

LAYERED DRIVER ARCHITECTURE FOR
NETWORK CONNECTIVITY OVER A POLICE RADIO

BY

ANIL K. CHINTALAPATI

Master of Science, University of New Hampshire, 2002

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Electrical and Computer Engineering

December, 2002

This thesis has been examined and approved

Thesis Director, Dr. W.Thomas Miller, Ph.D.,
Professor of Electrical Engineering

Dr. William H.Lenharth, Ph.D.,
Assistant Reseach Professor of Electrical
Engineering

Dr. Andrew Kun, Ph.D.,
Assistant Professor of Electrical Engineering

This Thesis is dedicated to my parents

ACKNOWLEDGMENTS

I would like first to thank Professor Dr. Tom Miller for his guidance, advice and support during my research and writing of this thesis. Knowledge and experience that I have gained working under his supervision will be of great value for me for all my life.

Also, thanks to Dr. Andrew L. Kun and to Dr. William H. Lenharth for being members of my committee, and for their advisory and feedback.

This project was sponsored by New Hampshire department of safety and supported by U.S. Department of Justice. I would like to thank them for their support and in funding the project.

I would like to thank my parents for their continued emotional support and advice.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS.....	iv
LIST OF FIGURES.....	vii
ABSTRACT.....	viii
INTRODUCTION.....	1
1. BACK GROUND.....	5
1.1 INTRODUCTION.....	5
1.2 GENERAL OVERVIEW.....	5
1.2.1 Data Service Types.....	6
1.2.2 Types of Data Configurations.....	6
1.3 PACKET DATA SPECIFICATION.....	8
1.3.1 FNE Data mode.....	8
1.3.2 Repeated Data mode.....	10
1.3.3 Direct mode.....	10
1.4 CONTROL SIGNALING.....	11
1.5 ADVANTAGES OF IP CONNECTIVITY OVER POLICE RADIO.....	15
1.6 CASE STUDY OF RADIO IP PRODUCTS.....	15
1.6.1 Compaq’s Radio IP product for Sacramento Police.....	15
1.6.2 Motorola’s Tetra IP.....	16
1.7 PURPOSE OF THE NETWORK DRIVER.....	17
1.8 MINIPOINT DRIVER SPECIFIC.....	18
1.8.1 Definition.....	18
1.8.2 NIC Miniport Types.....	20
1.8.3 NDIS Library.....	20
1.8.4 WDM lower Edge Miniport Driver.....	22
1.9 INTRODUCTION TO IDB DRIVER.....	23
1.10 CAN PROTOCOL STANDARD.....	24
1.11 IDB BUS.....	26
2. NETWORK MINIPOINT DRIVER.....	28
2.1 Miniport Driver Architecture.....	28
2.1.1 Driver Entry.....	28
2.1.2 Miniport Registration.....	29
2.1.3 Miniport Initialization.....	30
2.2 Driver Implementation Details.....	30
2.2.1 Write Operation.....	31
2.2.2 Read Operation.....	37
2.3 Different Implementations of the Network Driver.....	41
3 IDB BUS DRIVER.....	42
3.1 CATLAB Implementation of IDB bus controller.....	42

3.2 Purpose of IDB logical driver.....	44
3.3 Driver Implementation Details.....	44
3.3.1 Write Operation.....	47
3.3.2 Read Operation.....	48
3.3.3 Hardware Settings.....	50
3.4 User Interface to the IDB driver.....	51
4 RESULTS AND CONCLUSION.....	54
4.1 Installation Procedure.....	54
4.2 Results.....	55
4.2.1 Testing the miniport driver.....	55
4.2.2 Testing the IDB driver.....	56
4.2.3 Testing the Records application.....	56
4.3 Future Work.....	56
REFERENCES.....	58
APPENDIX A. Important sections of the driver code.....	59

LIST OF FIGURES

Figure 1-1 Radio to FNE data configuration.....	7
Figure 1-2 Mobile to fixed Host (FNE Data).....	10
Figure 1-3 Information Transfer stack.....	12
Figure 1-4 Control Information stack.....	12
Figure 1-5 MDP/MRC Interface.....	13
Figure 1-6 FNE frame header format.....	14
Figure 1-7 IP connectivity in the CATLAB project.....	18
Figure 1-8 Miniport driver network stack connectivity.....	19
Figure 1-9 Miniport driver's position in the OSI Reference Model.....	20
Figure 1-10 Connectionless Network Driver Environment.....	21
Figure 1-11 Miniport Driver for Serial port.....	23
Figure 1-12 Conceptual schematic of CAN protocol network.....	25
Figure 1-13 ITS Data Bus Architecture.....	27
Figure 2-1 Driver Initialization.....	29
Figure 2-2 Layered driver model.....	31
Figure 2-3 Ethernet Frame format.....	34
Figure 2-4 ARP Packet Frame Format.....	35
Figure 2-5 Flow chart for Miniport send operation.....	36
Figure 2-6 SLIP encoded IP packet received by Miniport driver.....	38
Figure 2-7 IP Packet after SLIP encoding.....	38
Figure 2-8 Ethernet framed IP.....	38
Figure 2-9 Miniport Read operation.....	39
Figure 2-10 Processing of a received SLIP packet by the miniport driver.....	40
Figure 2-11 Driver layering in a laptop computer.....	41
Figure 2-12 Driver layering in the embedded computer connected to IDB bus.....	41
Figure 3-1 IDB frame format.....	43
Figure 3-2 IDB bus components.....	43
Figure 3-3 Layered drivers for IDB data transmission and reception.....	45
Figure 3-4 Several Applications talking to the IDB driver.....	47
Figure 3-5 IDB Write Operation.....	48
Figure 3-6 IDB Read Operation.....	49
Figure 3-7 Kernel Read Thread Operation.....	50

ABSTRACT

DEVELOPMENT OF AN IP-BASED SERIAL PORT NETWORK DRIVER

By

Anil Chintalapati

University of New Hampshire, September 2002

The driving force for the work documented here was the need to run applications which use IP protocols such as UDP, TCP etc., between an embedded computer in a police car and a remote server, connected by a police radio link. This was implemented as a part of the CATLAB project, which is a collaborative research and development effort between the University of New Hampshire and the New Hampshire Department of Safety. The CATLAB project involves integrating embedded mobile computing equipment and wireless networking into New Hampshire State Police cruisers. A network layer miniport driver for the Microsoft Windows 2000 operating system has been layered over a logical driver so as to send the IP packets of the embedded computer connected to the police radio over its serial interface. The IP packets are encapsulated in a SLIP frame, which is a serial port point-to-point link layer protocol. A filter driver called the IDB driver was also implemented. This is a kernel mode filter driver that encapsulates the network data in a frame format specific to an IDB bus controller, which was developed to integrate the in-car electronics. Both the drivers were developed for the Windows 2000 operating system using the Windows Driver Development Kit.

INTRODUCTION

The CATLAB (Consolidated Advanced Technologies For Law Enforcement Program) Project addresses related problems in the integration of electronic devices within police mobile units and in the network communications integrating mobile units and law enforcement agencies, all of which impair the ability to seamlessly collect, interpret and exchange information in real time. The rapid advancement in performance and steady decrease in cost of advanced computing and communications technologies are leading to the increased use of specialized electronic devices and wireless data communications by law enforcement agencies. When deployed as an integrated mobile networking system, these devices promise to significantly enhance the information available to officers in the field. However, the development of specialized devices and software by independent vendors, often without emphasis on systems integration or inter-vendor compatibility, is a serious impediment to the creation of fully integrated information systems within individual law enforcement agencies. Furthermore, the lack of cross-jurisdiction coordination in procurement creates the likelihood of system incompatibilities across jurisdictions, limiting their ability to share information. The Consolidated Advanced Technologies (CAT) Program addresses these issues through the collaborative efforts of the University of New Hampshire and the New Hampshire Department of Safety.

The car based electronics in the CATLAB program, inside the police car were integrated by developing a common interface for the Intelligent Transportation Systems Data Bus that was implemented in each police car. The IDB bus is a low speed data bus

that is used to integrate the electronics in in-vehicular networks. The Control Area Network (CAN) protocol is used as the method of data transmission on the IDB. The IDB provides an efficient means of transporting data and commands from device to device. The IDB interface takes serial data from the peripheral devices in the car and sends it to the IDB interface connected to the embedded computer in a frame format compliant with the CAN protocol. In the same way the serial data from the embedded computer is sent to the IDB interface connected to it, in an implementation specific frame format, from which it reaches the IDB interface connected to the peripheral to which it is destined in the CAN protocol frame format.

The CATLAB project requires IP communication between the embedded computer inside the police car and a remote server at the headquarters connected by a police radio link, to run IP based applications. The Police radio is connected to the embedded computer over the IDB bus interface, which uses the CAN protocol. The CAN protocol is a serial bus standard suited for networking "Intelligent" devices as well as sensors and actuators within a system. Since there is no standard frame format for the link layer of the IDB interface, there are no drivers available off-the-shelf that bind the upper layer protocols such as IP, to the hardware. A network driver was thus implemented that provided a new interface to the IP layer for network communication. The network driver was implemented from the framework of a classic network driver, with all the hardware specific function calls skipped. Since the IDB bus uses serial interface, all the send and receive operations from the network driver are routed to the classic serial port driver. The IDB interface connected to the embedded computer understands only its implementation

specific frame format. In order to serve this purpose a filter driver was layered in between the network driver developed and the serial port driver. This filter driver, called the IDB driver, puts the IP network data in the implementation specific IDB frame format and routes it to the serial port driver. In this way the link layer of the OSI model protocol stack was implemented.

The driver development was done for Windows 2000 operating system using Windows Driver Development Kit (DDK), which is an environment to implement device drivers. A windows kernel mode debugger called WinDbg was used for the work. The work was successful and the embedded computer is now able to do IP transmission over the IDB bus.

Chapter 1 explains the background behind this work. It includes a detailed description of the IP stack over an APCO Project 25 compliant data radio. It then provides an introduction to the basic features of a Windows Miniport network Driver. This is followed by an introduction to the logical driver that frames customized packets for transmission over the IDB bus. It also explains the previous work done in the field of IP connectivity over police radio.

Chapter 2 explains in detail how the miniport network driver was implemented for the CATLAB project. This chapter goes into Windows DDK (Driver Development Kit) specifics on how the problems involving the driver development were solved.

Chapter 3 explains in detail the implementation of the IDB logical bus driver. This chapter is also Windows DDK specific with the software details on how frame transmission was accomplished over the CATLAB IDB bus.

Chapter 4 documents the tests conducted with the drivers assisting communication over the police radio and the results obtained.

Chapter 5 provides suggestions on the future work for the further development of this project.

1. Background

1.1 Introduction

This Chapter explains in detail the work done in the field of IP connectivity over a Police radio and the advantages it provides in the work of a police trooper. It also explains in detail the IP stack in an APCO Project 25 compliant data radio. An introduction to the Windows 2000 miniport network and kernel logical drivers is also given.

Before going into the details of how the IP connectivity was established over the police radio network, it is important to understand the radio network itself in the context of the IP protocol. This chapter gives an insight into the Project 25 police radio interface over which the IP layer was developed for this project.

1.2 General Overview

The Project 25 radio is capable of supporting both circuit and packet data. It is also capable of transporting multiple packet protocols such as TCP/IP, X.25 etc simultaneously. But in order to keep the police radio software to a minimum, the radio generally supports one circuit and one standard packet interface. IP is the standard packet protocol over which other packet protocols are layered. The APCO 25 system defines 2 different types of data services (circuit and packet) in 3 different data configurations, giving rise to totally 6 combinations of service/configuration pairs.

1.2.1 Data Service Types

Circuit Data: The Circuit data service can be provided by the police radio link with a BER $<10^{-6}$. The circuit data service can be invoked from a data peripheral (in this case the embedded computer) connected to the police radio using a TIA 602 standard AT circuit establishment protocol. The protocol standard implements error control and ARQ techniques to ensure the integrity of data transmitted. Quality of service might also be provided under perfect signal conditions.

Packet Data: In Packet data service, packets compliant to IP protocol format can be transmitted over the radio link. The radio also implements error control and ARQ protocols to ensure data integrity. The CATLAB project is currently using this packet data service over the radio link for running its applications. Although TCP applications are logically possible to be run, they might fail because of radio network latency. The TCP parameters have to be configured so as to run TCP successfully. As of now, the applications use only the UDP protocol, which is an unreliable transport protocol that can run without any manual configuration. The packet data service is explained in detail in a separate section.

1.2.2 Types of Data Configurations

Radio to FNE Data : In this configuration data services can be provided between a radio unit and an FNE (Fixed Ethernet) interface. Along with the FNE interface, a PSTN interface might also be supported. If PSTN internetworking interface is also supported, PSTN session control functions must be incorporated into the radio unit. When the circuit data service is provided between the radio unit and FNE interface, the TIA 602 standard AT protocol is implemented for establishment of service on the radio data peripheral. If

PSTN internetworking is also enabled, a PSTN linked host may dial a mobile data peripheral, using overdial or unique circuit data phone number assigned to a project 25 radio, which needs to receive PSTN originated data calls.

When supporting the packet data service in this configuration, packets compliant to the IP protocol standard will only be accepted for delivery across the radio network. The data peripheral devices attached to the network are identified by their IP addresses. The standard IP protocols such as UDP are remoted to the Project 25 through the SLIP protocol. A special protocol called RCP (Radio Control Protocol), is available between the data peripheral and mobile radio for radio specific control functions. The following diagram (Fig. 1-1) is an illustration of a possible Radio to FNE packet data configuration. Specifically the CATLAB project uses this configuration to talk to the server.

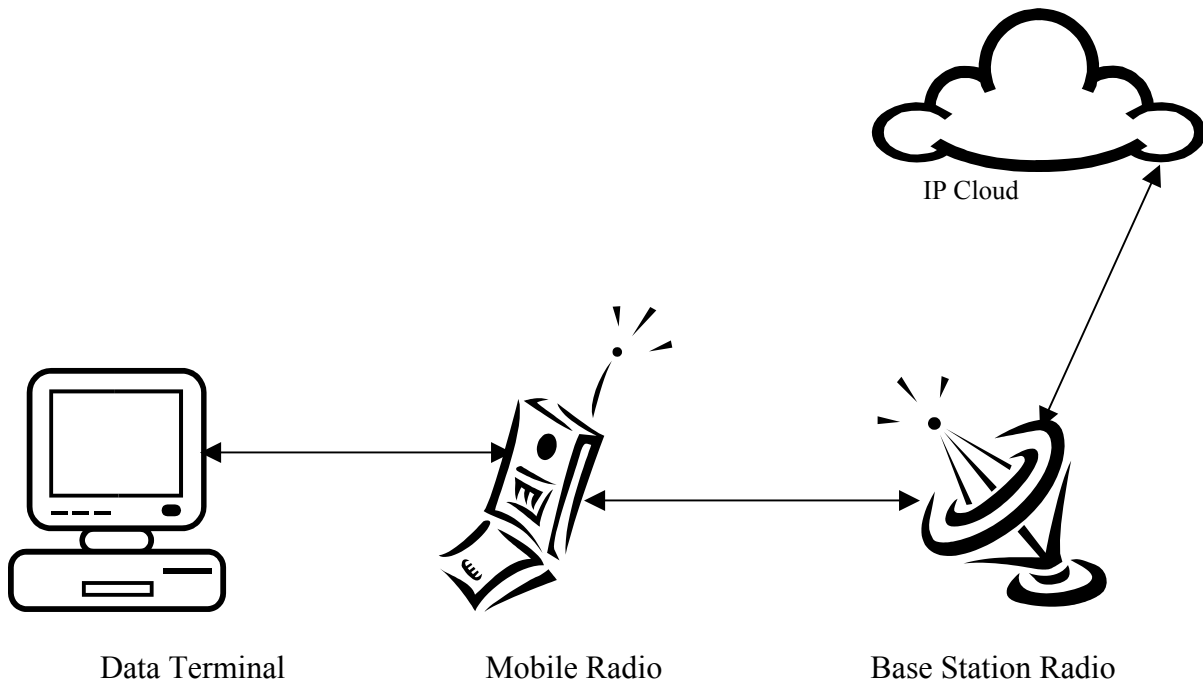


Figure 1–1. Radio to FNE data Configuration

Radio to Repeater to Radio Data: In this kind of configuration data peripheral

devices can communicate over the same area, where repeater services are provided for unit-to-unit communications. Both of the service types (circuit and packet) can be provided with this configuration. In this configuration the mobile originated data transmissions are received and re-transmitted by a base station. The radio subnetwork is a simple repeater.

