Design and Implementation of the CiNIC Software

Architecture on a Windows Host

A Thesis
Presented to the Faculty of
California Polytechnic State University
San Luis Obispo

In partial fulfillment
Of the Requirements for the Degree
Master of Science in Computer Science

by
Peter Huang
February 2001
AUTHORIZATION FOR REPRODUCTION
OF MASTER’S THESIS

I grant permission for the reproduction of this thesis (or project report) in its entirety or any its parts, without further authorization from me.

________________________________________
Signature

________________________________________
Date
Title: Design and Implementation of the CiNIC Software Architecture on a Windows Host

Author: Peter Huang

Date submitted: February, 2001

Dr. Hugh Smith

Advisor

Signature

Dr. Jim Harris

Committee Member

Signature

Dr Elmo Keller

Committee Member

Signature
Abstract

Design and Implementation of the CiNIC Software Architecture on a Windows Host

by:

Peter Huang

Internet usage has increased and diversified dramatically in the past several years. Many companies and universities created innovations to enhance the network’s performance and increase its capability to handle diverse types of data. The Calpoly/3Com intelligent Network Interface Card lab (3Com CiNIC lab) is a participant in research and development of methods for improving the capabilities of the Internet.

The CiNIC lab is focused on creating an intelligent Network Interface Card (NIC) to handle web caching, network security, differentiation of data packets and to increase the speed of TCP/IP stack computation. The CiNIC architecture consists of a host motherboard, a co-host network sub-system and a PCI bridge that isolates the host from the co-host. The co-host handles the network transmission for the host. The host OS, which includes Windows NT and Windows 2000, transmits socket function call parameters, network related data structures and network data to the co-host system. The co-host processes this information and makes the actual network socket calls.
This thesis examines the protocols that facilitates the transportation of the network socket call parameters, the socket related data structures and the network data between the Windows host and the EBSA-285 co-host. Our polling protocol, which relies on a polling mechanism, presents the simplest formula for the host/co-host interaction. However, inefficient organization of the shared memory causes too many data movements. Furthermore, the processor performs a busy wait on the host making the polling protocol an inefficient technique for host/co-host interaction.

To overcome these problems, we have developed an interrupt driven protocol. This protocol employs an interrupt mechanism on the host, minimizes the physical movement of the network data and supports multi-threaded access to the host-co-host communication interface. To verify the accuracy of our protocol, we have implemented a simulation. This simulation shows that our interrupt driven protocol successfully transfers data between a client and server application.
# Table of Contents

## Chapter 1 Introduction ............................................................................................... 1  
1.1 Background Concept............................................................................................. 1  
1.2 Recent Research..................................................................................................... 3  
1.3 Corporate Innovation .......................................................................................... 3  
1.4 The 3Com-Cal Poly Lab (CiNIC lab)...................................................................... 5  
1.5 History of the Current Design ............................................................................. 5  
1.6 Overview of the Thesis ....................................................................................... 6  

## Chapter 2 Overview of the Architecture ................................................................. 8  
2.1 Traditional Architecture Vs. Cal Poly Intelligent NIC Architecture ..................... 9  
2.2 Hardware Architecture Components...................................................................... 11  
  2.2.1 The Host (Intel Motherboard)........................................................................... 12  
  2.2.2 PCI bus and Intel 21554 .................................................................................. 13  
  2.2.3 EBSA-285....................................................................................................... 15  
2.3 Software Architecture and Components................................................................ 17  
  2.3.1 The Host Application and DLL......................................................................... 18  
  2.3.2 The Shared memory Interface.......................................................................... 19  
  2.3.3 StrongArm Linux............................................................................................... 20  
  2.3.4 TCP/IP Stack and the NIC driver ...................................................................... 21  

## Chapter 3 Host Socket Interface ............................................................................. 23  

## Chapter 4 The Polling Protocol Implementation and Simulation ......................... 25  
4.1 General Overview of the Host-EBSA 285 Polling Protocol .................................... 25  
  4.1.1 Shared Memory Header................................................................................... 26  
  4.1.2 Overview of the State Machine ...................................................................... 26  
4.2 Data transfer across the PCI Bus with Mapmem..................................................... 28  
4.3 Windows Architecture with the Polling Protocol................................................... 30  
  4.3.1 The Named Pipes and Communication from the DLL to PCICom Service ....... 31  
  4.3.2 EBSAsock DLL............................................................................................... 34  
  4.3.3 PCICom Service ............................................................................................. 35  
4.4 Simulation of the Polling Protocol on Windows NT............................................... 41  
4.5 Assessment of the Polling Protocol....................................................................... 43  

