  {
                          printf( "   " );
                  }
          }


          /* Print The ASCII Representations
          ---------------------------------- */
```

```
            printf( “    “ );

            for( i = 0; i < 16; ++i )
            {
                    if( len - i > 0 )
                    {
                            if( isprint( buf[ i ] ) )
                            {
                                    printf( “%c”, buf[ i ] );
                            }
                            else
                            {
                                    printf( “%c”, ‘.’ );
                            }
                    }
                    else
                    {
                            printf( “%c”, ‘.’ );
                    }
            }


            printf( “\n” );

            len -= 16;
            start += 16L;
            buf += 16;
      }

      return( 0 );
}
```

```
////////////////////////////////////////////////////////////////////////
//// OnPacketReceivedAPC
//
// Purpose
// Asynchronous Procedure Call (APC) called by the WinMIP protocol driver
// to pass a received packet to the application.
//
// Parameters
//
// Return Value
//
// Remarks
//

DWORD WINAPI OnPacketReceivedAPC( PUSER_PACKET_DATA pRAWUserPacketData )
{
     BOOL bSequenceError;
     static DWORD nLastSequenceNo = 0;

     assert( pRAWUserPacketData );

     if( !pRAWUserPacketData )
     {
          return( 0 );
     }

     assert( hDevice != INVALID_HANDLE_VALUE );

     if( hDevice == INVALID_HANDLE_VALUE )
     {
          return( 0 );
     }

     if( bShutdown )
     {
          return( 0 );
     }

     /* Update Statistics
     -------------------- */
     ++g_nReceivedPacketCountCount;

     bSequenceError = FALSE;

     /* Check Packet Sequence Number
     ------------------------------
      * A sequence number error means that the driver received a
      * packet which it could not pass to the application because no read
      * was posted.
      *
      * Large packet sequence errors can be expected when dumping packets
      * to the console.
      */
     if( nLastSequenceNo )
     {
          if( ++nLastSequenceNo != pRAWUserPacketData->nPacketNumber.u.LowPart )
          {
                  bSequenceError = TRUE;
                  ++g_nSequenceErrorCount;
          }
```

```
}

nLastSequenceNo = pRAWUserPacketData->nPacketNumber.u.LowPart;

if( g_bQuietDisplay )
{
     if( bSequenceError )
     {
             printf( "*" );
     }
     else
     {
             printf( "." );
     }

     return( 0 );
}

/* Display Packet Header
------------------------ */
printf( "Packet No.: 0x%8.8X%8.8X   ",
     pRAWUserPacketData->nPacketNumber.u.HighPart,
     pRAWUserPacketData->nPacketNumber.u.LowPart
     );

printf( "Time: 0x%8.8X   ", pRAWUserPacketData->nPacketTime );

printf( "Length: %d", pRAWUserPacketData->nPacketDataLength );

/* Display Packet Sequence Error Indicator
-----------------------------------------
 * If it is not the expected sequence number, append "* LOST PACKET *"
 * to the end of the line.
 */
if( bSequenceError )
{
     printf( " * LOST PACKET *" );
}

printf( "\n" );

/* Display Adapter Address
------------------------- */
switch( pRAWUserPacketData->nSelectedMedium )
{
     case NdisMediumLocalTalk:
             printf( "Destination Node: %2.2X   ",
                     (char * )pRAWUserPacketData->PacketBuffer[ 0 ]
                     );

             printf( "Source Node: %2.2X\n",
                     (char * )pRAWUserPacketData->PacketBuffer[ 1 ]
                     );
             break;

     case NdisMedium802_5:
     case NdisMedium802_3:
     default:
             printf( "Destination: %2.2X.%2.2X.%2.2X.%2.2X.%2.2X.%2.2X   ",
```