Radio-to-Radio Data: Radio-to-Radio data is a basic unit-to-unit configuration. The data terminals attached can communicate in the same environment, where radio users can talk unit to unit. In this configuration all mobile-to-mobile hosts interface with one another without any intervening wireless or wired network. The illustrating diagrams for the last two configurations were not provided for brevity.

1.3 Packet Data Specification

This section gives additional internal details of different Packet data configurations (Ref 6). The bearer services are provided by the IP protocol and hence each MDP (mobile data peripheral) is to be assigned an IP address so that the RF subsystem and all the MDPs behave as if they are on a single IP network. The MRCs (mobile Routing/Control) are the radio units that connect to the peripheral data unit over a SLIP link. Therefore each MRC is to be assigned an IP address for the SLIP transmission to and from the MDP. This IP address can be of local significance i.e. only unique within the SLIP link.

1.3.1 FNE data mode:

In this configuration (Figure 1.2) the RF subsystem has an RFG (Radio Gateway) which behaves as an IP router. The RFG has two interfaces, one that connects to the RF subsystem and another that connects to the fixed Ethernet network. Hence an RFG has

two IP addresses. The wireless link in this configuration is between an MRC and a base station. The radio subnetwork consists of switching facilities (RFS), control facilities (RFC) and internetworking gateway functions (RFG) that perform protocol translations to route IP datagrams between radio subnetwork and the Ethernet connected network. The MRC at the mobile end and the RFG at the fixed end needs to be configured before any network transmission is to take place. The SLIP connection between the MDP and the MRC needs to be opened and the RFG must make a binding between MDP's IP address and MRC's CAI address (explained later), which will be explained later. The OSI layers 1, 2 and 3 at the serial interface are TIA/EIA-232-E, serial link Internet protocol and IP respectively. Along with network communication between mobile unit and fixed host (Ethernet), datagrams can be sent between two MDPs, which are routed by the RFG. The MDP and MRC must know each other's IP address so as to exchange control information. The MRC must be placed in FNE (fixed Ethernet) data mode. In this mode the MRC uses non-enhanced addressing in CAI frames, which makes the RFG the destination of all frames sent. The IP software in the MDP must use an MTU (maximum transfer unit) so that a single CAI frame can be mapped to a single IP packet and vice-versa. The RFG must contain a binding between the MDP's IP address and the MRC's CAI address so as to route the IP packets. A data packet generated by a higher layer protocol at the MDP invokes the IP software to build the IP packet with the data and send it over the SLIP link to the MRC. The MRC receives the datagram from the SLIP driver, which is passed to the IP software. The IP software hands over the packet to the CAI interface on detecting that the packet is to be destined to a different MDP. The CAI places the packet in a CAI frame addressed to the CAI address of the RFG. The packet is

delivered to the RFG over the “Um” reference point. The IP software in the RFG receives the packet over its RF subsystem interface. The RFG translates the destination IP address in the packet to the CAI address of the attached MRC. The packet is destined to that MRC which on detecting the destination to be its MDP transmits it over the SLIP link. The following diagram depicts the internal details of the FNE packet data mode.

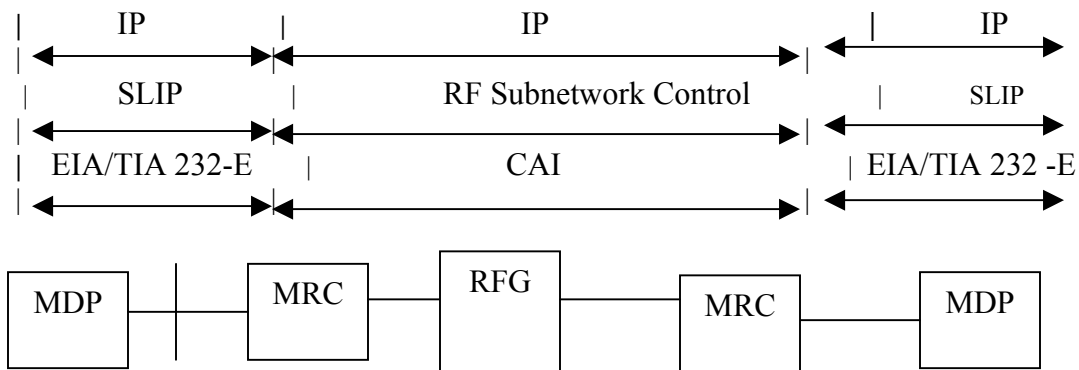


Figure 1-2. Mobile to fixed Host (FNE DATA)

1.3.2 Repeated Data mode:

In this mode the MRC needs to be configured to be in Repeated Data mode. The MRC needs to maintain a binding between the IP addresses of MDPs and the CAI addresses of their corresponding MRCs. The MRC uses an enhanced addressing for CAI frames. All the remaining configurations are same as the previous configurations except that the RF subsystem doesn't have any gateways or switching elements. There is only a repeater base station.

1.3.3 Direct Mode:

This mode is very much similar to the previous configuration except that the

MRCs need to transmit in different radio frequencies for each MDP. There is no repeater element in the RF subsystem. The MRC needs to be configured in Direct mode.

1.4 Control Signaling

This section briefly describes the control signaling involved between the data terminal and its attached police radio connected by the SLIP link (Ref. 7). This control signaling uses the IP bearer service and hence the control packets are encapsulated in IP datagrams. The two control message protocols currently used are ICMP and RCP. ICMP is an established standard for reporting errors occurring at the higher layers above the IP layer. It is a way for gateways and routers to send error messages to hosts (Ref. 4). RCP is a Project 25 defined control protocol (Ref. 7). It provides

- a) Signaling used to initiate and configure an MRC by an MDP.
- b) Signaling used to request information from the MRC.
- c) Signaling used by the MRC to send asynchronous alarms or unsolicited event reports.

In short it is a police radio specific IP stack protocol. Since RCP is not a generic standard special applications need to run both in the MDP and the MRC to send and handle the RCP messages. The application running at the data terminal (MDP) is called as “mobile host configuration and control application”. The RCP protocol runs over UDP. Figure 1-3 and Figure 1-4 show the stack diagrams between an MDP and MRC both when transferring user information as well as in control signaling.

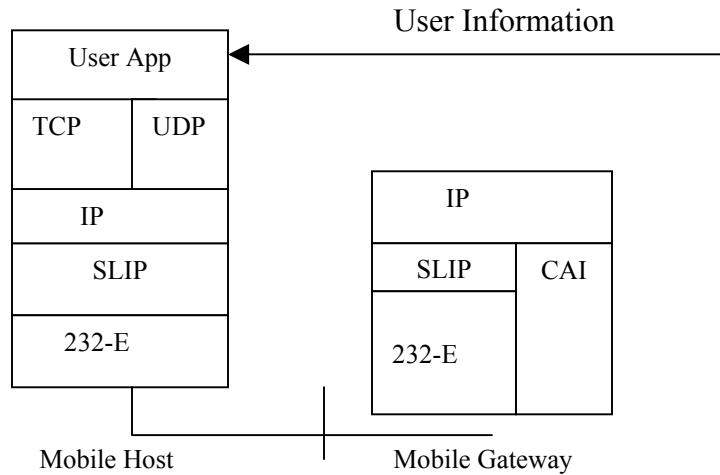


Figure 1-3. Information transfer stack

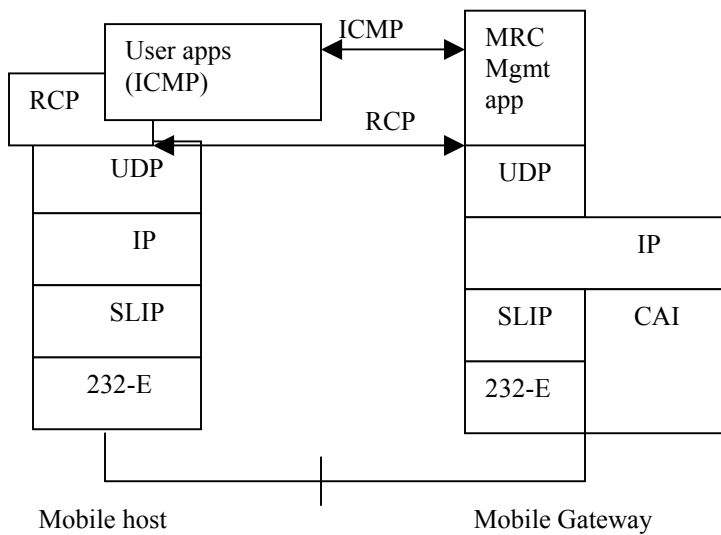


Figure 1-4. Control Information stack

RCP is a request-response architecture protocol. The RCP protocol is not discussed here in detail for brevity.

MDP/MRC Interface: The interface between MDP and MRC specifies the lower 3 layers of the OSI reference Model.

Physical layer: The physical layer uses a subset of the signals defined for the DTE/DCE data interface by EIA/TIA-232-E and CCITT v.24.

Data Link Layer: The MDP/MRC interface at the data link layer is the serial link Internet protocol (SLIP). The CATLAB project has an application specific implementation for this layer. The point-to-point serial link is replaced by an IDB bus serial interface, which is an in-vehicular bus technology.

Network layer: The MDP/MRC interface at the network layer is the Internet protocol (IP). The IP layer in the MDP is a standard host implementation of the TCP/IP protocol stack. This project in particular binds the standard TCP/IP stack to the physical and data link layers using network and link layer drivers (p25com.sys and IDB.sys) respectively. The IP software in the MRC must be capable of relaying IP datagrams among its two physical interfaces. They are the serial interface and the RF interface. The Figure 1-5 depicts the MDP/MRC interface in an APCO25 police radio.

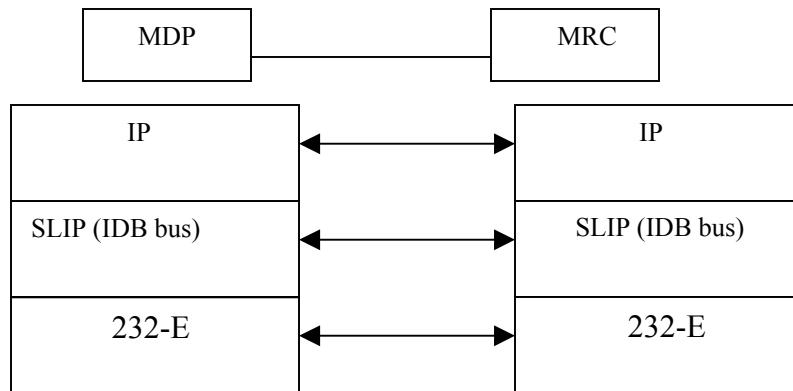


Figure 1-5. MDP/MRC interface

Air Interface:

The radio has the capability of sending IP datagrams over the CAI's (Common Air Interface) confirmed delivery service. The CAI uses two types of addressing on its interface. In non-enhanced addressing, only the source radio device for inbound

messages or destination radio device for outbound messages is explicitly named in the CAI frame header. In enhanced addressing, both the source and destination radio devices are explicitly named in the CAI frame header. The CATLAB project uses non-enhanced addressing.

In FNE data mode, non-enhanced addressing is always used. The resolution of IP addresses to CAI addresses are not required in mobile and portable radios. The FNE frame is capable of transmitting up to 512 bytes of user data. The following Figure 1-6 is the FNE frame format, which is transmitted over the radio interface, in which IP datagrams are encapsulated.

0	A	0	1 0 1 1 0
1	1	SAP identifier	
Manuf. ID			
Logical Link ID			
F	Blocks to Follow		
0	0	0	Pad Octet Ct
S	N (S)		FSNF
0	0	Data header Offset	
Header CRC			

Figure 1-6. FNE frame header format

Here

A: 1(confirmed)

SAP identifier: CAI-SAP_Packet_Data

Logical Link ID: CAI address of source radio device (MRC->FNE) or

CAI address of destination radio device (FNE->MRC)

Data Header Offset: 0

1.5 Advantages of IP Connectivity over Police Radio

As a requirement of the project, the embedded computer needs to run IP based networking applications over the police radio transceiver to communicate over the radio link. This link to the external data source helps quicken the work of the police officers giving them on-the-spot access to information such as DMV files. Officers can check this readily available data before entering dangerous situations, thus providing them a potential safety advantage. Officers can also file reports from their vehicles thus saving their time. A wealth of supporting information will be available to help the troopers make the right decision. This leads to faster response times and higher incident handling capability. Many companies have teamed up with state police organizations to develop products that provide IP connectivity between the police on duty and the base stations, thus improving their safety and efficiency in their tasks at hand. A brief introduction of some of these products is given here to provide a view of on-going research in the field.

1.6 Case study of Radio IP Products

1.6.1 Compaq's Radio IP product for Sacramento Police

Compaq developed a multi-purpose environment in the police cars, which included in-car computing and Radio IP data networking. The police will access these applications over an 800-MHz Data Radio Inc., wireless network that uses Radio IP's RadioRouter software to send and receive data via TCP/IP. The product provides a secure network over the radio link. The software optimized TCP/IP by removing unnecessary overhead,

eliminating duplicate packets generated by the TCP/IP protocol and uses the radio network's confirmation, ensuring security of the information that is transferred. The RadioRouter product combined with the radio's Data communication infrastructure provides high performance as well as compressed and embedded GPS information from a patrol car reported to the server. (Ref.14)

1.6.2 Motorola's Tetra IP

Motorola has already developed a series of products named the Tetra series, which have been deployed in the German police cars. These products provide improved levels of geographic coverage and connectivity to their existing infrastructures. These products provide wireless coverage across very large geographical areas. This allows regional police forces to operate on one integrated system, providing seamless connection across all switches in the system. The Tetra IP's IP connectivity allows for convergence between information technologies and wireless technologies. By using IP, the Tetra product provides for the usage of third party applications as well as the customer's existing IT systems. The design goal of developing the driver for IP in the CATLAB project is also the same. By using Tetra IP products the Police trooper can easily enquire about any record from a central database using packet IP technology.