## Chapter 5 The Interrupt Driven Protocol ............................................................... 44  
5.1 Interrupt on Windows 2000 ................................................................................ 45  
5.2 The Shared Memory ............................................................................................ 48  
  5.2.1 The shared memory blocks............................................................................. 48  
  5.2.2 Shared memory headers................................................................................ 49
Chapter 6 Implementation and Simulation of the Interrupt Driven Protocol .... 58

6.1 Overview of Window 2000 Device Driver ....................................................... 58
   6.1.1 Basic terms ................................................................................................. 58
   6.1.2 The Driver Stack ......................................................................................... 60
   6.1.3 General Driver Routines ........................................................................... 61
6.2 WinHost Interrupt Functions and Data Structure ............................................ 62
   6.2.1 ISR ............................................................................................................ 62
   6.2.2 Interrupt Queue Buffer ............................................................................. 64
   6.2.3 DpcForIsr .................................................................................................. 65
6.3 Application Access Routines ............................................................................ 67
   6.3.1 write() .................................................................................................... 67
   6.3.2 ioctl() ...................................................................................................... 68
6.4 Simulation of the Interrupt Protocol ................................................................. 68
   6.4.1 EBSA-285 interface ................................................................................ 69
   6.4.2 IntPoll Program ....................................................................................... 71
   6.4.3 Testing the simulation ............................................................................ 73

Chapter 7 Conclusion and Future Work ................................................................ 77

Bibliography ........................................................................................................... 79

Appendix A EBSAsock structure declaration ...................................................... 81

Appendix B Declaration of shared memory header, shared memory blocks and
   communication signals between the host and EBSA-285 ................................ 84

Appendix C The source code CD ....................................................................... 87
List of Tables

Table 6-1 Doorbell register bits and EBSA-285 signals.................................................. 63
List of Figures

Figure 2-1 The CiNIC Architecture................................................................. 8
Figure 2-2 The Traditional Network Architecture.............................................. 9
Figure 2-3 The Complete Diagram of the CiNIC architecture ......................... 10
Figure 2-4 Hardware architecture .................................................................. 11
Figure 2-5 Host architecture with 21554 drawbridge ...................................... 12
Figure 2-6 EBSA-285 PCI configuration space ................................................ 14
Figure 2-7 Software architecture of the CiNIC board ....................................... 17
Figure 2-8 The comparison of the TCP/IP stack and the ISO seven layer model ... 21
Figure 4-1 The CiNIC architecture with the polling protocol ........................... 30
Figure 4-2 Software Architecture of the CiNIC board with the Polling Protocol ... 31
Figure 4-3 Steps of creating a dedicate pipe between the PCICom Service and a thread.... 33
Figure 4-4 Operations perform by the funcHandle function inside the EBSAsock DLL... 34
Figure 4-5 The PCICom Service........................................................................ 36
Figure 4-6 The state of the PCICom service after an application pipe thread put itself to sleep .............................................................................. 38
Figure 4-7 The PCICom after the Host State machine copies the data package to the shared memory ............................................................................ 39
Figure 4-8 Review of the Data Object ............................................................... 39
Figure 4-9 The PCICom actions after EBSA-285 returns the socket call information to the host...................................................................................... 40
Figure 4-10 Operations of the EBSA-285 Emulation........................................... 42
Figure 4-11 Illustration of how the data return from the recv socket call is already serialized for the trip back to the host ......................................................... 42
Figure 5-1 The IRQL levels of a processor operating on Windows NT/2000 .......... 46
Figure 5-2 Operations of the Windows Interrupt and IDT ................................. 47
Figure 5-3 Registers of the shared memory header.......................................... 50
Figure 5-4 Flow chart of the Host Write Protocol .............................................. 52
Figure 5-5 Flow chart of the synchronization of the Critical Section.................. 54
Figure 5-6 Handshaking of the Host Write Protocol ......................................... 55
Figure 5-7 Handshaking of the Host Read Protocol .......................................... 56
Figure 5-8 Host thread operations after a thread had been “awaken” ................. 57
Figure 6-1 The Windows Driver Stack and Kernel Modules ............................. 60
Figure 6-2 The interactions of interrupt queue buffer with both ISR and DPCForIsr ... 65
Figure 6-3 Operation inside IsrForDpc when receives H_WRITE_CLEAR_ACK ....... 66
Figure 6-4 The software architecture of Interrupt Protocol simulation................. 69
Figure 6-5 Flow chart of the EBSAInterrupt program ........................................ 70
Figure 6-6 Operation of the IntPoll simulation program until the socket call ......... 71
Figure 6-7 Operation of the IntPoll simulation program after the socket call ......... 72
Figure 6-8 The components of the simulation test with SendE ........................... 73
Figure 6-9  The screen dump of the sendE program, which includes the DLL output........... 75
Figure 6-10  The screen dump of the intpoll program when executing the sendE program... 76
Figure 6-11  The screen dump of the interrupt signals by the EBSA Interrupt Program on the EBSA-285 ...........................................................76
Chapter 1 Introduction