```
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 0
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 1
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 2
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 3
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 4
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MDstAddr + 5
    ]
                        );

                printf( "Source: %2.2X.%2.2X.%2.2X.%2.2X.%2.2X.%2.2X\n",
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 0
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 1
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 2
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 3
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 4
    ],
                        (char * )pRAWUserPacketData->PacketBuffer[ MSrcAddr + 5
    ]
                        );
                break;
        }

        /* Dump The Packet To The Console
        ------------------------------- */
        HexDump( 0,
             pRAWUserPacketData->PacketBuffer,
             pRAWUserPacketData->nPacketDataLength
             );

        printf( "\n" );

        return 0;
    }
```

```
//////////////////////////////////////////////////////////////////////////
//// CtrlHandler
//
// Purpose
// Console Ctrl-C handler.
//
// Parameters
//
// Return Value
//
// Remarks
// The Ctrl-C handler stops packet reception by calling the driver
// via the Win32 DeviceIoControl mechanism. The driver blocks the calling
// thread until all queued APC calls have been processed. Note that
// while in the process of stopping, the OnPacketReceivedAPC function
// will continue to be called.
//

BOOL WINAPI CtrlHandler( DWORD dwCtrlType )
{
     DWORD                nBytesReturned;// Req'd for DeviceIOControl call
     OVERLAPPEDOverLapped;
     BOOL                 bResult;

     /* Make Sure Driver Has Been Opened
     ---------------------------------- */
     assert( hDevice != INVALID_HANDLE_VALUE );

     if( hDevice == INVALID_HANDLE_VALUE )
     {
          return( FALSE );
     }

     switch( dwCtrlType )
     {
          case CTRL_C_EVENT:
          case CTRL_BREAK_EVENT:
          case CTRL_CLOSE_EVENT:
          case CTRL_SHUTDOWN_EVENT:
          default:
                  bShutdown = TRUE;

                  printf( "\nStopping Packet Reception\n" );

                  printf( "Received Packet Count: %d\n",
g_nReceivedPacketCountCount );

                  if( g_nSequenceErrorCount )
                  {
                          printf( "Sequence Error Count : %d\n",
g_nSequenceErrorCount );
                  }

                  /* Create The OVERLAPPED Event To Wait On
                  ---------------------------------------- */
                  OverLapped.hEvent = CreateEvent(
                                                                  NULL,//
Security Attributes
                                                                  FALSE,//
Auto-Reset
```

```
                                                                    FALSE,//
Initial State Signaled
                                                                    NULL//
Event-obkect Name
                                                                    );

                ResetEvent( OverLapped.hEvent );

                bResult = DeviceIoControl(
                                                hDevice,

IOCTL_RAWETHER_STOP_RECEPTION,
                                                NULL, 0,
                                                NULL, 0,
                                                &nBytesReturned,
                                                &OverLapped          //
REQUIRED For FILE_FLAG_OVERLAPPED
                                                );

                if( !bResult )
                {
                        bResult = GetOverlappedResult(
                                                        hDevice,
                                                        &OverLapped,
                                                        &nBytesReturned,
                                                        TRUE       // Wait
For Completion
                                                        );
                }

                CloseHandle( OverLapped.hEvent );

                return( FALSE );// Let default handler do it's job!
     }

     return( FALSE );
}

char *NAMETBLE[] =
{
     "802.3",
     "802.5",
     "Fddi",
     "Wan",
     "LocalTalk",
     "Dix",
     "Arcnet (Raw)",
     "Arcnet (878.2)",
};
```

```
//////////////////////////////////////////////////////////////////////////
//// DisplayAdapterInformation
//
// Purpose
// Display adapter information returned from IOCTL_RAWETHER_QUERY_LOWER_INFO
// call to the protocol driver.
//

VOID DisplayAdapterInformation( VOID )
{
     LOWER_INFOLowerInfo;
     DWORD               nBytesReturned;// Req'd for DeviceIOControl call
     OVERLAPPEDOverLapped;
     BOOL                bResult;