Many such products have been developed and are underway that help the Police in their safety and efficiently obtaining data. The CATLAB projects IP solution has also served its purpose of running the Project54 Records application and classical IP based applications at the same time allowing for flexibility and ease in the usage of the existing IP radio link. The project design details are explained in separate chapters. (Ref.15)

1.7 Purpose of the Network driver

Previously, IP based applications in the CATLAB project were running over a DLL (Dynamic Link Library) that encapsulated the data in UDP/IP packet and sent it over the serial interface. But having a separate application for IP encapsulation means that running IP applications bound to well-known IP ports such as ping, FTP etc is not possible. Application specific encapsulation and deencapsulation of network traffic with the IP header doesn't really provide an interface to the operating system's IP stack. This means a socket interface for the well-known applications is not available. The goal of this project was to provide a network miniport driver, to provide a real connectivity rather than a temporary solution. A Windows miniport network driver, within the framework of an Ethernet driver, was developed with all the network transmission directed to the serial port. This miniport driver along with a logical driver for IDB frame formatting provided the final solution. The arrowed path in the Figure 1-7 shows the path of IP connectivity achieved.

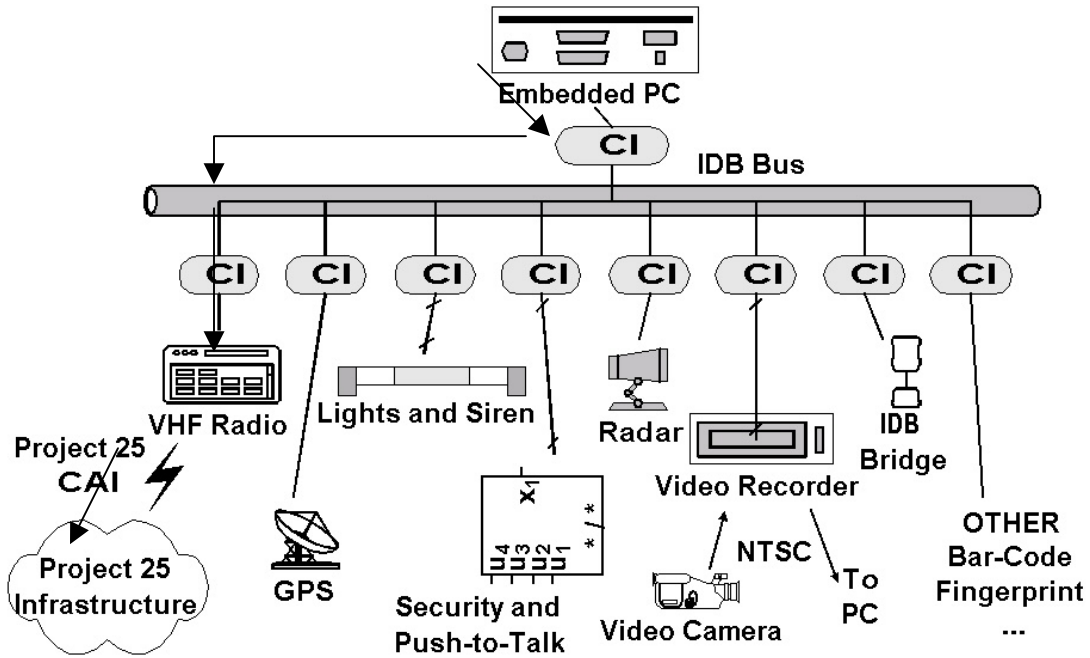


Figure 1-7. IP connectivity in the CATLAB project

1.8 Miniport Driver Specific

1.8.1 Definition

A miniport driver controls a network hardware interface at its lower edge and provides the service of transmitting the network packets handed over by the higher-level protocol drivers at its upper edge. It performs the following basic operations:

- controls a network interface such as registering network interface hardware, sending data, receiving data, resetting a network interface etc.
- binds the higher-level drivers, such as intermediate drivers and transport protocol drivers to the network interface.

The network protocol stack connectivity is shown in figure 1.8

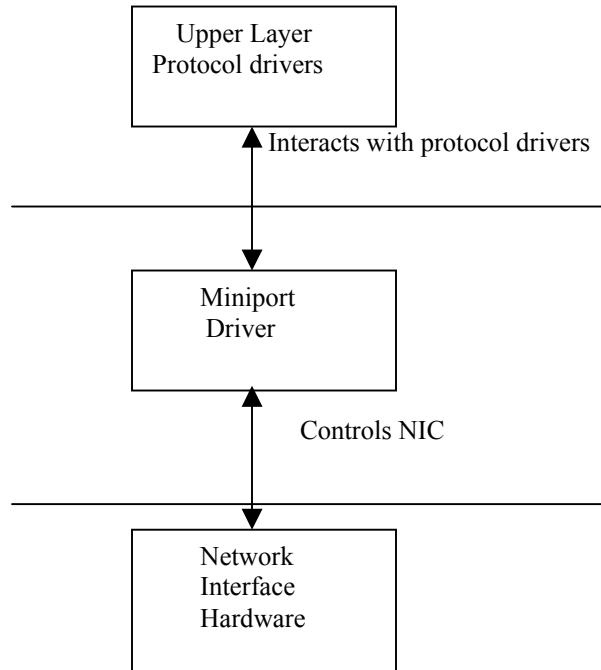


Figure 1-8. Miniport Driver Network stack connectivity

A device driver is a set of Operating system routines put together that bind the protocol layers to the network hardware. A Network miniport driver implements the functionality of a Media Access Control in the Data Link layer of the OSI Reference model. The MAC sub layer has to do encoding of bits, framing, detection of errors and controlling of media access, all of which are implemented in the miniport driver. The functionality provided by miniport driver in the OSI model is shown in figure 1-9.

Application
Presentation
Session
Transport
Network
Data Link
Physical

Figure 1-9. Miniport driver’s position in OSI Reference Model

1.8.2 NIC Miniport Types

Generally miniport drivers will be one of the two following types:

Connectionless Miniports: These miniports manage network cards for connectionless media types such as Ethernet, Token Ring etc.

Connection Oriented Miniports: These miniports manage network cards for connection oriented media types such as ATM.

Since the driver implemented in this work runs over a serial interface it can be classified as being connectionless.

1.8.3 NDIS Library

Windows provides an NDIS library, which is a set of routines that are exported for the proper functioning of any network driver. The library exports a set of function declarations which the miniport driver can call, to manage the network interface at its lower edge and higher protocol layers at its upper edge. This way, any miniport driver developed for any type of network hardware will have a common interface at its lower and upper edges. In turn, the miniport driver provides an interface (a set of routines),

which the NDIS can call to route any requests sent by the higher protocol drivers. The library also provides an interface for the higher layer protocol drivers to route their packets to the miniport driver below them. The library can store state information and parameters for the network drivers such as handles, system values etc.

The NDIS library is implemented as a kernel mode export library called *ndis.sys*. The library functions are declared in *ndis.h*, which the miniport has to include in all its files to call the NDIS functions. The NDIS library significance is shown in

Figure 1-10

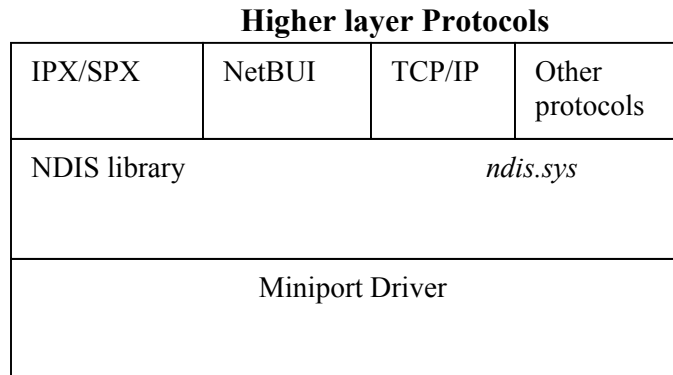


Figure 1-10. Connectionless Network Driver Environment

The NDIS calls MiniportXxx functions (miniport exported interface) whenever it has a packet to be sent, on behalf of the higher layer drivers. The miniport uses NdisXxx functions (Ndis exported interface) to send and receive packets from the upper edge. When a packet is received by the NIC asynchronously, it signals a hardware interrupt, which is handled by the NDIS by calling the appropriate MiniportXxx function, to notify the miniport of the new packet received. The miniport notifies this to the upper layer protocols using an appropriate NdisXxx function. A miniport driver can function in two ways.

- Synchronously, where it responds completely to any request in the same thread context in which the request is received.
- Asynchronously, where it can return a pending status for a request in the same thread and then respond to the request in an arbitrary thread context different from the one in which the request is received. If the miniport driver detects that some operation cannot be performed at the same instant, such as sending a packet through the NIC, it sends a status of pending to the higher layer drivers. When the operation is actually completed it notifies the higher layers of the completed operation by calling an appropriate *NdisXxx....Complete* function.

1.8.4 WDM lower Edge Miniport Driver

A miniport driver with a WDM lower edge is one that doesn't control a network interface card, but a remote device connected to a bus. The driver talks to the remote device over the bus to which the device is connected. Such a driver uses NDIS interface to communicate at the upper edge and the class interface of the particular bus at the lower edge. Specifically for this project, the miniport driver has no NIC, but it communicates over the serial port through which actual data transmission occurs. Hence the driver implemented is conceptually a WDM driver.

A WDM miniport driver calls both NDIS and non-NDIS functions. NDIS and non-NDIS support routines have to be separated into different source files before compiling for efficient functioning of these routines. The NDIS_WDM flag has to be declared for an NDIS-WDM driver that calls WDM support kernel routines. The layered architecture for non-classical network interfaces is shown in Figure 1-11.

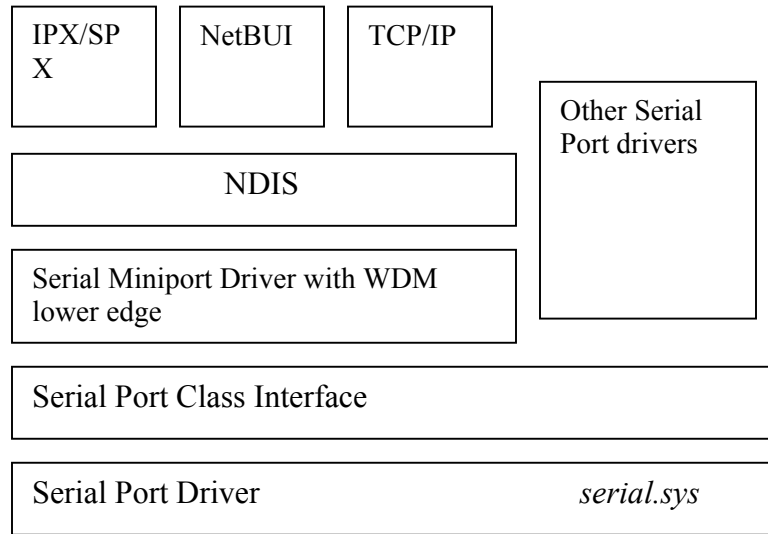


Figure 1-11. Miniport driver for serial port

1-9 Introduction to IDB driver

The CATLAB project involves integrating embedded mobile computing equipment and wireless networking into New Hampshire State Police Cruisers. The task of integrating the car-based electronics systems was achieved by developing a controller for the Intelligent Transportation System Data Bus (IDB). Control Area Networks (CAN) protocol was used as the data transmission and bus arbitration protocol on the IDB. All the electronic peripheral devices in the police car are connected by the IDB bus and exchange data in CAN protocol compliant frames.

IDB is a serial communication bus that creates an open, non-proprietary standard architecture to allow multiple electronic devices to be installed easily and safely in any vehicle. CAN is a serial bus system suited for networking "intelligent" devices within a system or sub-system. CAN protocol can send up to eight bytes of data in each packet from device to device on the IDB bus.

1.10 CAN protocol standard

CAN is a serial bus system with multi-master capabilities, in which all the CAN nodes can transmit data and several CAN nodes can request the bus simultaneously. It provides a complete solution to communication between multiple CPUs. The CAN protocol does not define stations and station addresses, but only their messages. A unique identifier in the network identifies each message. The message identifier defines the content as well as the priority of each message. This is significant when several stations are contending for the bus. The protocol is used widely in in-vehicular networks as well as general-purpose network applications because of its low cost, high performance and the availability of various CAN protocol implementations.

The content oriented addressing scheme provides a high degree of system and configuration flexibility. New stations can be added to an existing CAN network without any hardware or software modifications to the existing stations as long as the new stations are purely receivers. This provides the feature of modular electronics and also allows multiple reception and synchronization of distributed processes. Data can be transmitted in the network in a way that receiving stations do not know the transmitter. This allows easy to service and to upgrade networks, as data transmission is not based on the existence of any particular station. The figure 1-12 depicts the bus architecture and the connection interface of a CAN protocol based network.

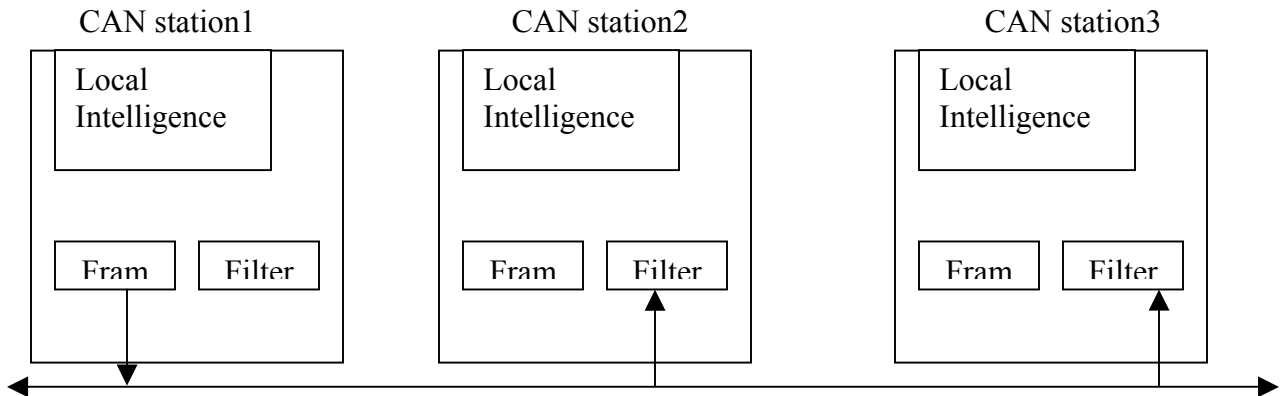


Figure 1 -12. Conceptual schematic of CAN protocol network

The requirements of in-vehicular networks are unique in that they require a high level of error detection, low latency times and configuration flexibility. The CAN protocol satisfies these requirements. First, being a standard communications protocol it simplifies the task of interfacing subsystems from various vendors. This is also actually the primary goal of the CATLAB project. Secondly the overhead of the host CPU is leveraged to the intelligent peripherals. The host CPU (embedded computer inside the police car) can thus run its other high priority tasks. Thirdly, its design helps in having smaller physical lengths of wires connecting the different peripherals, saving the need to have separate point-to-point connections between each device and the host cpu. CAN is a CSMA/CD – A (Carrier Sense Multiple Access by Collision Detection using Arbitration) protocol. Prioritized messages of length 8 bytes or less are sent on the serial bus. Error detection mechanisms such as CRC provide a high level of data integrity.

The standard CAN protocol does not use addressing between its devices. The messages are broadcast on the bus. Each peripheral filters the data at its discretion. But the CATLAB project was designed to have addresses for the peripherals. Each CAN

controller unit has a set of jumpers, which can be set to determine its address. Hence the addresses of the peripheral devices were hard coded. A maximum transmission rate of 1 Mbit/s can be reached over the CAN bus.