Internet usage has increased exponentially for the last several years. The type of data transmitted through the Internet, which consisted mostly of raw text data just one decade ago, has diversified to include audio, voice, graphs and charts, e-commerce, pictures, video broadcast and eventually video on demand. The increase in Internet usage and the diversified data has lead to dramatic increase in the network traffic, which demands higher bandwidth on the transmission medium. As the result, many network corporations place major effort to increase the speed of network transmission.

1.1 Background Concept

The purpose of the network is to facilitate the transfer of information between computers over a transmission medium. The transmission medium typically consists of copper cable, fiber optics, radio waves, and/or network devices such as hubs, switches, bridges and/or routers that direct the path of a network data packet. The Internet is the electronic computer network that connects networks and computer facilities around the world.

When a computer desires to transmit data to another computer, the network subsystem on the computer’s Operating System (OS) must first encapsulate the data along with the error control information, routing information, destination computer address and possibly other network related information into one or more packets before sending the packets out to the network. After it has traveled through the transmission medium and arrived
at the destination computer, the data is extracted from the packet and often error checked before it is regarded as valid.

The amount of time for the sending computer and the receiving computer to process the packet is referred to as sending and receiving processing time. (Denote Ts for send processing time and Tr for receive processing time.) The amount of the time the data spends on the transmission medium is known as the propagation time (Tp). The total transmission time (Tt) from point A to point B is defined as:

\[ T_t = T_s + T_p + T_r \]

Increases in the speed of the network propagation time mean that the processing time becomes a more significant portion of the total transmission time.

Several terms are important in understanding of the network’s performance. Latency is the general term that describes the total transmission time, \( T_t \). Naturally, the smaller the latency, the faster a packet travels from the source to the destination. Throughput describes the amount of data to pass through the network within a set amount of time. The higher the throughput, the more data passes through the network in a fixed interval. A desktop computer performs numerous tasks besides network transmission. The less time a CPU engages in network related operation, the more time it has for other tasks. CPU utilization is the percentage of time a CPU spends in performing a certain task.

Most of the endeavors to increase network transmission speed have focused on increasing the bandwidth of the transmission medium, creating newer communication protocols, and improving the machines that perform the routing. Currently, the bottleneck is the bandwidth of the communication medium rather than the processing time. With the coming of the gigabit Ethernet and the research into the ten-gigabit high-speed network, experts believe that the network bottleneck will shift to the protocol processing time by the
OS[1]. Therefore, the next major improvement for the performance of the overall network would be to decrease the protocol processing latency.

1.2 Recent Research

The TCP/IP network protocol is the dominant network protocol connecting the network infrastructure. There are noteworthy papers on the study of OS processing of the TCP/IP network protocol. The first paper, “Profiling and Reducing Processing Overheads in TCP/IP”[2], presents a detail analysis of the various overhead operations of the TCP/IP processing. The study measures the latency time of various overhead categories such as checksum computation, data movement and data structure manipulation verses data size. It also examines how the distribution of message size differs by transport protocol and environment. Finally, this study suggests a possible means of increasing data throughput by eliminating TCP checksum if the destination is within the local subnet since the data link layer already performs a check sum.