     /* Make Sure Driver Has Been Opened
     ---------------------------------- */
     assert( hDevice != INVALID_HANDLE_VALUE );

     if( hDevice == INVALID_HANDLE_VALUE )
     {
          return;
     }

     printf( "Adapter Information:\n\n" );

     nBytesReturned = 0;

     /* Create The OVERLAPPED Event To Wait On
     ---------------------------------------- */
     OverLapped.hEvent = CreateEvent(
                                                                  NULL,//
Security Attributes
                                                                  FALSE,//
Auto-Reset
                                                                  FALSE,//
Initial State Signaled
                                                                  NULL//
Event-obkect Name
                                                                  );

     ResetEvent( OverLapped.hEvent );

     bResult = DeviceIoControl(
                                                  hDevice,

IOCTL_RAWETHER_QUERY_LOWER_INFO,
                                                  &LowerInfo, sizeof(
LOWER_INFO ),// Input Data (i.e., To Driver)
                                                  &LowerInfo, sizeof(
LOWER_INFO ),// Output Data (i.e., From Driver)
                                                  &nBytesReturned,
                                                  &OverLapped          //
REQUIRED For FILE_FLAG_OVERLAPPED
                                                  );

     if( !bResult )
     {
          bResult = GetOverlappedResult(
                                        hDevice,
```

```
                                        &OverLapped,
                                        &nBytesReturned,
                                        TRUE                // Wait For
Completion
                                        );
     }

     CloseHandle( OverLapped.hEvent );

     if( !bResult )
     {
     fprintf(stderr, "Failed to get lower info\n");

          return;
     }

     /* Display The "Adapter Name"
     ----------------------------
      * The "AdapterName" is really the name of the adapter key under
      * HKLM\System\CurrentControlSet\Services\Class\Net. A typical
      * "AdapterName" would be "0002".
      */
     printf( "Adapter Name   : %s\n", LowerInfo.szAdapterName );


     /* Display The Adapter Medium
     ---------------------------- */
     if( NdisMedium802_3 <= LowerInfo.nSelectedMedium
          && LowerInfo.nSelectedMedium < NdisMediumArcnet878_2 )
     {
          printf( "Adapter Medium : %s\n", NAMETBLE[ LowerInfo.nSelectedMedium ]
);
     }

     /* Display Adapter Address
     ------------------------- */
     switch( LowerInfo.nSelectedMedium )
     {
          case NdisMediumLocalTalk:
                  printf( "Adapter Address: %2.2X\n",
                          (char * )LowerInfo.szAdapterAddress[ 5 ]
                          );
                  break;

          case NdisMedium802_5:
          case NdisMedium802_3:
          default:
                  printf( "Adapter Address:
%2.2X.%2.2X.%2.2X.%2.2X.%2.2X.%2.2X\n",
                          (char * )LowerInfo.szAdapterAddress[ 0 ],
                          (char * )LowerInfo.szAdapterAddress[ 1 ],
                          (char * )LowerInfo.szAdapterAddress[ 2 ],
                          (char * )LowerInfo.szAdapterAddress[ 3 ],
                          (char * )LowerInfo.szAdapterAddress[ 4 ],
                          (char * )LowerInfo.szAdapterAddress[ 5 ]
                          );
                  break;
     }
}
```

```
//////////////////////////////////////////////////////////////////////////
//// SendTestPacket
//
// Purpose
// Send a test packet using the WinMIP VxD services.
//
// Parameters
// None
//
// Return Value
// None
//
// Remarks
//

VOID SendTestPacket( VOID )
{
     PUCHAR       pSendData;
     DWORD               nBytesReturned;// Req'd for DeviceIOControl call
     BOOL                bResult;
     OVERLAPPEDOverLapped;
     DWORD               nWaitResult;

     printf( "Sending Test Packet..." );