1.11 IDB bus

The Intelligent Transport System Data bus (IDB) is a serial communications bus that is intended to provide common network interface for consumer devices, which may be integrated into vehicles. The long development times of automobiles and short times for the electronic devices has led to the in-car electronics being outdated and also difficult to be interoperable. The differences in the design cycles of automobiles and in-car electronics has led to the car being always equipped with older technology. The electronic device needs to undergo rigorous testing before it is installed in a car and this has led to the difference in time cycles. Using an open, standard interface architectural model has solved this problem as in the case of the computer industry. This open standard allowed hardware and software vendors to build value added products for vehicles.

The automobile's primary multiplex bus used for vehicle control has not converged into a standard. Different automobile manufacturers use different standards. This has led to a need to develop different versions of an electronic device for different bus standards. Also the electronic device cannot be connected directly to the multiplex bus, thus interrupting its function. In order to improve safety, a dual bus architecture has been proposed, in which an ITS Data bus (IDB) is connected to the auto's multiplex bus through a gateway. This allows device manufacturers to make a single product, which

can be plugged into different car standards. The gateway acts as firewall between the auto's multiplex bus and the IDB bus allowing only authorized message traffic to flow between the buses, ensuring safe operation of vehicles.

The IDB, which is a serial communications protocol, was designed with a primary goal of connecting consumer electronic devices to a common network in a vehicle without requiring the electronic device manufacturers to develop interfaces to the proprietary vehicle buses. The ITS-bus creates a plug-and-play environment that allows customers to purchase off-the-shelf electronics that works with all of the cars networks.

The Figure 1-13 shows the ITS data bus architecture.

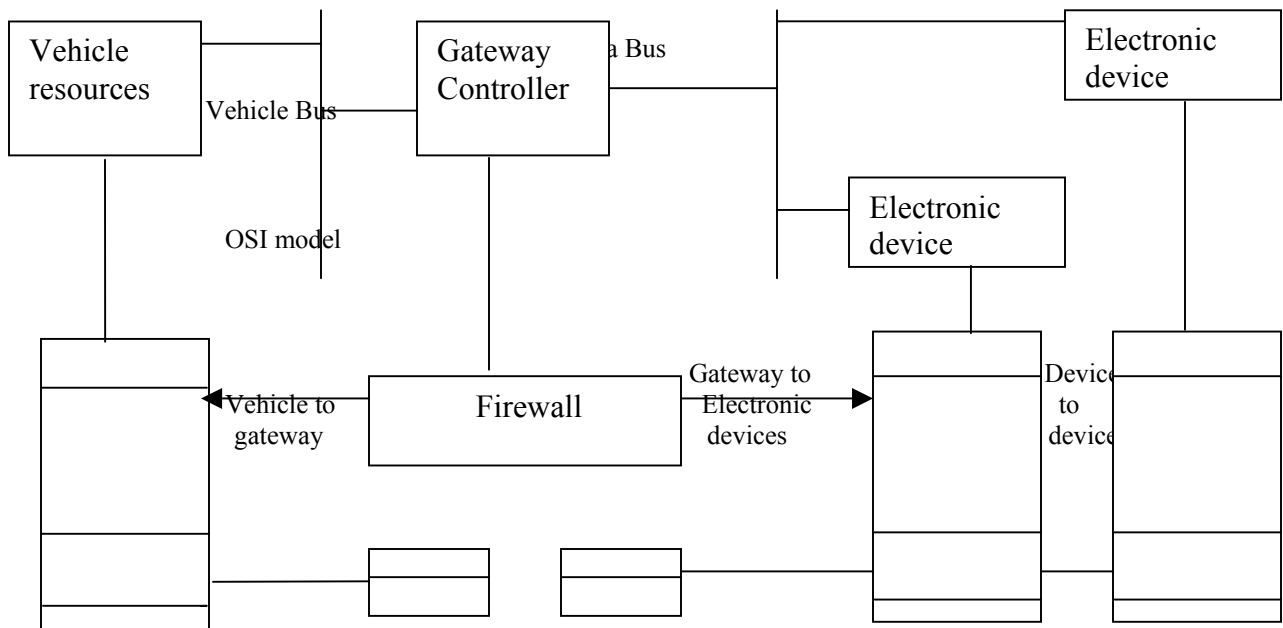


Figure 1-13. ITS Data Bus Architecture

2. NETWORK MINIPOORT DRIVER

2.1 Miniport Driver Architecture

2.1.1 DriverEntry

Any device driver begins execution at its DriverEntry function. The Driver Entry function is a note to the operating system, where the driver has to start execution. The operating system loads the driver into memory by calling this function for that driver. This function performs necessary initialization and boot strapping for the driver such as registering its dispatch routines with the NDIS library, adding the driver to the list of those, which are recognized by NDIS etc. The DriverEntry is passed two parameters: a pointer to the driver object that recognizes this driver in the system, and a pointer to the registry path where driver parameters are stored in the registry. The prototype to the DriverEntry function is