The paper “Communication Performance over a Gigabit Ethernet Network”[1] presents a detailed study of the communication latency and throughput of Gigabit Ethernet. The study compared the TCP/IP protocol stack on various Linux kernel versions, different types of processor and speed, two Network Interface Cards (NIC) and drivers, and various MTU size. Their result concluded that the communication bottleneck has shifted from the bandwidth to hardware and software components such as CPU speed, I/O speed, bus architecture, network adapters, device drivers and protocol stack processing.

1.3 Corporate Innovation

There is a need to accelerate the TCP/IP throughput. And as a result, many leading companies have developed products that shorten the processing time in the TCP/IP stack. For
instance, Microsoft Corporation provides Windows 2000 APIs allowing software developers to offload certain functions of the TCP/IP stack[3], such as check sum and segmentation, from the Windows 2000. In coordination with this enhancement, 3Com Corp supplies an intelligent Network Interface Card[4] (NIC) that contains the 3XP processor built specifically to handle the segmentation and checksum calculation of the TCP/IP stack, in addition to performing flow control for the OS.

The Intel Corporation developed the Internet Exchange Architecture (IXA)[5], a network solution that contains a rich set of software programmable hardware building blocks with a common programming abstraction API. The hardware contains fully programmable processors, including the Intel StrongARM processor, network processors, network chipset sets, standard peripheral components and “microengines” that are specifically designed for high speed packet processing. Both the processors and the microengines are programmable using the Intel IXA SDK API as well as the assembly language for IXA. For example, a developer can program a specific hardware block such as TCP checksum and TCP segmentation into the IXA using certain SDK APIs. An OS can then easily offload those functionalities onto an IXA board.

Besides performing the basic network stack operations, both the 3Com 3XP NIC and Intel’s IXA provide value-added network services, such as IP Security, encryption and multicast. IXA also supports packet differentiating, VPN, load balancing, firewall, intrusion detection and network monitoring. As a result, these services can also be offloaded from the OS if the OS contains one of these sophisticated network devices.
1.4 The 3Com-Cal Poly Lab (CiNIC lab)

The 3Com-Cal Poly Lab, recently renamed as the 3Com Calpoly Intelligent Network Interface Card lab (3Com CiNIC lab), was created in 1996 to investigate maximization of network performance on PC platforms. The initial part of the project focused on the measurement of the network performance. However, the methodologies of measuring network performance must be verified before conducting any network performance measurements. Yu Guang Liang[6] developed techniques to measure the performance of the network performance with a logic analyzer. Su Ming Lo [7] showed that accurate network performance measurement could be realized by software time stamping.

Next, Jim Fischer and Peter Xie performed the actual hardware and software measurements on the network. Jim Fischer implemented a custom client application on Windows and Linux to control and configure the instrumentation, which consists of a logic analyzer probing the PCI bus to capture network packets [8]. Peter Xie time stamped the stack operations with the CPU tick counter and output the timestamp result to a Windows Debugger (Windbg)[9]. Finally, Jim Fischer and Bo Wu compared the hardware and the software instrumentation by conducting the test simultaneously on the network host with IPv6 protocol stack in the Windows NT environment. Based on an analysis of the instrumentation results, it was found that the processing overhead of the TCP/IP stack in both Windows NT and Linux contribute substantially to the latency of the network transmission[8].