     /* Make Sure Driver Has Been Opened
     ---------------------------------- */
     assert( hDevice != INVALID_HANDLE_VALUE );

     if( hDevice == INVALID_HANDLE_VALUE )
     {
          printf( "FAILED (1)\n" );
          return;
     }

     /* Allocate Data Buffer
     ---------------------- */
     pSendData = (PUCHAR )malloc( MAX_ETHER_SIZE );

     assert( pSendData );

     if( !pSendData )
     {
          printf( "FAILED (2)\n" );
          return;
     }

     /* Create The OVERLAPPED Event To Wait On
     ---------------------------------------- */
     OverLapped.hEvent = CreateEvent(
                                                                    NULL,//
Security Attributes
                                                                    FALSE,//
Auto-Reset
                                                                    FALSE,//
Initial State Signaled
                                                                    NULL//
Event-object Name
                                                                    );
```

```
      assert( OverLapped.hEvent );

      if( !OverLapped.hEvent )
      {
            printf( “FAILED (3)\n” );
            free( pSendData );
            return;
      }

      if( !ResetEvent( OverLapped.hEvent ) )
      {
            printf( “FAILED (4)\n” );
            CloseHandle( OverLapped.hEvent );
            free( pSendData );
            return;
      }

      /* Fill The Data Buffer
      ---------------------- */
      memset( pSendData, ‘A’, MAX_ETHER_SIZE );// For Visibility On Analyzer

      // ATTENTION!!! This should be media-dependent!!!
    pSendData[0] = 0xFF;
    pSendData[1] = 0xFF;
    pSendData[2] = 0xFF;
    pSendData[3] = 0xFF;
    pSendData[4] = 0xFF;
    pSendData[5] = 0xFF;

      /* Send The Data Buffer
      ---------------------- */
      bResult = DeviceIoControl(
                                  hDevice,
                                  IOCTL_RAWETHER_BUFFER_SEND,
                                  pSendData, 64,                          // Input
Data (i.e., To Driver)
                                  NULL, 0,                                //
Output Data (i.e., From Driver)
                                  &nBytesReturned,
                                  &OverLapped       // REQUIRED For
FILE_FLAG_OVERLAPPED
                                  );

//    assert( bResult );

      if( bResult )
      {
            nWaitResult = WaitForSingleObject( OverLapped.hEvent, 10 );

            if( !nWaitResult )
            {
                  printf( “SENT\n” );
            }
            else
            {
                  printf( “FAILED (5)\n” );
            }
      }
      else
      {
```

```
            printf( “FAILED (6)\n” );
      }

      /* Free Resources
      ----------------- */
      CloseHandle( OverLapped.hEvent );

      free( pSendData );
}

#define NUM_PACKET_PACKS48


typedef
struct _PacketPack
{
      USER_PACKET_DATAUserPacketData;
      DWORD                                 nBytesReturned;
      OVERLAPPED           OverLapped;
}
      PacketPack, *PPacketPack;

////////////////////////////////////////////////////////////////////////////
//// DestroyPackages
//
// Purpose
// Close event handles and free memory associated with packet packages
// created previously by CreatePackages.
//

void DestroyPackages( PPacketPack pPackageBase, DWORD nPackageCount )
{
      int                                   i;
      PPacketPack          pPackage;

      /* Sanity Checks
      ---------------- */
      assert( pPackageBase );

      if( !pPackageBase )
      {
            return;
      }

      /* Close Event Handles For Each Package
      ------------------------------------- */
      for( i = 0; i < NUM_PACKET_PACKS; i++ )
      {
            pPackage = &pPackageBase[ i ];

            if( pPackage->OverLapped.hEvent )
            {
                  CloseHandle( pPackage->OverLapped.hEvent );
            }
      }

      /* Free Memory Allocated For The Packages
      -------------------------------------- */
      free( pPackageBase );
}
```