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN  
PUNICODE_STRING RegistryPath);
```

The DriverEntry function calls two functions *NdisMInitializeWrapper* and *disMRegisterMiniPort* that store the driver parameters in the registry and associate the miniport driver with the NDIS. The driver initialization scenario is shown in figure 2-1. (Ref. 2 & Ref. 9)

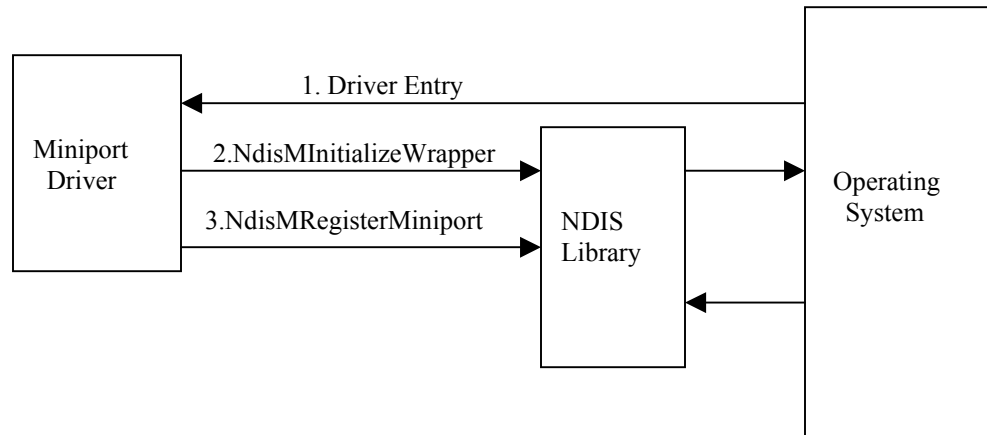


Figure 2-1. Driver Initialization

2.1.2 Miniport Registration

A miniport driver has to register its NDIS version number and the addresses of its dispatch routines with the NDIS. All the elements, which the driver has to register, are packaged in a structure called *NDIS_MINIPORT_CHARACTERISTICS*. The miniport has to supply the addresses of all its mandatory and optional MiniportXxx functions that it implements in this structure, which is passed as a parameter to the function NdisMRegisterMiniport. The Operating system calls these entry points based on the requests it receives, that are to be performed by the driver. These entry points include

- MiniportHalt
- MiniportInitialize
- MiniportQueryInformation
- MiniportReset
- MiniportReturnedPacket
- MiniportSend
- MiniportSetInformation

Some of these functions have been initialized to NULL, since the network driver doesn't

need to handle those requests. The entry points also include interrupt handlers, which are called when an interrupt occurs on the network interface that the driver controls. Since the driver implemented in this project is layered over the serial interface, which handles all the serial interrupts, it need not have any interrupt handlers.

2.1.3 Miniport Initialization

NDIS calls the miniport's Initialization function when the DriverEntry function returns. The driver allocates any buffers (memory) and system resources such as I/O ports, interrupts etc. in this initialization function. NDIS passes a list of media of among which the miniport must select the medium it supports. Since the driver developed here is a pseudo ethernet driver, a medium type of "Ethernet" (NdisMedium802_3) is selected. In this initialization function the driver allocates memory for an internal structure in which it stores its context information. The network interface is considered to be fully initialized when this function returns.

2.2 Driver Implementation details

The network data going from the miniport driver needs to be actually encapsulated in IDB packet frame format before transmitting it over the serial interface. This encapsulation could have been performed at the network driver itself. But there are several other data applications in the CATLAB project which need the IDB packet encapsulation and deencapsulation. For this purpose a separate logical driver called the IDB driver was developed as part of the project. It is explained in chapter 3. The miniport driver was implemented by layering it over a logical IDB driver which actually encapsulates the network data in IDB packet frame format and sends it over

the serial interface. Whenever the network driver needs to send data it encapsulates it in an IP packet and sends it in an IRP (Interrupt Request) request to the IDB driver. In the same way when it needs to receive data it requests the data from the IDB driver in a read IRQ. The driver-layering scheme is shown in figure 2-2.

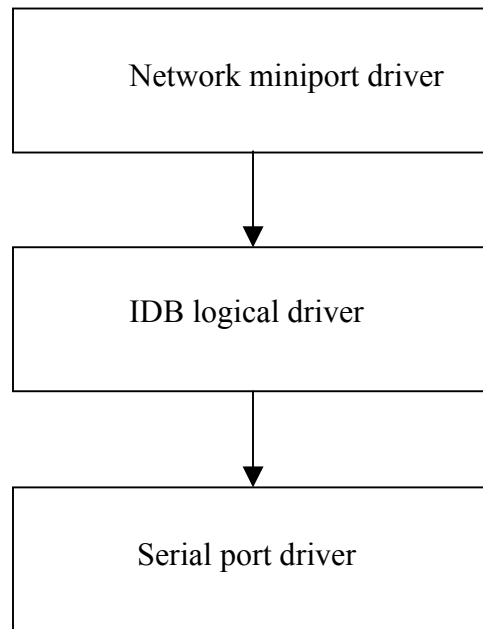


Figure 2-2. Layered driver model

The network driver needs to handle two types of IRP requests, which are

- IRP_MJ_READ
- IRP_MJ_WRITE

The two IRPs correspond to the read and write operation respectively.

2.2.1 Write Operation

The IP stack of the police radio is SLIP encoded. Whenever it receives IP data it considers the data as a SLIP packet and decodes it. In the same way it encodes the network data in a SLIP frame before transmitting it. The IP layering in a police radio was discussed in detail in chapter 1. When the network driver needs to send data, it

encapsulates it in an SLIP frame format and sends it in an IRP_WRITE to the IDB driver.

2.2.1.1 SLIP Encoding

The IP transmission between the embedded computer in the police cruiser and the remote computer at police headquarters is over a radio link. But the in-vehicular network that connects the in-car electronics and the embedded computer by the IDB bus is a point to point serial link. Thus the IP tunnel created is initially a serial link followed by a radio link. The Police radio deencapsulates the data it receives according to the SLIP frame format, which is a point-to-point serial link framing protocol running IP. Thus the network data going through this network driver (interface) was encapsulated and deencapsulated in accordance with SLIP protocol.

There are two special characters in the SLIP protocol, END and ESC. END is Octal 300 (decimal 192) and ESC is octal 333 (decimal 219). When sending a packet, a SLIP host starts sending the data normally as a packet beginning with an END character. If a data byte in the packet is the same code as the END character, a two-byte sequence of ESC and octal 334 (decimal 220) is sent instead. If a data byte is the same as an ESC character, a two-byte sequence of ESC and octal 335 (decimal 221) is sent instead. When the last byte in the packet has been sent, an END character is then transmitted. Thus, END characters occur only between packets, and all bytes between two END characters form a single packet. The initial END character is transmitted to flush off any existing line noise in the channel. Similarly when deencapsulating the packet, the END characters at the beginning and the end are stripped off. Any two successive ESC and octal 334 characters occurring in the data stream are replaced by an END character and any two successive ESC and octal 334 characters are replaced by an ESC character.

There are a few issues to be handled for the SLIP protocol. They are

Addressing: Both at the computers in a SLIP link need to know each other's IP addresses for routing purposes. For this reason the embedded computer in the police car should always know the IP address of the remote computer. The IP addresses of the two machines used will be fixed and they are from the private pool since this link is not connected to the internet.

Type identification: SLIP has no type field. Thus, only a single upper layer can be run over a SLIP connection. Since the tunnel created is only for IP traffic this was not a problem for this implementation.

2.2.1.2 ARP Protocol:

Whatever may be the upper layer protocol suite (eg. TCP/IP), ultimately communication must be carried out by physical networks using the physical address that the hardware supplies. The higher level addresses (IP) need to be mapped into physical addresses for the real communication in a local area network. This is called the address resolution problem, and the protocol used to solve it is called address resolution or commonly the ARP protocol, which is used in broadcast networks such as Ethernet, Token Ring etc.

Any machine with IP address I_A sending an IP packet to a destination at IP address I_B in the same network, first sends an ARP broadcast packet requesting the target machine's hardware address. All the machines in the network receive this packet.

But only the machine whose IP address corresponds to that in the request packet replies directly to the requestor with a packet giving its hardware address in the response packet.

The machine that receives the hardware address binds the IP address to the received hardware address. This is called ARP binding. The miniport driver receives two kinds of

packets from NDIS for transmission on the network (Ref.1). They are

- Ethernet ARP packet to query a destination address.
- Normal IP traffic from peer to peer, embedded in an Ethernet Frame.

The miniport driver developed was based on a classic network driver with all the calls to the network card skipped and the send and receive operations routed to the driver below it. Whenever an IP packet is to be sent, the driver receives an ARP packet from the upper layer protocols to be broadcast in the network, requesting the MAC address of the destination IP address. A reply cannot be received in such a non-broadcast environment, because there is no ARP server or the destination does not respond with the ARP response. This problem was solved by programming the driver to fabricate and return the ARP reply to NDIS by itself, rather than transmitting the ARP request on the network. The upper protocols are unaware of this loop back-answering scheme. All they need is a response to the ARP request to actually send data. Any hardware address except the broadcast address is valid in this scenario since the actual data is not transmitted using this hardware address, but over the serial port.

The Figure 2-3 illustrates the format of an Ethernet frame as defined in the original IEEE 802.3 standard:

Preamble (7-bytes)	Start Frame Delimiter (1-byte)	Dest. MAC Address (6-bytes)	Source MAC Address (6-bytes)	Length / Type (2-bytes)	MAC Client Data (0-n bytes)	Pad (0-p bytes)	Frame Check Sequence (4-bytes)
--------------------	--------------------------------	-----------------------------	------------------------------	-------------------------	-----------------------------	-----------------	--------------------------------

Fig. 2-3. Ethernet Frame Format.

In an Ethernet frame the Length/Type field determines the type of protocol packet Ethernet is framing. The ARP protocol is a separate protocol family with respect to

Ethernet in that it is provided a separate opcode for its family of packets. The Figure 2-4 illustrates an ARP packet sent or received over an Ethernet frame.

Dest. MAC Addr (6)	Src MAC Addr (6)	Protocol Type (2)	Protocol Addr. Space (2)	Byte Length (1)	Byte Length (1)
Opcode (2)	Hardware Addr of sender (n)	Protocol Addr of sender (m)	Hardware Addr of receiver (n)	Protocol Addr of receiver (m)	

Figure 2-4 ARP packet frame format

The miniport driver was designed in such a way that whenever a frame in this format and field values (ARP packet) is received by the driver for transmission, it copies the source address to destination address, and in the place of source hardware address a fabricated value is placed and sent back as an ARP reply packet to the upper layer protocols as if the packet has been received over the network connection. The fabricated address is built from the IP address of the destination so that there is a one-to-one mapping between the fabricated MAC address and the corresponding IP address. The lower order 4 bytes of the MAC address is copied directly from the IP address. A two-byte common value of 0x0001 is appended to the higher order two bytes. For example, if the IP address of server is say 10.3.0.2, then based on the above algorithm the fabricated MAC address will be 00:01:10:03:00:02. This way the intelligence of ARP protocol was obtained locally without ever really searching for a hardware address. This logic works fine as the hardware address of the server is not really required by the host. It is determined by the police radio, which eventually frames the packet in a proper hardware frame.

When a non-ARP packet is received from the top layers, it has to be transmitted

over the network and is therefore handled differently. When a non-ARP IP packet is received, the miniport driver strips off the Ethernet header from the frame, encodes it in a SLIP frame and sends it to the IDB driver, which then sends it over serial port after its own formatting. The Ethernet header includes 6 bytes of source Ethernet address, 6 bytes of destination address (obtained from ARP protocol) and 2 bytes of protocol type, which is IP. This first 14 bytes are taken out of the packet received from upper layers before sending it to the IDB driver. The flow chart of the miniport writes operation in figure 2-5 explains the steps involved when network data is transmitted.

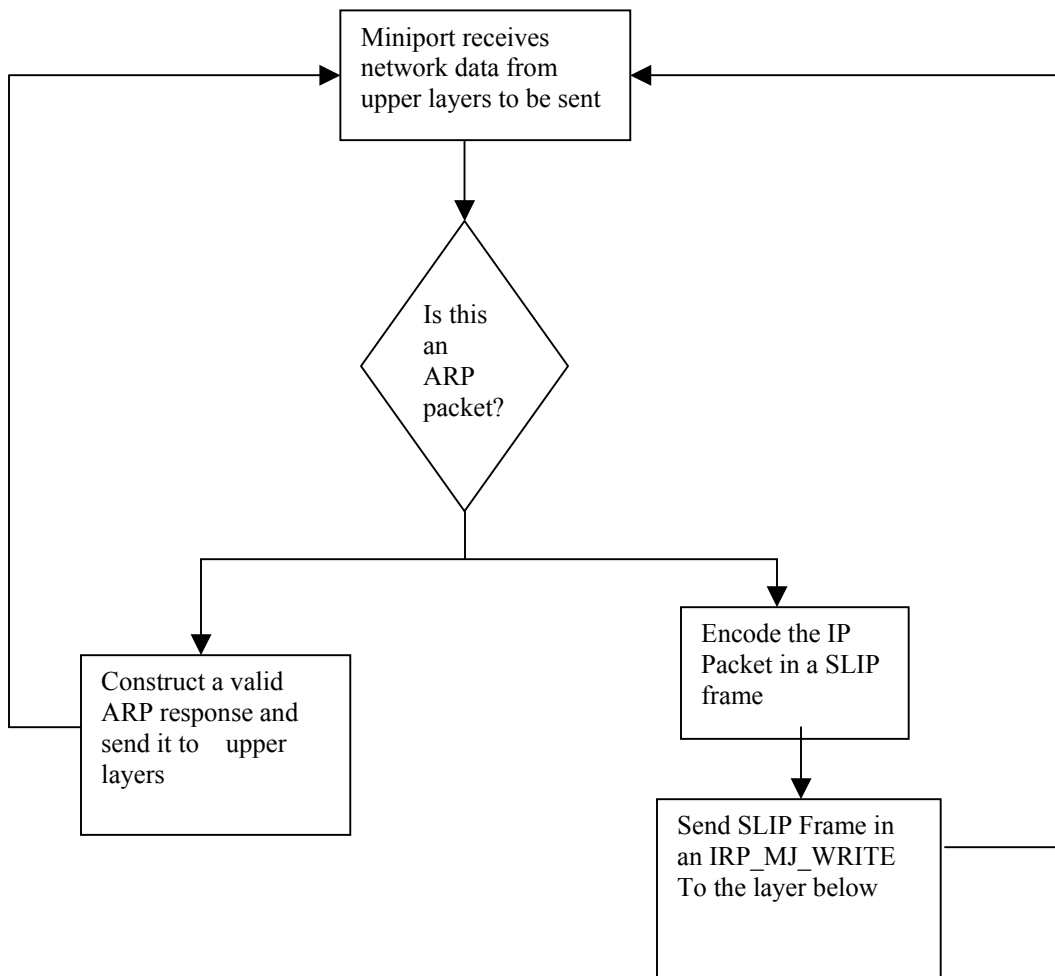


Figure 2-5 Flow chart for Miniport send operation

2.2.2 Read Operation

The miniport driver is not really bound to any hardware interface to actually receive hardware interrupts. Therefore the miniport driver was implemented without an ISR (interrupt request routine). This means that the driver cannot read data asynchronously. That is, it cannot detect the presence of data over the network interface. Hence it has to poll for read data continuously so that the data received is not delayed or lost in its reception. The driver upon its initialization spawns a kernel thread that makes requests for any read data from the layer below at regular intervals. The timeout was calculated for optimality to be 50ms. The kernel thread makes an IRP_MJ_READ request and sends it to the IDB driver every 50 ms. If it receives data it makes another request without delay, because it is highly probable that a large amount of data might have arrived at the interface in a single burst, which cannot be read in a single request. If the read request returns zero data the kernel goes back to a sleep state for 50ms again.

The data received by the miniport driver from the IDB driver is a SLIP encoded IP packet. This packet is first stripped of its SLIP encoding. Upon which the IP packet is encapsulated in an Ethernet frame and sent to the upper protocol layers as if the packet had been received over an Ethernet interface. The framing of an Ethernet header around the IP packet includes placing of the 6 bytes of arbitrary Ethernet address sent as response to ARP packet previously, 6 bytes of the Ethernet address of the host (running miniport driver) as the destination of the packet and 2 bytes of IP protocol opcode in the Type field of an Ethernet packet. The Ethernet protocol intelligence is implemented locally in such a way that the upper layer protocols are ignorant of it. The real transmission and reception is only in a SLIP encoded IP packet over the IDB bus and

then over the radio network.

The SLIP encoded IP packet received by the miniport driver, the packet after SLIP decoding and its Ethernet format when sent to upper layers are shown in the following figures.

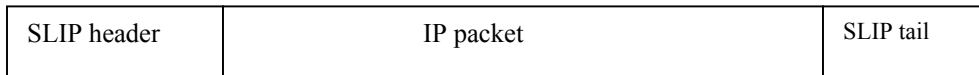


Figure 2-6 SLIP encoded IP packet received by Miniport Driver

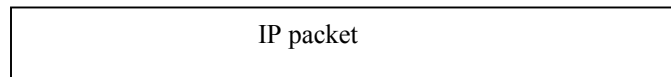


Figure 2-7. IP Packet after SLIP decoding.

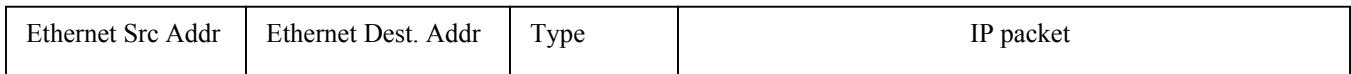


Figure 2-8. Ethernet framed IP.

The miniport driver read operation is depicted as a flow chart in figure 2-9.

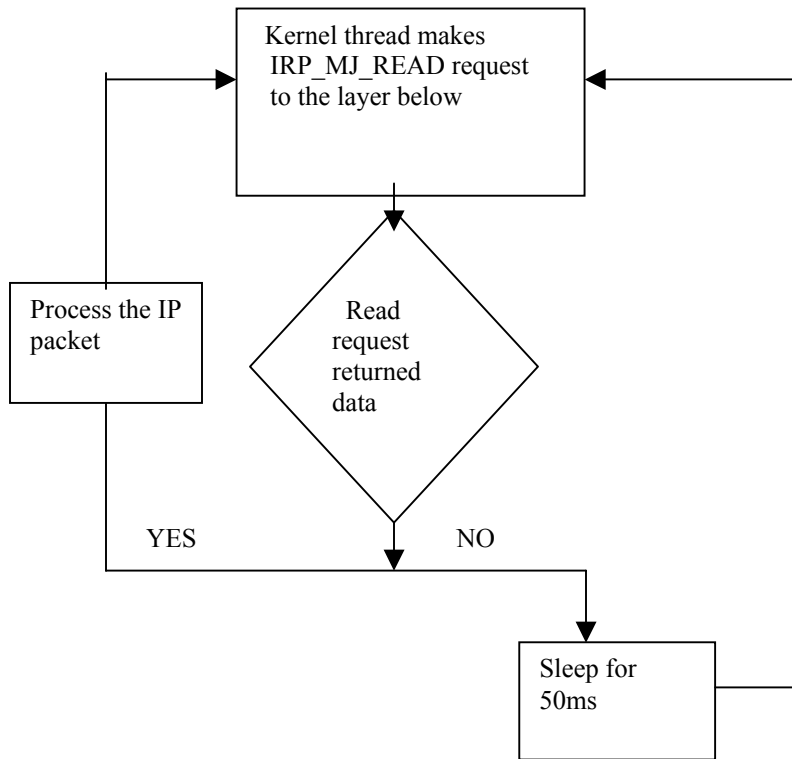


Figure 2-9. Miniport Read operation

The Processing of the SLIP packet and how it is sent to upper layers is shown in figure 2.10

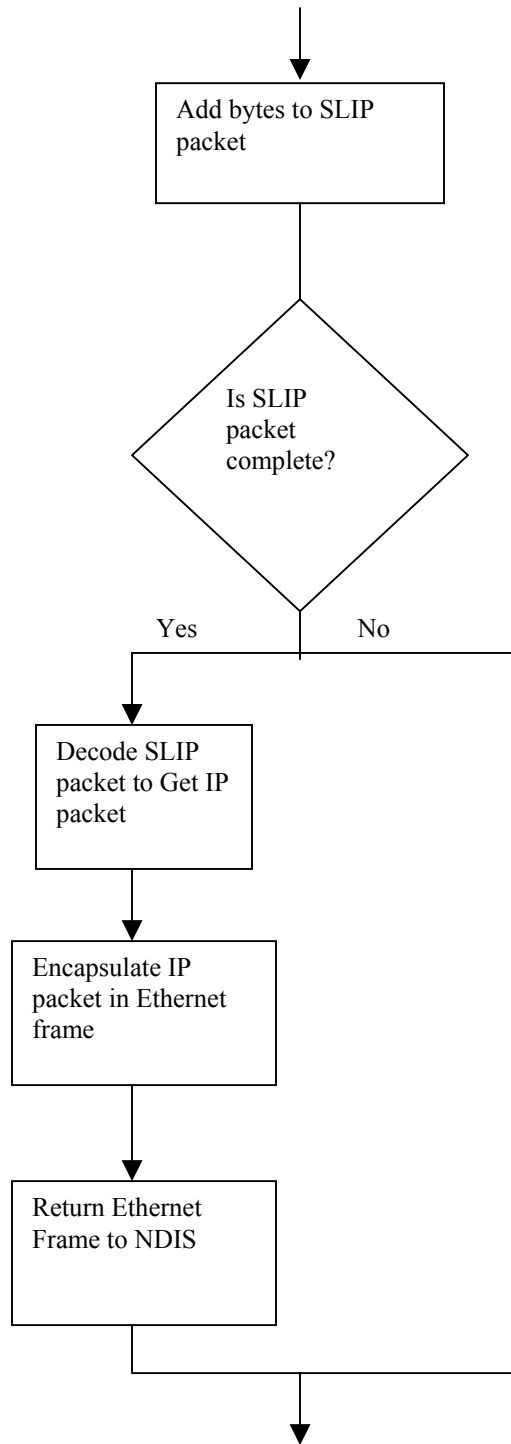


Figure 2.10 processing of a received SLIP packet by the miniport driver

2.3 Different implementations of the Network driver

There are actually two network driver implementations in the CATLAB project. One in which the driver runs in a laptop computer, which connects to the police radio directly via the serial port, and the other in which the driver runs on a embedded computer connected to the radio over the IDB bus. In the first scenario the miniport driver was layered directly over a serial port driver as shown in figure10. since the network data need not be encapsulated in IDB format. In the second case the miniport driver is layered over the logical IDB driver. Except for this layering scheme the functionality of both the scenarios are same. The Figures 2-11 and 2-12 illustrate the differences in the layered stacks.

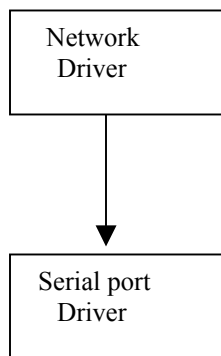


Figure 2-11. Driver layering in a laptop computer

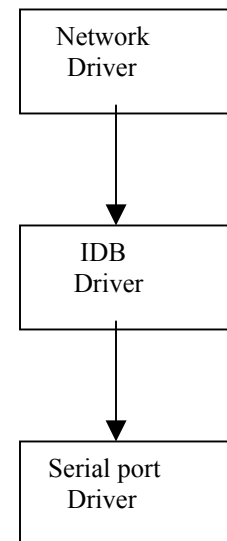


Figure 2-12. Driver layering in embedded Computer connected to IDB bus

III. IDB BUS DRIVER

3.1 CATLAB implementation of IDB bus controller

There already exist many legacy CAN controllers in the market. These designs were altered to be application specific for CATLAB. The IDB bus consists of two types of controllers. One that is connected to the embedded computer side and the other to the peripheral device side. Each Car peripheral electronic device such as a police radio has one IDB controller connected to its serial port. The IDB controller on the device side is designed in such a way that whenever it needs to send data, it transmits the data over the serial interface. The IDB controller connected to this interface takes the serial data from that device and puts it into packets for transmission on the IDB. The IDB controller connected to the embedded computer would take in the packets from the IDB bus and sends it serially to the serial port of the embedded computer. The scenario is exactly the same in the opposite direction of data flow. No peripheral device can talk directly to any other peripheral device. It can only send messages to the embedded computer which interprets it and takes the required action. Also, commands from the embedded computer to less intelligent devices such as the lightbar can be received and interpreted by the IDB controller to perform specific actions to interface with that device's controller. The IDB controller on the device side was implemented with a message filtering so that it received only messages addressed to it. The message filtering does not exist on the computer side controller as the packets sent from any of the device controllers are directed only to it (Ref.11).

The data protocol used between the host PC and the IDB interface was designed to meet the requirements of the CATLAB project. The IDB packet can be of various sizes with the maximum being 8 bytes (Ref.12 &Ref.13). Each packet going from the embedded PC must include the destination device address, number of bytes in the packet, the packet data and checksum. Data sent to the embedded PC from the IDB interface also follows the same packet format. The figure 3-1 shows the data format

Address	Number of Bytes	Data Bytes	Checksum
---------	-----------------	------------	----------

Figure 3-1. IDB Frame Format

All common IDB interfaces (with the exception of the computer side common IDB interface) power-up with transmitters disabled and the baud rates / hardware handshaking information not initialized. In order to set up each common IDB interface one packet needs to be sent to each IDB controller. This packet is of size 4 bytes. The first two bytes are used to identify it as the settings packet. The third byte selects the baud rate for the device. The fourth byte is a flag, which determines whether the device uses hardware handshaking or not. Once the transmitters are set up it is possible to toggle (on/off) the ability for a common IDB interface to communicate on the bus. The common IDB interfaces will come up in a state where it cannot transmit. There are two special packets which can be transmitted to toggle the transmit state of each peripheral device. The figure 3.2 explains the components involved in the Project54 IDB bus.

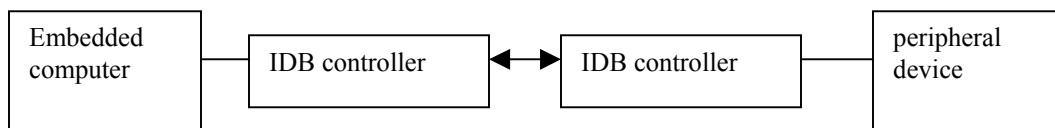


Fig 3.2 IDB bus components

3.2 Purpose of IDB logical driver

Previously, the applications in the embedded computer that needed to talk to the car peripheral devices over the IDB interface used a DLL (Dynamic link library) that had the logic for framing the packets in IDB packet frame format and sending it over the serial interface. The Network driver that was implemented to run IP traffic over the police radio is in the kernel mode. Thus it is an application that needs the IDB frame formatting logic to exist in the kernel mode. Hence the IDB logic had to be shifted to the kernel level. The logic that the DLL implemented was used in the kernel mode driver, with a few modifications.

The functionality of the IDB driver is to take the data it receives from the higher layers (drivers or applications) and sends it in the IDB frame format over the IDB bus. It also needs to receive the data from the IDB bus, de-encapsulate it and give it to the requesting applications. The windows debugging application called WinDbg was used to debug the driver (Ref. 10). The debugger runs on another PC, connected to the embedded computer over a serial interface. The embedded computer, in which the driver was tested, was started in debug mode and its debug statements transmitted and displayed on the debugger were analyzed, to verify the correctness of the driver running in kernel mode. The well-known serial port application HyperTerminal was also used to test the driver.

3.3 Driver implementation Details

Since the IDB bus is connected to the serial interface of the host, the IDB driver

was implemented by layering it over the classical serial port driver. Whenever the IDB driver receives any request from the running applications or upper layer drivers, it performs the processing and sends an I/O request to the serial port driver which does the actual transmission or reception. The IDB driver declares itself to the higher layer applications by creating a symbolic link to itself and notifies it to the Operating System, so that the user applications can make system calls to it (Ref. 16). The Windows Driver Development call to create this symbolic link name is

IoCreateSymbolicLink (device_name, symbolic_link_name);

The figure 3.3 explains the driver layering from the IDB perspective.

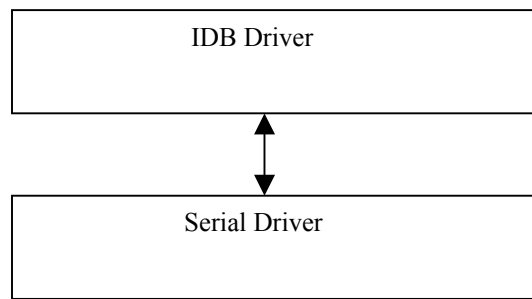


Figure 3-3. Layered drivers for IDB data transmission and reception

The IDB driver handles two types of I/O requests coming from the top. They are

- READ and
- WRITE

The computer architecture is designed in such a way that each I/O peripheral device is connected to the CPU by an I/O controller, which actually controls the device. A Device driver is a software module that controls this I/O controller. The I/O controller will consist of some special registers, which the device driver can read from or write to, to

indirectly control the actual device. The I/O controller will in turn drive the device. The device driver registers an interrupt with the Operating system, corresponding to the I/O controller. Whenever the device controlled by the I/O controller reads or writes data or changes its state, it gets detected by the I/O controller, which generates an interrupt to the OS. The OS calls the interrupt handler that the device driver had registered corresponding to the controller. The interrupt handler processes the interrupt by either reading or writing or noting the state change of the device.

In the case of the IDB driver there is no hardware (I/O controller) involved. Therefore the IDB driver does not know when to read from the serial port driver. As there is no notification mechanism (interrupts), the IDB driver manually polls the serial port driver for any data read. The driver reads from the serial port driver continuously and puts the read data into buffers allocated for each application, which have registered with the IDB driver.

The CATLAB project involves different peripheral devices such as the police radio, the light bars etc. which are connected to the embedded computer. Each peripheral device is assigned a hard coded channel number, by which it is addressed. Any application such as the network driver or a device specific application, which needs to use the services of the IDB, makes an IDB_OPEN request to the IDB driver with the channel number of the peripheral device it wants to talk to. The IDB driver upon receiving the request allocates a handle based on the channel number and returns it to the application. Each handle is unique within the scope of the channel number. There can be several applications with different handles to the same channel number (peripheral device). Each application can also register itself with the IDB driver for different channel

numbers. The figure 3-4 depicts the scenario in which different applications talk to the IDB driver.

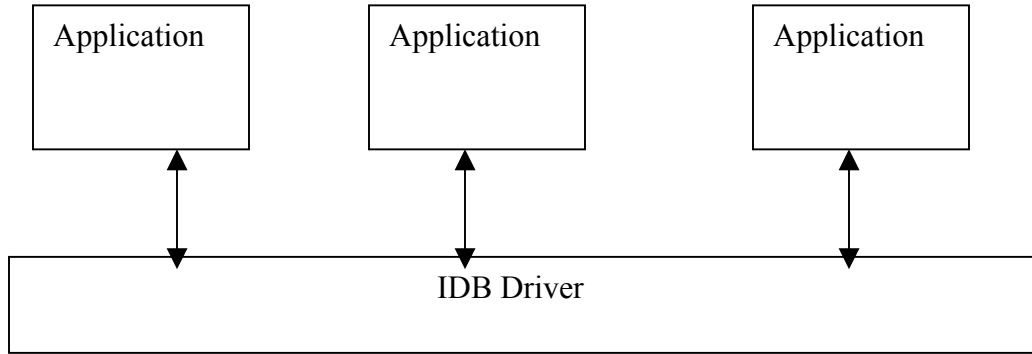


Figure 3-4. Several applications talking to the IDB driver

3.3.1 Write Operation

When the application needs to send a command or write something to a peripheral device, it sends the data along with the channel number to the IDB driver in an IDB_WRITE request. The driver upon receiving such a request puts the data in the CAN protocol frame format and sends it to the serial port driver in an IRP_MJ_WRITE request. The serial port driver upon receiving the IRP_MJ_WRITE, sends the data over the serial port to the host IDB bus controller. Each device connected to the bus, gets the frame and filters it based on the channel number in the frame. In order to make the write operation optimal, the IDB driver was coded in such a way that it can send multiple frames in a single IRP_MJ_WRITE request to the serial port driver, so as to reduce the number of requests made to the serial port driver. The following flow chart gives an overview of the IDB write operation.

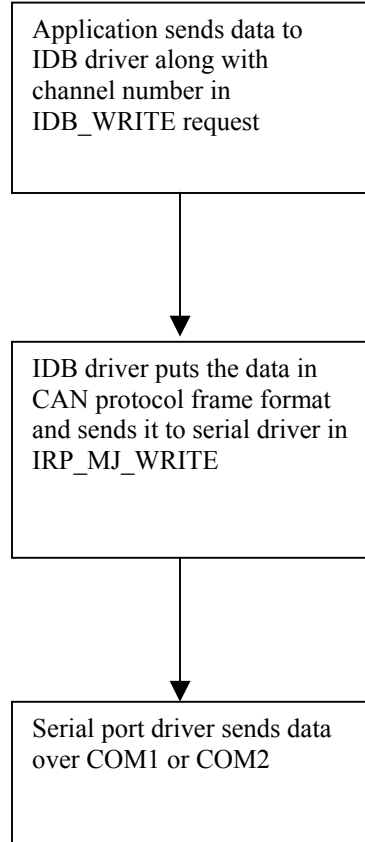


Figure 3-5. IDB Write Operation

3.3.2 Read Operation

As has been discussed previously the IDB driver cannot detect the presence of readable data from the serial port driver asynchronously because of the absence of I/O controller hardware. Thus it polls the serial port driver for read data. This was achieved by sending an IRP_MJ_READ request to the serial port driver at regular intervals. The serial port driver upon receiving this request puts the data along with the number of bytes read, in an IRP and sends it to the IDB driver. The IDB driver upon receiving the data writes it into the buffers it had allocated for each channel number. The data can arrive from any of the peripheral devices connected. The source of the data is recognized from the channel number corresponding to the source, which it puts in the frame header it

sends. The IDB driver upon initialization spawns a kernel thread, which runs independently as a separate task. This thread gives the read requests to the serial driver at regular intervals. In order to improve the performance of the read operation, the thread was coded in such a way that if a read request to the serial driver generates no data, then the time interval for the next read is increased to 50ms. If a read request generates data, then the next request is not delayed. This is a statistical approach for a process. Each application that needs to read data sends an IDB_READ request along with the channel number and amount of data to read, to the IDB driver. The IDB driver upon receiving the request looks for any data in the read buffers for the corresponding channel number and sends it to the applications. When all the applications which have registered with the IDB driver for the particular channel number have read the data, the IDB driver frees the corresponding read buffer so as to reuse the memory.

The following two diagrams give an overview of the IDB read operation.

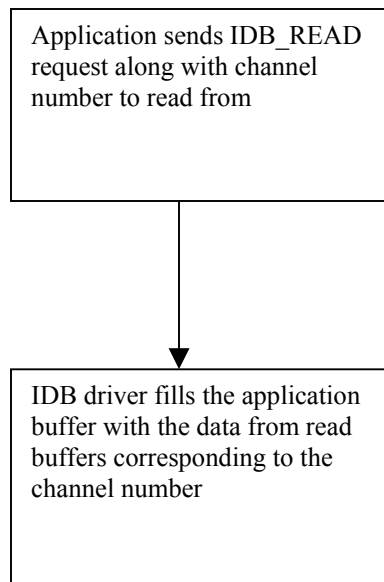


Figure 3-6. IDB read operation

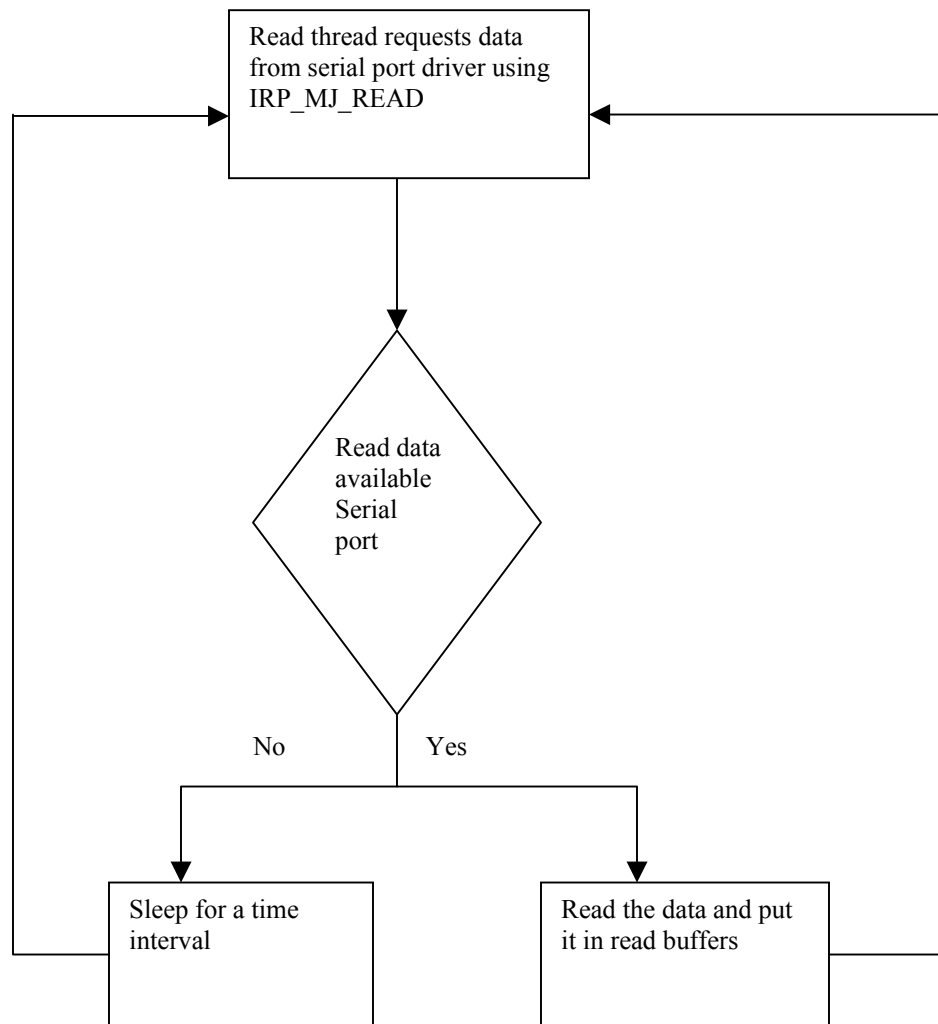


Figure 3-7. Kernel Read Thread operation

3.3.3 Hardware settings

As has been stated before, the IDB bus is connected to the serial port of the embedded computer. The serial port of the computer needs to be configured with hardware settings that match those of the police radio and the IDB bus (explained in the next chapter). The IDB bus has a two-baud rate configuration, one each for its computer side and peripheral device side. On the computer side it transmits and receives at 38400 Hz and on the device

side it transmits and receives at 9600HZ. So the IDB driver when it opens the serial port device driver sets the baud rate to 38400Hz. The police radio is set for RTS/CTS handshaking. The embedded computer behaves as a Data Terminal Equipment (DTE) in an RS-232 configuration and so sends RTS signal whenever it needs to send data and a CTS signal when it is ready to receive data. The police radio on the other hand behaves as a Data Communication Equipment (DCE). The IDB driver upon initialization configures the serial port to perform RTS/CTS handshaking. (Ref. 3)

3.4 User Interface to the IDB driver

The CATLAB project has several applications that communicate with the remote server over the IDB bus. Before communication, each application has to open an interface with the IDB driver. If each application runs as a separate process, then each of them can open the traditional Win32 driver interface and talk to the server over this interface. But all the applications run in a single process and as separate threads. In such a scenario the operating system does not permit multiple connection openings to the same IDB driver and the driver cannot keep track of the data channel of each application. In order to solve this problem the concept of IDB channel handle was conceived in order to multiplex the data between different applications. Each application that needs to communicate opens an IDB channel handle using the *IDBPortOpen* API call. The handle allocated by the driver is returned as output to this function call.

But before obtaining an IDB channel the interface to the IDB driver itself has to be opened by the first application making the *IDBPortOpen* API call. So the *IDBPortOpen* function call checks whether the IDB driver is not open and makes the following generic function call with the symbolic name of the IDB driver as an argument.