1.5 History of the Current Design

Off loading of the TCP/IP stack from the OS to a network specific co-host was our solution to decrease the overhead processing time. Though we do not have a co-processor
swift enough to handle faster networks, we can study the process of offloading the TCP/IP stack and the behavior of the co-host subsystem. Angel Yu designed a network architecture that offloads the TCP/IP stack from the host to a co-host that attaches to the primary PCI bus[10]. Initially, the co-host consisted of the XINU operating system running on 3Com’s Anaconda Network Interface Card power by Intel SA-110 CPU. The XINU operating system was chosen since it’s an open sourced system and it contains a TCP/IP stack. More importantly, it has a small footprint to accommodate the 1Megabyte of RAM restriction on the Anaconda board. Though this co-host could not perform the TCP/IP stack processing faster than the host CPU, this architecture is sufficient for examining how the TCP/IP stack can be offloaded from the host and what the services the co-host can provide. Unfortunately, the XINU assembly code was discovered to be totally incompatible with the SA-110 chip. Therefore, this plan was scrapped in favor of loading Linux onto a new architecture using an EBSA-285 board. The EBSA-285 board contains the same SA-110 chip with 16 Megabyte of RAM. Eric Engstrom spearheaded the effort to compile and load Linux onto EBSA-285 [11]. Since Linux on EBSA-285 would contend with the host OS as the bus master, Ihab Bishara isolated the EBSA-285 Linux from the host with a PCI to PCI drawbridge [12], which creates two separate PCI buses. Please refer to Figure 2-3 for the current architecture.

1.6 Overview of the Thesis

The current hardware architecture consists of the host, the Intel 21554 drawbridge evaluation board and Linux OS on EBSA-285, which will be explained in detail in the next chapter. Chapter two also explains the software architecture of the entire system. Chapter three presents the new socket interface. Chapter four describes the polling protocol, the simulation on the NT and the shortcomings of this protocol. Chapter five discusses the
interrupt driven protocol. Chapter six contains the implementation and simulation of the interrupt driven protocol on Windows 2000. Finally, Chapter seven includes the conclusion and possible future works.
Chapter 2  Overview of the Architecture

As mentioned in the previous chapter, the current focus in the CiNIC project is to offload the TCP/IP stack from the OS to a peripheral board. The computer system with the OS that contains the TCP/IP stack, which is referred to as the host, must interact with the peripheral board, also known as the co-host. Besides the host and the co-host, the CiNIC architecture includes a PCI drawbridge chip (Intel 21554) that is located on the Intel 21554 board. This drawbridge board contains a secondary PCI bus that is isolated from the primary PCI bus. Finally, a NIC card on the secondary PCI bus allows the co-host to interface with the network. Figure 2-1 shows a photograph of this architecture.
The purpose of this architecture is to facilitate the off-loading of the TCP/IP stack from the host to the co-host. It enables us to examine the additional functionalities and services that are available with the co-host. This architecture does not decrease the TCP/IP stack processing time, which hopefully will be provided for in the future architectures.

2.1 Traditional Architecture Vs. Cal Poly Intelligent NIC Architecture

The current network architecture of a PC workstation generally runs either a variant of Windows or Unix OS, which includes the TCP/IP network protocol. A socket interface allows applications to access the TCP/IP stack. Once the TCP/IP stack encapsulates the data plus the control protocol in a packet, the packet is transferred to the NIC via the PCI bus and sent out to the network. Data arriving from the network travels back the other direction, as shown in Figure 2-2:

![Figure 2-2 The Traditional Network Architecture](image)

*Figure 2-2 The Traditional Network Architecture*
Figure 2-3 shows the CiNIC architecture:

In the CiNIC architecture, instead of accessing the TCP/IP stack, the functions in the new socket interface stores the socket call parameters in a data structure that are then passed to the host shared memory interface. The shared memory interface passes the socket-call data structure through the primary PCI bus, the 21554 PCI to PCI bridge, the Secondary PCI bus...
and finally to the shared memory located on the EBSA-285. The shared memory interface on the EBSA 285 is managed by the Linux OS, which passes the data to the Linux TCP/IP stack. All network operations are transmitted or received through the NIC on the Secondary PCI bus. Thus, the host TCP/IP stack is bypassed in this architecture.

### 2.2 Hardware Architecture Components

![Hardware Architecture Diagram](image)

**Figure 2-4 Hardware architecture**

Figure 2-4 demonstrates how the 21554 chip interfaces with the primary and the secondary PCI bus. The 21285 core logic provides the interface between the SA-110, the secondary PCI bus and memory on the EBSA 285 board. The Host chip set provides the host CPU, the peripheral PCI bus and other peripheral devices. Other devices such as the NIC and
SCSI adopter can be attach to the PCI devices. In the CiNIC architecture, a NIC and a SCSI adapter are connected to the secondary PCI bus.