```
//////////////////////////////////////////////////////////////////////////
//// CreatePackages
//
// Purpose
// Allocate memory and create event handles for the specified number of
// packet “packages”.
//
// Parameters
//   nPackageCount - The number of packet packages to create.
//
// Return Value
//   If successfull, returns pointer to an array of packet packages.
//
// Remarks
// WinMIP uses multiple concurrent asynchronous IOCTL_RAWETHER_PACKET_READ
// calls as a means to reduce packet loss.
//
// Each packet “package” is a data structure (defined elsewhere) which
// contains a USER_PACKET_DATA, OVERLAPPED and other fields needed to
// make one asynchronous IOCTL_RAWETHER_PACKET_READ call to the RAWETHER
// driver.
//
// This function simply allocates and initializes multiple packet packages.
//

PPacketPack CreatePackages( DWORD nPackageCount )
{
     int                                  i;
     PPacketPack          pPackage, pPackageBase;

     /* Allocate Memory For The Packages
     ---------------------------------- */
     pPackageBase = (PPacketPack )malloc( sizeof( PacketPack ) * nPackageCount );

     assert( pPackageBase );

     if( !pPackageBase )
     {
          return( NULL );
     }

     /* Zero The Package Memory
     ------------------------- */
     for( i = 0; i < NUM_PACKET_PACKS; i++ )
     {
          pPackage = &pPackageBase[ i ];

          memset( pPackage, 0x00, sizeof( PacketPack ) );
     }

     /* Create Event Handles For Each Package
     --------------------------------------- */
     for( i = 0; i < NUM_PACKET_PACKS; i++ )
     {
          pPackage = &pPackageBase[ i ];

          /* Create The OVERLAPPED Event To Wait On
          ---------------------------------------- */
          pPackage->OverLapped.hEvent = CreateEvent(
```

```
                                                                    NULL,//
Security Attributes
                                                                    FALSE,//
Auto-Reset
                                                                    FALSE,//
Initial State Signaled
                                                                    NULL//
Event-obkect Name
                                                                    );

          /* Verify That The Event Was Created
          ----------------------------------- */
          assert( pPackage->OverLapped.hEvent );

          if( !pPackage->OverLapped.hEvent )
          {
                  /* Destroy Partially Allocated Packages
                  ------------------------------------ */
                  DestroyPackages( pPackageBase, nPackageCount );

                  return( NULL );
          }
     }

     return( pPackageBase );
}

HANDLEPackageHandles[ NUM_PACKET_PACKS ];

VOID GetAppVersion( void )
{
     char *pszFileName = NULL;

     wsprintf( g_szAppVersionString, “Unknown” );

     pszFileName = malloc( MAX_PATH );

     if( !pszFileName )
     {
          return;
     }

     if( GetModuleFileName( NULL, pszFileName, MAX_PATH) > 0 )
     {
          DWORD dwType;
          DWORD dwDataLen = GetFileVersionInfoSize( pszFileName, &dwType );

          void *pszBuffer = malloc(dwDataLen + 1);

          if (pszBuffer == NULL)
                  return;

          if (GetFileVersionInfo(pszFileName, 0L, dwDataLen, pszBuffer) == TRUE)
          {
                  VS_FIXEDFILEINFO *pFileInfo;

                  if (VerQueryValue(pszBuffer, “\\”,
                                                                  (void
**)&pFileInfo, (UINT *)&dwDataLen) == TRUE)
                  {
```

```
                        memcpy( &g_AppVersionInfo, pFileInfo, sizeof(
VS_FIXEDFILEINFO ) );