```
CreateFile( driversymbolicname,  
          GENERIC_READ | GENERIC_WRITE,  
          0,  
          0,  
          OPEN_EXISTING,  
          FILE_FLAG_OVERLAPPED,  
          0);
```

Then the application makes a call to the IDB driver requesting for a channel handle.

The subsequent calls to *IDBPortOpen* by the applications will just obtain an IDB channel handle from the driver interface, which has already been open.

The Read and Write operations are fairly simple wrapper functions called as *IDBPortRead* and *IDBPortWrite* respectively. They pass the channel handle number and data buffer as arguments to the generic win32 device driver API system calls. Using the generic read and write function calls was not possible as they allowed for only a single input to be passed as an argument. The problem was solved by encoding the read and write as sub-operations to the generic *IRP_MJ_DEVICE_CONTROL* request. The data structure for an *IRP_MJ_DEVICE_CONTROL* request provides for both an input and an output buffer. The control codes passed through the *IRP_MJ_DEVICE_CONTROL* for the different IDB operations are as follows.

IOCTL_IDB_OPEN: This call is a request for a channel handle from the Project54 application to the IDB driver. The channel number is passed as the first byte of the input data to the driver. The driver creates a unique handle for the corresponding channel and sends it back to the requesting application.

IOCTL_IDB_READ: This call is a request for data to be read on a channel handle from the Project54 application to the IDB driver. The channel handle itself is passed in the input buffer and the read length is specified in the output buffer length argument so as to

get around the problem of two inputs. The driver on receiving this requests copies any data for this channel handle into the output buffer allocated by the application.

`IOCTL_IDB_WRITE`: This call requests data to be written over a channel for this handle. The channel handle itself is passed as the first word of the input buffer. The actual write data is appended to this word. The driver decodes the buffer and sends the requested data over the channel number specified.

`IOCTL_IDB_CLOSE`: This call is similar to the `IOCTL_IDBP_OPEN` call, except that the channel handle itself is passed as the input argument. The driver upon receiving this request deallocates any resources allocated for this handle.

IV. RESULTS AND CONCLUSION

4.1 Installation Procedure

As has been discussed previously, the network driver implementation was done in two ways with implementations intended for two scenarios. One for a standalone laptop computer connected directly to a Police radio and the other for an embedded computer connected to the Police radio over IDB bus. The installation of the first case is very simple. Going through the classic network interface installation in Windows, the network driver file for this implementation is selected which is P25com1.sys. The installation in the second case is slightly complicated. The following steps are involved in this case.

- The IDB.sys file corresponding to the IDB driver needs to be generated by compiling the source files using Windows DDK. This file needs to be copied into C:/System32/drivers directory.
- A primary key named, as “IDB” needs to be generated using REGED32.exe program under the HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services directory. The following subkeys with corresponding values need to be created under this “IDB” key.

ErrorControl :	0x01
Start:	0x01
Type:	0x01
Group:	None

ImagePath: System32\DRIVERS\IDB.sys

- The system needs to be reloaded now. The IDB driver is now loaded. To check for the result of the load process, the REGEDT32 program needs to be run again and the IDB key needs to be monitored. If the driver failed to load, “START: FAILED” message will be displayed under the key.
- The procedure for installing a network driver from the control panel is followed and P25com2.sys (second implementation) which is the loadable module is selected, which actually loads the network driver. This step completes the process of installing the IP drivers.

All the software loadable modules created are compiled and built using the Windows DDK software package.

4.2 Results

The results were taken in 3 steps, one each for the driver involved. The third result was that of actually running the records application in a CATLAB police cruiser with both the drivers loaded.

4.2.1 Testing the miniport driver: The miniport driver was tested separately by loading the driver without the IDB driver interface. In this scenario, the miniport driver is layered directly over the serial port and talks to the police radio over serial interface. “Ping” command, which is the well-known ICMP application was run between the radio and the remote server. The server responded successfully, thus verifying the IP link. The data given below is a snapshot of the ping command run.

Pinging 10.3.0.2 with 32 bytes of data:

```
Reply from 10.3.0.2: bytes=32 time<210ms TTL=2270
Reply from 10.3.0.2: bytes=32 time<210ms TTL=2270
Reply from 10.3.0.2: bytes=32 time<230ms TTL=2280
Reply from 10.3.0.2: bytes=32 time<230ms TTL=2290
```

Ping statistics for 10.3.0.2:

```
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 210ms, Maximum = 230ms, Average = 220ms
```

4.2.2 Testing the IDB driver: The IDB driver was tested separately by loading the driver in a PC connected to the IDB interface over serial port. The IDB software implements a ping command by which the interface replies back to a ping request addressed to it. A user mode application opens an interface to the driver and sends a ping request to which the IDB interface responded with the correct response (a single byte of data which is 0x7e). By this the framing the de-framing functionality of IDB driver was tested.

4.2.3 Testing the records application: This test is the actual real-time test with both the drivers loaded in a police cruiser. The new network interface generated is that of the miniport/IDB interface and any network application running over the radio network uses this interface. The records application was tested by successfully logging on the remote server. After the login phase, a number of record queries were executed, to which the server responses were as expected. This tested the whole network stack.

4.3 Future Work

The IP network over the IDB bus has been successfully implemented and tested. The network interface provides flexibility for an application programmer to develop many user level programs, talking to the server at a remote police station.

The police radio supports all the IP protocols. But it is difficult to run TCP because of the

network latency of the radio link. If the radio network's speed is improved in future, TCP applications also can be run efficiently over this link. Standard TCP applications like FTP, telnet etc. can be directly used, which would improve the IP network functionality. The network and IDB driver read threads have been set to a timeout of 50ms, i.e. these threads will check for any read data after every 50ms. A behavioral analysis of the radio link might give a better timeout value to be defined. The Windows driver has been implemented as it is using the basic guidelines of implementing a driver. If the Windows DDK provides a way for a lower level driver to send asynchronous IRP messages to the drivers above it, the network driver's polling read thread can be skipped. This will improve the performance of the driver.

REFERENCES

- [1] An Ethernet Address Resolution Protocol, Network Working Group, David C. Plummer, November 1982.
- [2] TCP/IP Implementation Details for Windows 2000 RC1, White Paper, By Dave MacDonald, July 16 1999.
- [3] Serial Communications in Win32, By Allen Denver, December 11 1995.
- [4] Internet Control Message Protocol, Network Working Group, J.Postel, September 1981.
- [5] Internet Protocol, Protocol Specification, Defense Advanced Research Projects Agency, September 1981.
- [6] TIA/EIA Interim Standard, Project 25 Packet Data Specification, Telecommunications Industry Association, April 1998.
- [7] TIA/EIA Interim Standard, Radio Control Protocol, Telecommunications Industry Association, January 1998.
- [8] TIA/EIA Interim Standard, Project 25 Data Overview, Telecommunications Industry Association, January 1998.
- [9] Walter Oney, Programming the Microsoft Windows Driver Model, pp. 1-217.
- [10] Debugging programs using WinDbg, Hitachi Micro Systems Inc., <http://semiconductor.hitachi.com/tools/apnote5.pdf>
- [11] Martin, Michael E., Hludik, Francis C., and Miller, W. Thomas, "The Project54 common interface for the Intelligent Transportation System Data Bus", IEEE Spring VTC2002, Birmingham, AL, May 6-9, 2002.
- [12] Common IDB Interface Software Updates, Michael E. Martin, 2001
- [13] Project Report for EE 790, IDB controller for CATLAB project, By Michael Martin, May 2001.
- [14] Compaq's Radio IP product for Sacramento Police, Press Release, 2002
- [15] Motorola's Tetra IP Product, Press Release, 2001.
- [16] Pop Open a Set of Privileged services for Windows Kernel Mode Drivers, Resource from www.microsoft.com.

APPENDIX A. Important sections of the driver code

I. Miniport Driver Initialization Routine

```
extern
NDIS_STATUS
P25comInitialize(
    OUT PNDIS_STATUS OpenErrorStatus,
    OUT PUINT SelectedMediumIndex,
    IN PNDIS_MEDIUM MediumArray,
    IN UINT MediumArraySize,
    IN NDIS_HANDLE MiniportAdapterHandle,
    IN NDIS_HANDLE ConfigurationHandle
)

{
    PP25COM_ADAPTER Adapter;

    ULONG i;

    NDIS_STATUS Status;

    //traverses through the hardware media list and selects Ethernet
    802.3

    for (i = 0; i < MediumArraySize; i++){
        if (MediumArray[i] == NdisMedium802_3){
            break;
        }
    }

    if (i == MediumArraySize){
        return (NDIS_STATUS_UNSUPPORTED_MEDIA);
    }

    *SelectedMediumIndex = i;

    // allocate memory for the Ethernet adapter structure
    Status = NdisAllocateMemory((PVOID *)&Adapter,
        sizeof(P25COM_ADAPTER),
        0,
        HighestAcceptableMax
    );

    if (Status != NDIS_STATUS_SUCCESS) {
```

```

        return Status;
    }

    NdisZeroMemory (Adapter, sizeof(P25COM_ADAPTER));

    if (Status != NDIS_STATUS_SUCCESS) {
        NdisFreeMemory(Adapter, sizeof (P25COM_ADAPTER), 0);
        return Status;
    }

    Adapter->MiniportAdapterHandle = MiniportAdapterHandle;

    //set the attribute of the adapter to deserialize
    NdisMSetAttributesEx(
        Adapter->MiniportAdapterHandle,
        (NDIS_HANDLE)Adapter,
        0,
        NDIS_ATTRIBUTE_DESERIALIZE,
        0
    );

    //open an interface to the IDB adapter
    Status = IdbPortOpen(
        (NDIS_HANDLE)Adapter);

    if(Status!= NDIS_STATUS_SUCCESS)
        return NDIS_STATUS_FAILURE;

    P25comMiniportBlock.Adapter= Adapter;

    return NDIS_STATUS_SUCCESS;
}

```

II. Miniport Driver Write (Send) Operation Routine

```

NDIS_STATUS
P25comSend(
    IN NDIS_HANDLE MiniportAdapterContext,
    IN PNDIS_PACKET Packet,
    IN UINT Flags
)
{
    UINT i = 0, j = 0, CurBufLen=0, PacketLength=0;
    PNDIS_BUFFER CurBuffer = NULL;
    PVOID CurBufAddress=NULL;
    NDIS_STATUS status=NDIS_STATUS_SUCCESS;
    PCHAR FullBuffer, StoreBuffer, slip_buffer;

```

```

int slip_count=640;

PP25COM_ADAPTER Adapter =
    (PP25COM_ADAPTER) (MiniportAdapterContext);

/*the data coming from the upper protocols is a scattered gathered
list. This needs to be setup in a single buffer so as to be sent in a
single go to the IDB driver*/

NdisGetFirstBufferFromPacket (Packet, &CurBuffer,
(PVOID *) &CurBufAddress, &CurBufLen, &PacketLength);

status = NdisAllocateMemory(
(PVOID *) &FullBuffer, PacketLength, 0, HighestAcceptableMax);

if(status!=NDIS_STATUS_SUCCESS)
    return NDIS_STATUS_FAILURE;

status=NdisAllocateMemory((PVOID *) &slip_buffer,
                          PacketLength+50, 0,
                          HighestAcceptableMax);

StoreBuffer=FullBuffer;

while(i<PacketLength)
{
    j=0;
    while(j++<CurBufLen)
    {

        *FullBuffer++=*(char *)CurBufAddress;
        ((char *)CurBufAddress)++;

    }
    i+=CurBufLen;

    if(i>=PacketLength)
        break;

    NdisGetNextBuffer (CurBuffer, &CurBuffer);
    NdisQueryBuffer (CurBuffer, (PVOID *) &CurBufAddress, &CurBufLen);
}

FullBuffer=StoreBuffer;
// if the packet is an ARP packet swap the src addr to dest addr and
set src addr to the arbitrary fixed value*/

if (*(FullBuffer+12)==(char) 0x08&&
*(FullBuffer+13)==(char) 0x06&&
*(FullBuffer+14)==(char) 0x00&&
*(FullBuffer+15)==(char) 0x01&&
*(FullBuffer+16)==(char) 0x08&&
*(FullBuffer+17)==(char) 0x00 &&
*(FullBuffer+18)==(char) 0x06&&
*(FullBuffer+19)==(char) 0x04&&
*(FullBuffer+20)==(char) 0x00&&

```

```

*(FullBuffer+21)==(char)0x01&&
(FullBuffer[28]!=FullBuffer[38]||
FullBuffer[29]!=FullBuffer[39]||
[30]!=FullBuffer[40]||
FullBuffer[31]!=FullBuffer[41])

)
{

*(FullBuffer)=(char)0x00;
*(FullBuffer+1)=(char)0x00;
*(FullBuffer+2)=(char)0x00;
*(FullBuffer+3)=(char)0x01;
*(FullBuffer+4)=(char)0x01;
*(FullBuffer+5)=(char)0x01;

*(FullBuffer+6)=0x00;
*(FullBuffer+7)=0x00;
*(FullBuffer+8)=FullBuffer[38];
*(FullBuffer+9)=FullBuffer[39];
*(FullBuffer+10)=FullBuffer[40];
*(FullBuffer+11)=FullBuffer[41];

*(FullBuffer+32)=0x00;
*(FullBuffer+33)=0x00;
*(FullBuffer+34)=0x00;
*(FullBuffer+35)=0x01;
*(FullBuffer+36)=0x01;
*(FullBuffer+37)=0x01;

FullBuffer[38]=FullBuffer[28];
FullBuffer[39]=FullBuffer[29];
FullBuffer[40]=FullBuffer[30];
FullBuffer[41]=FullBuffer[31];

FullBuffer[21]=0x02;

FullBuffer[28]=FullBuffer[8];
FullBuffer[29]=FullBuffer[9];
FullBuffer[30]=FullBuffer[10];
FullBuffer[31]=FullBuffer[11];

FullBuffer[22]=0x00;
FullBuffer[23]=0x01;
FullBuffer[24]=FullBuffer[8];
FullBuffer[25]=FullBuffer[9];
FullBuffer[26]=FullBuffer[10];
FullBuffer[27]=FullBuffer[11];

//send the fabricated packet back to upper level protocols
NdisMEthIndicateReceive(
    Adapter->MiniportAdapterHandle,
    (NDIS_HANDLE)Adapter,
    FullBuffer,
    P25COM_HEADER_SIZE,
    FullBuffer+P25COM_HEADER_SIZE,

```

```

        PacketLength-P25COM_HEADER_SIZE,
        PacketLength-P25COM_HEADER_SIZE
    );

NdisMethIndicateReceiveComplete(Adapter->MiniportAdapterHandle);
status=NDIS_STATUS_SUCCESS;
}

//if it is an IP packet encode it in SLIP format and send it to
// IDB driver

if(*(FullBuffer+12)==(char)0x08&&
    *(FullBuffer+13)==(char)0x00)
{
    // performs SLIP encoding
    IPtoSLIP(FullBuffer+14, (int)PacketLength-
        14,slip_buffer,&slip_count);

    // send it to IDB driver
    Status = IdbPortSend(
        (NDIS_HANDLE) Adapter,
        slip_buffer,
        slip_count,
        CHNL_HANDLE
    );
}

NdisFreeMemory (FullBuffer, PacketLength, 0);
NdisFreeMemory (slip_buffer, PacketLength+50, 0);
return status;
}

```

III. Miniport Driver Read(Receive) Function