2.2.1 The Host (Intel Motherboard)

![Host architecture with 21554 drawbridge](image)

In support of our architecture, the host transports the socket call parameters from an application program to the PCI bus. The Intel motherboard contains an Intel chipset, an Intel Pentium CPU, memory and other peripheral devices. The chipset contains the north and the south bridge. The higher speed and priority devices such as the CPU, APG and the main memory are attached to the north bridge. The lower speed devices such as the ISA bus, the USB and the IDE are connected to the south bridge. The PCI bus is connected to both the north and the south bridge as shown in Figure 2-5. The 21554 evaluation board plugs into the slot on the PCI bus.

The host architecture supports two methods for transporting the socket parameters. In an I/O write method, the host CPU moves the data from the memory to the 21554 board
through the north bridge. With the DMA method, the host CPU hands a list of physical address to the DMA engine on 21285 core logic. The DMA engine then can directly access the memory through the north bridge without disturbing the host CPU. Thus, the host CPU is free to perform other operations.

2.2.2 PCI bus and Intel 21554

On any system with a PCI bus, the PCI BIOS is responsible for probing and then initializing the hardware devices that are attached to the PCI bus during the system boot up. The initialization process includes assigning of resources to the PCI devices, system self-checks and, with Plug and Play extensions, ensuring that the information used to initialize the system is appropriate.

Once the initialization of hardware devices on the PCI bus has completed and the PCI configuration space for the devices has been created, control of the PCI bus is transferred from the PCI BIOS to the OS. The OS now attempts to load a device driver for each of the detected devices. The device driver requests its desired device by providing the vendor and device ID to the OS. The OS then scans the PCI bus for the requested information. The driver can obtain the location of the resource information during this scan to access the device's particular region of PCI configuration space, e.g., which INT line and processor interrupt has been assigned, etc. Once the driver is loaded for a particular device, the OS moves on to load a driver for the next device.

Figure 2-6 shows the PCI configuration space registers for EBSA-285. The only current relevant register of the PCI configuration space is the “SDRAM Base Address” Register, which contains the starting PCI-relative address of the shared memory on EBSA-285.
Figure 2-6 EBSA-285 PCI configuration space

The 21554 is a non-transparent PCI to PCI bridge, commonly known as a PCI drawbridge. This bridge isolates and controls the traffic from one PCI bus to another PCI bus. As shown on Figure 2-1, the 21554 evaluation board attaches to a PCI slot on the motherboard. The 21554 drawbridge chip isolates the secondary PCI bus located in the 21554 evaluation board from the primary bus on the host motherboard. From the perspective of the host CPU and PCI BIOS, the 21554 chip appears as a single PCI device. All the other devices located on the secondary PCI bus, such as the NIC, are invisible to the host CPU, PCI BIOS, and thus the host OS. This allows the host to consider the co-host subsystem, which includes the co-host on the EBSA-285, the NIC, the SCSI hard drive and eventually any devices located on the secondary PCI bus, as one subsystem.
As a result, the PCI BIOS on the secondary PCI bus limits its detection scope to within the secondary PCI bus itself. Therefore, the 21554 drawbridge closes off the primary bus from the secondary PCI bus just as the secondary PCI bus is invisible from the perspective of the primary PCI bus. Because of this, the EBSA-285 cannot detect the devices on the primary PCI bus. The EBSA-285 Linux regards the 21554 drawbridge as a local PCI device. Both the host OS and the Linux EBSA-285 OS can thus gain control of its own PCI bus once the PCI BIOSs of the respective PCI buses has initialized their PCI devices.

2.2.3 EBSA-285

The EBSA-285 contains a SA-110 microprocessor, the 21285 core logic controller, memory and I/O ports and devices. The card essentially forms a co-processing computer system with the form-factor of a PCI card. The EBSA-285 board allows the SA-110 processor to operate at 16 core clock frequencies between 88.3 MHz and 287 MHz. The factory provided memory is a 16Mbyte DIM upgradable to 256Mbyte.

The EBSA-285 comes furnished with 4Mbytes of flash ROM for its boot path and non-volatile storage. The flash ROM is divided into sixteen 256Kbyte blocks that are individually programmable. A rotary switch selects one of these blocks as the boot block of the PC ROM. In other words, the EBSA-285 can fetch its boot code from any one of the 16 blocks as determined by the rotary switch.