                        wsprintf( g_szAppVersionString, “%d.%2.2d.%2.2d.%2.2d”,
                                HIWORD( g_AppVersionInfo.dwFileVersionMS ),
                                LOWORD( g_AppVersionInfo.dwFileVersionMS ),
                                HIWORD( g_AppVersionInfo.dwFileVersionLS ),
                                LOWORD( g_AppVersionInfo.dwFileVersionLS )
                                );
                }
          }
          free(pszBuffer);
     }

     free( pszFileName );
}
```

```
////////////////////////////////////////////////////////////////////////
//// main
//
// Purpose
// Console application MAIN entry point.
//

int main( int argc, char **argv )
{
     DWORD                              nBytesReturned;// Req'd for
DeviceIOControl call
     OVERLAPPED          OverLapped;
     DWORD                              nSleepResult, nWaitResult;
     char                               *pDriverName;
     BOOL                               bResult;
     TCHAR                              szSymbolicLink[MAX_LINK_NAME_LENGTH];
     int                                i, nNextPackage;
     PPacketPack         pPackage, pPackageBase = NULL;

     GetAppVersion();

     printf( "Ndis 3.0 Windows NT Network Packet Dump Utility ");

     if( argc > 1 )
     {
          g_bQuietDisplay = TRUE;
     }

     pDriverName = RAWETHER_TRANSPORT_NAME;

     /* Open the WinMIP Protocol Driver VxD
     ---------------------------------------- */
     hDevice = INVALID_HANDLE_VALUE;

     wsprintf(
          szSymbolicLink,               // szSymbolicLink Used As Scratch
Buffer...
          TEXT("\\Device\\%s"),
          RAWETHER_TRANSPORT_NAME
          );

     bResult=DefineDosDevice(
                         DDD_RAW_TARGET_PATH,
                         RAWETHER_TRANSPORT_NAME,
                         szSymbolicLink
                         );

     if( bResult )
     {
     }

     wsprintf(
          szSymbolicLink,
          TEXT("\\\\.\\%s"),
          RAWETHER_TRANSPORT_NAME
          );

     //
     // Note that the file is created with the FILE_FLAG_OVERLAPPED flag
     // set. This allows asynchronous operations to be performed in the
```

```
      // file handle.
      //
      // In addition, opening a handle for overlapped I/O imposes the
      // requirement that a pointer to a valid OVERLAPPED structure MUST
      // be passed to all calls to DeviceIoControl. Otherwise, unpredictable
      // errors can result.
      //
      hDevice = CreateFile(
                                    szSymbolicLink,
                                    0,0,NULL,
//                                  CREATE_ALWAYS,
                                    OPEN_EXISTING,
//                                  FILE_FLAG_DELETE_ON_CLOSE |
FILE_FLAG_OVERLAPPED,
                                    FILE_FLAG_OVERLAPPED,
                                    0
                                    );

      if( hDevice == INVALID_HANDLE_VALUE )
      {
       fprintf(stderr, “Cannot Load %s Protocol Driver Error=%08lx\n”,
                   pDriverName,
                   GetLastError()
                   );

           exit( 1 );
      }

      /* Display Adapter Information
      ------------------------------ */
      DisplayAdapterInformation();

      printf( “\nPress ENTER To Start...\n” );

      while( !kbhit() )
      {
           Sleep( 0 );
      }

      getchar();           // Eat ENTER Character


      /* Create Packet Packages
      ------------------------ */
      pPackageBase = CreatePackages( NUM_PACKET_PACKS );

      assert( pPackageBase );

      if( !pPackageBase )
      {
           printf( “Could Not Create Read Packages\n” );

           exit( 2 );
      }

      /* Build Handle Array For Call To WaitForMultipleObjects
      -------------------------------------------------------- */
      for( i = 0; i < NUM_PACKET_PACKS; i++ )
      {
           pPackage = &pPackageBase[ i ];
```

```
            PackageHandles[ i ] = pPackage->OverLapped.hEvent;
      }

      /* Set The Ctrl-C Handler
      ------------------------ */
      SetConsoleCtrlHandler( CtrlHandler, TRUE );

      printf( “Press Ctrl-C To Exit...\n”);