```

void HandleIdbPortInput(
    NDIS_HANDLE MiniportAdapterContext,
    PVOID Buffer,
    UINT Length
)
{
    int ip_count =1500,temp=0,flag=0;
    PP25COM_ADAPTER Adapter= (PP25COM_ADAPTER) (MiniportAdapterContext);

    while(temp<(int)Length)
    {
        ip_count=1500;
        flag=SLIPtoIP((char *) Buffer+temp, (int)Length-
            temp,recv_buffer+P25COM_HEADER_SIZE+recv_index,&ip_
            _count);

        recv_index+=ip_count;

        while(*(char *)Buffer+temp)!=-64&&*((char *)Buffer+temp)!=192)
        {
            temp++;
            if(temp==(int)Length)

```

```

        break;

    }
    if(temp<(int)Length)
        temp++;

    // put the buffer in an ethernet frame and send it to upper
    // protocols

    if(!flag&&recv_index)
    {
        // the src address is set to the fixed arbitrary
        // choosen previously

        recv_buffer[8]=recv_buffer[P25COM_HEADER_SIZE+12];
        recv_buffer[9]=recv_buffer[P25COM_HEADER_SIZE+13];
        recv_buffer[10]=recv_buffer[P25COM_HEADER_SIZE+14];
        recv_buffer[11]=recv_buffer[P25COM_HEADER_SIZE+15];

        // send the packet to upper protocols
        NdisMethIndicateReceive(
            Adapter->MiniportAdapterHandle,
            (NDIS_HANDLE)Adapter,
            recv_buffer,
            P25COM_HEADER_SIZE,
            recv_buffer+P25COM_HEADER_SIZE,
            recv_index,
            recv_index
        );

        NdisMethIndicateReceiveComplete(
            Adapter->MiniportAdapterHandle);

        recv_index=0;
    }
}
return;
}

```

IV. The WinDDK specific routine that sends an IRP request to the driver Layered below the current driver

```

NTSTATUS DeviceIoControlRequest(
    IN ULONG IoControlCode,
    IN PDEVICE_OBJECT DeviceObject,
    IN PVOID InputBuffer,
    IN ULONG InputBufferLength,
    OUT PVOID OutputBuffer,
    IN ULONG OutputBufferLength)
{
    NTSTATUS status;
    PIRP pDevControl=NULL;
    PIO_STACK_LOCATION Next_Stack;
    KEVENT SyncEvent;
}

```

```

    // set up sync flag
    KeInitializeEvent(
        &SyncEvent,
        NotificationEvent,
        FALSE
    );

    pDevControl=IoAllocateIrp (DeviceObject->StackSize, FALSE);

    if (pDevControl==NULL)
        return STATUS_INSUFFICIENT_RESOURCES;

    Next_Stack=IoGetNextIrpStackLocation (pDevControl);

    Next_Stack->MajorFunction=IRP_MJ_DEVICE_CONTROL;

    pDevControl->AssociatedIrp.SystemBuffer=InputBuffer;

    Next_Stack->Parameters.DeviceIoControl.
    OutputBufferLength=InputBufferLength;

    Next_Stack->Parameters.DeviceIoControl.
    InputBufferLength=InputBufferLength;

    Next_Stack->Parameters.DeviceIoControl.IoControlCode =IoControlCode;

    IoSetCompletionRoutine (pDevControl, (PIO_COMPLETION_ROUTINE) IoSpecialCompleted, (PVOID) &SyncEvent, TRUE, TRUE, TRUE);

    status=IoCallDriver (DeviceObject, pDevControl);

    if (status==STATUS_PENDING ) {

        status = KeWaitForSingleObject(
            &SyncEvent,
            Executive,
            KernelMode ,
            FALSE,
            NULL
        );

    }

    IoFreeIrp (pDevControl);
    return status;
}

```

V.This function calls a read request to the serial Port driver

```

void ComPortRead(PLOCAL_DEVICE_INFO pLocalDevInfo,
                char *Buffer,
                int size,
                int* value)
{
    NTSTATUS status;

```

```

PIRP irp;
PIO_STACK_LOCATION stack;
KEVENT SyncEvent;

//set up sync flag
KeInitializeEvent(
    &SyncEvent,
    NotificationEvent,
    FALSE
);

// allocate an IRP
irp = IoAllocateIrp(
    pLocalDevInfo->PComPortDeviceObject->StackSize,
    FALSE);

stack = IoGetNextIrpStackLocation(irp);
stack->MajorFunction = IRP_MJ_READ;
stack->Parameters.Read.Length = size;
irp->AssociatedIrp.SystemBuffer = Buffer;

IoSetCompletionRoutine(irp,
    (PIO_COMPLETION_ROUTINE) IoSpecialCompleted,
    (PVOID) &SyncEvent, TRUE, TRUE, TRUE);

// send the driver request
status =
    IoCallDriver(
        pLocalDevInfo->PComPortDeviceObject,
        irp
    );

if (status==STATUS_PENDING ) {

    status = KeWaitForSingleObject(
        &SyncEvent,
        Executive,
        KernelMode ,
        FALSE,
        NULL
    );

}

*value=(UINT) irp->IoStatus.Information;
IoFreeIrp(irp);
KeClearEvent(&SyncEvent);

}

```

VI. This function issues read request to the comport by sending read request irp

```
char getnextchar(PLOCAL_DEVICE_INFO pLocalDevInfo)
{
    static char * Buffer=NULL,*copybuffer;
    PIO_STATUS_BLOCK IoStatusBlock;
    UINT Length;
    static int rval = 0;
    static int rpnt = 0;
    LARGE_INTEGER time;

    time.QuadPart=-1*50*10000;

    // allocate memory for input buffer
    if(Buffer==NULL)
        Buffer = (char *)allocatebuffer(READBUFFERSIZE+1);

    if (rpnt >= rval) {

        rval = 0;
        rpnt = 0;

        while (rval == 0) {

            copybuffer=Buffer;
            ComPortRead(pLocalDevInfo,Buffer,READBUFFERSIZE,&rval);

            if (rval==0)
            {
                KeDelayExecutionThread(KernelMode,TRUE,&time);
                timer++;
            }
            else
            {
                timer=0;
            }

        }

    }

    return copybuffer[rpnt++];
}
```

VII. This is the read thread for IDB driver which continuously polls the comport driver for data

```
void read_thread(PLOCAL_DEVICE_INFO pLocalDevInfo)
{
    char chnl;
    char numdata;
    char data[8];
    char checksum;
    char testsum;
    char i;
    int j;
    int k;
    int dataerr;

    LARGE_INTEGER time;

    time.QuadPart=-1*25*10000;

    while(1) {

        //exit the read thread if the driver is uninstalled
        if(!pLocalDevInfo->ComPortOK)
        {
            read_exit_flag=1;
            PsTerminateSystemThread(STATUS_SUCCESS);
        }

        // get next IDB packet: <channel><size><data ...><checksum>
        dataerr = 1;

        while (dataerr > 0) {

            dataerr = 0;
            // get channel
            chnl = getnextchar(pLocalDevInfo);

            testsum = chnl;

            if (chnl == (char)0) chnl = (char)0xff;

            // get packet size
            numdata = (int)getnextchar(pLocalDevInfo) & 0xff;

            testsum += (char)numdata;

            if (timer > 2)
                dataerr = 1;
            if (numdata > 8)
                dataerr = 2;

            if (dataerr == 0) {

                // get all data in packet
```

```

for (i=0; i<numdata; i++) {
    data[i] = getnextchar(pLocalDevInfo);
    testsum += data[i];

    if (timer > 2) {
        dataerr = 1;
        break;
    }
}
if (dataerr == 0) {

    // get error checksum
    checksum = getnextchar(pLocalDevInfo);

    if (timer > 2)
        dataerr = 1;
    if (checksum != testsum)
        dataerr = 3;
}
}

if (dataerr > 0) {
    char TXon[4] = {0x01, 0x01, 0x01, 0x03};
    char TXoff[4] = {0x01, 0x01, (char)0x81,
(char)0x83};
    char rbuf[128];

    ComPortSend(
        pLocalDevInfo,
        TXoff,
        4);
    KeDelayExecutionThread(KernelMode, TRUE, &time);
    i = 1;
    while(i>0)

        ComPortRead(pLocalDevInfo, rbuf, 128, &i);

        ComPortSend(
            PLocalDevInfo,
            TXon,
            4
        );
}

//We reach here only if data is correct .put the data in
the shared memory

i = 0;
while ((i < NUMCHNL) &&
(m_channel[2*i] != (char)0) &&
(m_channel[2*i] != chnl)) ++i;

```



```

HANDLE ThreadHandle;

InitializeObjectAttributes(&objAttrib, NULL,
                          OBJ_KERNEL_HANDLE, NULL, NULL);

RtlInitUnicodeString(&ComPort, L"\\DosDevices\\COM1");

status=IoGetDeviceObjectPointer(
    &ComPort,
    STANDARD_RIGHTS_ALL,
    &pLocalDevInfo->PComPortFileObject,
    &pLocalDevInfo->PComPortDeviceObject
);

if(status!=STATUS_SUCCESS)
{
    RtlInitUnicodeString(&ComPort,L"\\DosDevices\\COM2");

    Status = IoGetDeviceObjectPointer(
        &ComPort,
        STANDARD_RIGHTS_ALL,
        &pLocalDevInfo->PComPortFileObject,
        &pLocalDevInfo->PComPortDeviceObject
    );

    if(status!=STATUS_SUCCESS){

        RtlInitUnicodeString(&ComPort,L"\\DosDevices\\COM3");

        Status = IoGetDeviceObjectPointer(
            &ComPort,
            STANDARD_RIGHTS_ALL,
            &pLocalDevInfo->PComPortFileObject,
            &pLocalDevInfo->PComPortDeviceObject
        );

        if(status!=STATUS_SUCCESS)
            return STATUS_INSUFFICIENT_RESOURCES;
    }
}

// set the timeouts

timeouts.ReadIntervalTimeout = 100;
timeouts.ReadTotalTimeoutMultiplier = 0;
timeouts.ReadTotalTimeoutConstant = 1000;
timeouts.WriteTotalTimeoutMultiplier = 0;
timeouts.WriteTotalTimeoutConstant = 0;

```

```

status = DeviceIoControlRequest(
    IOCTL_SERIAL_SET_TIMEOUTS, // IN ULONG IoControlCode,
    pLocalDevInfo->PComPortDeviceObject,
    &timeouts, // IN PVOID InputBuffer OPTIONAL,
    sizeof(SERIAL_TIMEOUTS), // IN ULONG InputBufferLength,
    &timeins, // OUT PVOID OutputBuffer OPTIONAL,
    sizeof(SERIAL_TIMEOUTS) // IN ULONG
OutputBufferLength,
);

if (status != STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

// set the data size
linecontrol.StopBits = STOP_BIT_1;
linecontrol.Parity = NO_PARITY;
linecontrol.WordLength = 8;

status = DeviceIoControlRequest(
    IOCTL_SERIAL_SET_LINE_CONTROL, // IN ULONG IoControlCode,
    pLocalDevInfo->PComPortDeviceObject,
    &linecontrol, // IN PVOID InputBuffer OPTIONAL,
    sizeof(SERIAL_LINE_CONTROL), // IN ULONG
    InputBufferLength,
    NULL, // OUT PVOID OutputBuffer OPTIONAL,
    0 // IN ULONG OutputBufferLength,
);

if (status != STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

// set the queue size
queuesize.InSize = 4096;
queuesize.OutSize = 4096;
status = DeviceIoControlRequest(
    IOCTL_SERIAL_SET_QUEUE_SIZE, // IN ULONG IoControlCode,
    pLocalDevInfo->PComPortDeviceObject,
    &queuesize, // IN PVOID InputBuffer OPTIONAL,
    sizeof(SERIAL_QUEUE_SIZE), // IN ULONG
    InputBufferLength,
    NULL, // OUT PVOID OutputBuffer OPTIONAL,
    0 // IN ULONG OutputBufferLength,
);

if (status != STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

```

```

// set the baud rate
baudrate.BaudRate = 38400;
status = DeviceIoControlRequest(
    IOCTL_SERIAL_SET_BAUD_RATE,        // IN ULONG
    IoControlCode,
    pLocalDevInfo->PComPortDeviceObject,
    &baudrate,        // IN PVOID InputBuffer OPTIONAL,
    sizeof(SERIAL_BAUD_RATE), // IN ULONG
    InputBufferLength,
    NULL,            // OUT PVOID OutputBuffer OPTIONAL,
    0                // IN ULONG OutputBufferLength,
);

if (status != STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

// set the flow control
handflow.ControlHandShake =
    SERIAL_CTS_HANDSHAKE;

Handflow.FlowReplace = SERIAL_RTS_HANDSHAKE;
handflow.XonLimit = 2048;
handflow.XoffLimit = 512;

status = DeviceIoControlRequest(
    IOCTL_SERIAL_SET_HANDFLOW, // IN ULONG IoControlCode,
    pLocalDevInfo->PComPortDeviceObject,
    &handflow,        // IN PVOID InputBuffer OPTIONAL,
    sizeof(SERIAL_HANDFLOW), // IN ULONG InputBufferLength,
    NULL,            // OUT PVOID OutputBuffer OPTIONAL,
    0                // IN ULONG OutputBufferLength,
);

if (status != STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

// mark the com port open
pLocalDevInfo->ComPortOK = 1;

//create a system thread for polling read requests

status = PsCreateSystemThread( &ThreadHandle, THREAD_ALL_ACCESS,
&objAttrib, NULL, NULL,
read_thread,pLocalDevInfo);

if (status!=STATUS_SUCCESS)
{
    ObDereferenceObject( pLocalDevInfo->PComPortFileObject );
    return STATUS_INSUFFICIENT_RESOURCES;
}

```

```

}

return(STATUS_SUCCESS);
}

```

// function that closes the com port connection

```

NTSTATUS
ComPortClose(
    IN PLOCAL_DEVICE_INFO pLocalDevInfo
)
{
    // mark the com port closed
    pLocalDevInfo->ComPortOK = 0;

    while(!read_exit_flag)
        ;
    // close device (file) object for COM port
    ObDereferenceObject(
        pLocalDevInfo->PComPortFileObject
    );

    return STATUS_SUCCESS;
}

```

IX. This function sends buffer of output data from IDB driver to the COM port driver

```

NTSTATUS
ComPortSend(
    IN PLOCAL_DEVICE_INFO pLocalDevInfo,
    IN unsigned char * Packet,
    IN UINT PacketSize
)
{
    PIRP irp=NULL;
    void * temp_buf;
    PIO_STACK_LOCATION stack;

    temp_buf = allocatebuffer(PacketSize+2);

    if(temp_buf == NULL)
        return STATUS_INSUFFICIENT_RESOURCES;

    RtlCopyMemory(temp_buf, Packet, PacketSize);

    Irp = IoAllocateIrp(
        pLocalDevInfo->
        PComPortDeviceObject->StackSize, FALSE);

    if(irp == NULL)
    {
        freebuffer(temp_buf);
        return STATUS_INSUFFICIENT_RESOURCES;
    }
}

```

```
}  
  
stack = IoGetNextIrpStackLocation(irp);  
stack->MajorFunction=IRP_MJ_WRITE;  
irp->AssociatedIrp.SystemBuffer = temp_buf;  
stack->Parameters.Write.Length = PacketSize;  
  
IoSetCompletionRoutine(irp, (PIO_COMPLETION_ROUTINE) IoCompleted,  
(PVOID) temp_buf, TRUE, TRUE, TRUE);  
  
IoCallDriver(pLocalDevInfo->PComPortDeviceObject, irp);  
  
return (STATUS_SUCCESS);  
}
```