      /* Create The OVERLAPPED Event To Wait On
      ---------------------------------------- */
      OverLapped.hEvent = CreateEvent(
                                                                  NULL,//
Security Attributes
                                                                  FALSE,//
Auto-Reset
                                                                  FALSE,//
Initial State Signaled
                                                                  NULL//
Event-obkect Name
                                                                  );

      ResetEvent( OverLapped.hEvent );

      /* Call Driver To Start Reception
      -------------------------------- */
      bResult = DeviceIoControl(
                                 hDevice,
                                 IOCTL_RAWETHER_START_RECEPTION,
                                 NULL, 0,
                                 NULL, 0,
                                 &nBytesReturned,
                                 &OverLapped       // REQUIRED For
FILE_FLAG_OVERLAPPED
                                 );

      if( !bResult )
      {
            bResult = GetOverlappedResult(
                                       hDevice,
                                       &OverLapped,
                                       &nBytesReturned,
                                       TRUE                 // Wait For
Completion
                                       );
      }

      CloseHandle( OverLapped.hEvent );

      if( !bResult )
      {
      fprintf(stderr, “Start Reception Failed\n”);

            DestroyPackages( pPackageBase, NUM_PACKET_PACKS );

            exit( 3 );
      }

      bShutdown = FALSE;
```

```
	/* Start All The Packet Reads
	---------------------------- */
	for( i = 0; i < NUM_PACKET_PACKS; i++ )
	{
		pPackage = &pPackageBase[ i ];

		/* Reset The Package Event
		------------------------- */
		if( !ResetEvent( pPackage->OverLapped.hEvent ) )
		{
			printf( “FAILED (4)\n” );
			break;
		}

		/* Post A Packet Read
		--------------------- */
		bResult = DeviceIoControl(
					hDevice,
					IOCTL_RAWETHER_PACKET_READ,
					NULL, 0,			// Input
Data (i.e., To Driver)
					&pPackage->UserPacketData,
					sizeof( USER_PACKET_DATA ),// Output
Data (i.e., From Driver)
					&pPackage->nBytesReturned,
					&pPackage->OverLapped
					);
	}

	nNextPackage = 0;// Index Of Next Package To Read On

	/* Loop Until Shutdown
	---------------------- */
	while( !bShutdown )
	{
		/* Wait For One Or More Packet Reads To Complete
		----------------------------------------------
		 * To exit this application the user presses the Ctrl-C key. This
		 * causes the CtrlHandler to be called. The CtrlHandler simply sets
		 * the bShutdown flag and returns. The bShutdown flag must be noticed
		 * in this while loop in order to exit the application.
		 *
		 * If no packets were being received, then the WaitForMultipleObjects
		 * would not expire and the application would never exit.
		 */
		nWaitResult = WaitForMultipleObjects(
							NUM_PACKET_PACKS,
							PackageHandles,
							FALSE,
							500	//
.5-sec.
							);

		if( bShutdown )
		{
			break;
		}

		/* Handle Based On Wait Result
		------------------------------ */
```

```
            if( WAIT_OBJECT_0 <= nWaitResult &&
                            nWaitResult <= WAIT_OBJECT_0 + NUM_PACKET_PACKS - 1
                    )
            {
                    /* Notes
                    --------
                     * Entry into this case means that one (or more) reads completed
                     * successfully.
                     *
                     * The variable i, calculated below, identifies the lowest
package
                     * index whose read operation caused WaitForMultipleObjects to
                     * return. This value MUST BE IGNORED!!!
                     *
                     * In order to insure that packets are received sequentially,
the
                     * nNextPackage parameter identifies the index number of the
next
                     * packet package that we “expect” to read on. A call is made
                     * to GetOverlappedResult to confirm that the expected packet
                     * package can, in fact be read on. If it can’t, then wait until
                     * the expected packet package is actually ready.
                     *
                     * This scheme works because the NDISHOOK driver handles reads
                     * in a FIFO fashion.
                     *
                     * The reason that the variable i MUST BE IGNORED can be
illustrated
                     * at package array rollover. Consider the case when the there
are
                     * 64 packages numbered 0 - 63 and package 62 has been
processed.
                     * If two packets are received quickly, then packages 63 and 0
should
                     * be processed in succession when WaitForMultipleObjects
returns.
                     * In this case, the nNextPackage parameter would correctly
indicate
                     * that package 63 should be handled next. However, the
parameter i
                     * would indicate that package 0 should be next - which is
INCORRECT.
                     */
                    i = nWaitResult - WAIT_OBJECT_0;// ATTENTION!!! Ignore This
Value!!!

                    pPackage = &pPackageBase[ nNextPackage ];// Pointer To Expected
Package

                    /* Sequentially Read On Expected Package Sequence
                    ------------------------------------------------
                     * When WaitForMultipleObjects returns, one OR MORE reads have
                     * completed successfully. The following loop reads on packet
                     * packages sequentially until one is encountered which still
                     * has I/O pending.
                     */
                    do
                    {
                            /* Handle The Received Packet
                            ---------------------------- */
```

```
                        OnPacketReceivedAPC( &pPackage->UserPacketData );

                        /* Post Another Read On The Packet Package
                        ---------------------------------------- */
                        if( !bShutdown )
                        {
                                /* Reset The Package Event
                                ------------------------ */
                                if( !ResetEvent( pPackage->OverLapped.hEvent ) )
                                {
                                        printf( “FAILED (4)\n” );
                                        break;
                                }

                                /* Post The Read
                                --------------- */
                                bResult = DeviceIoControl(
                                                                        hDevice,
IOCTL_RAWETHER_PACKET_READ,
                                                                        NULL, 0,
// Input Data (i.e., To Driver)
&pPackage->UserPacketData,
                                                                        sizeof(
USER_PACKET_DATA ),// Output Data (i.e., From Driver)
&pPackage->nBytesReturned,
&pPackage->OverLapped
                                                                        );
                        }

                        /* Move To The Next Sequential Packet Package
                        ------------------------------------------- */
                        if( ++nNextPackage >= NUM_PACKET_PACKS )
                        {
                                nNextPackage = 0;  // Wrap-around
                        }

                        pPackage = &pPackageBase[ nNextPackage ];// Pointer To
Expected Package
                }
                while( GetOverlappedResult(
                                                                hDevice,
&pPackage->OverLapped,
&pPackage->nBytesReturned,
                                                                FALSE
                                                                )
                                        );
        }
        else if( WAIT_ABANDONED_0 <= nWaitResult &&
                        nWaitResult <= WAIT_ABANDONED_0 + NUM_PACKET_PACKS - 1
                )
        {
                /* Notes
                --------
                 * Entry into this case means that one (or more) reads were
```

```
                     * abandoned.
                     *
                     * The parameter i, calculated below, identifies which read
                     * operation caused WaitForMultipleObjects to return.
                     */
                    i = nWaitResult - WAIT_ABANDONED_0;

                    break;
            }
            else if( nWaitResult == WAIT_TIMEOUT )
            {
                    if( bShutdown )
                    {
                            break;
                    }
            }
            else
            {
                    break;
            }

#ifdef ZNEVER
            /* Sleep In An Alertable State
            ------------------------------ */
            nSleepResult = SleepEx( SEND_TEST_INTERVAL * 1000, TRUE );

            /* Send A Packet Occasionally
            ------------------------------ */
            if( GetTickCount()
                            > nLastSendTime + ( SEND_TEST_INTERVAL * 1000 )
                    )
            {
                    SendTestPacket();
                    nLastSendTime = GetTickCount();
            }
#endif
      }

      /* Destroy The Packet Packages
      ------------------------------ */
      DestroyPackages( pPackageBase, NUM_PACKET_PACKS );

      return( 0 ); // No Complaints
}
```

COS/CCS 2008-17

www.kth.se