The EBSA-285 functions either in the host mode or in the add-in card mode. Our original plan was to operate it as an add-in card since it was to be directly connected to the PCI bus on the motherboard. Once we realized that the Linux OS on the EBSA-285 cannot operate as an add-in card on the motherboard PCI bus, the configuration was changed to the
host mode on the secondary PCI bus. The only requirement to change the mode is to move a jumper on the EBSA-285.

The EBSA-285 serves as an evaluation board for the SA-110 microprocessor and 21285 Core Logic Controller. It provides a software test and debug environment as a PCI peripheral device or PCI arbiter device. It allows the PC host to interact with the 21285 PCI interface and access the local memory on the EBSA-285. Finally, it supplies a DMA engine that can access the host memory.

The term *shared memory* indicates the memory can be accessed by more than one process. Operating systems with multiple process support and virtual address capabilities such as Linux, Unix or Windows provide a mechanism for multiple programs or processes to communicate with each other via a block of physical memory that the OS maps into the virtual address space of each process. The memory located on EBSA-285 is shared by the programs, processes and processors located either on EBSA-285 Linux or on the host through the PCI bus. The host can obtain the physical address placed on the BAR 3 of the Intel’s 21554 Configuration Space on the primary PCI bus and map this physical address to its virtual address.

The shared memory represents the primary communication area between the Host and EBSA-285. The size of the shared memory can be manually determined by the PCI BIOS or the Linux operating system during boot time. When the host desires to transfer data to the shared memory, it writes the data to the shared memory on the EBSA-285 and notifies the EBSA-285 through a designated location in the shared memory. On the current design, EBSA-285 operates on the data directly in the shared memory. When the EBSA-285 wishes to send data to the host, it notifies the host by setting a bit in the shared memory or by
interrupting the host. The host then copies the data out of the shared memory through the PCI bus and bridges.

### 2.3 Software Architecture and Components

There are two major steps in the software design. The first step is to transport a socket API function call and its parameter information from the host to EBSA-285 Linux so Linux can perform a socket function call with this information. The second step is to return the result of the socket function calls from the EBSA-285 to the host. Besides these two steps, the software design must allow multiple units of execution on the host to be able to utilize this socket architecture.

![Software architecture of the CiNIC board](image)

*Figure 2-7 Software architecture of the CiNIC board*

On most operating systems, the applications interface with the network sub-system is through the BSD socket functions. The socket functions place their function parameters in data structures that are then passed down to the network sub-system. In our implementation, the host stores socket function parameters inside a data structure and then passes the data structure to the shared memory on the shared memory interface to transport the data structure
to the memory on EBSA-285. Any of the socket function parameters that are pointers have their “pointed to” object shipped to the shared memory. Linux on EBSA-285 then performs the actual socket call based on the data structure and objects provided by the host. The result of the socket function call are copied back to the host and delivered to the calling application.

The host and the co-host can communicate with each other through the polling mechanism and/or the interrupt driven mechanism. With the polling mechanism, each side possesses a status register for the purpose of handshaking. Each side has the permission to write to its own status registers but it can only read but not write to the other side’s status register. The polling protocol allows the host and the co-host to transfer data through handshaking with the status registers.

In the interrupt driven mechanism, the co-host communicates with the host by interrupts the CPU of the host. The host saves its working state information and processes the interrupt from the co-host. The host CPU does not need to continuously polling a register and therefore can handle other tasks. The protocol based on interrupts from the co-host to the host is known as the interrupt driven protocol. However, in our initial CiNIC implementation, the host communicates with the co-host through the polling mechanism. In the future, we will enable the host to interrupt the co-host, which would free up the SA-110 on the co-host.

2.3.1 The Host Application and DLL

The applications that access the network are either a network client or a network server. The server could be a standard server that listens on a standard TCP port such as SMTP (port 25) or telnet (port 22), or other custom server applications that use some other
ports. A client application requests a connection with the server through the server’s port number and machine name. The server then creates a socket to interact with the client.

In this thesis, most of the design and explanation around the host OS will be specific to Windows 2000 architecture. The Windows 2000 socket functions are located within the Windows Dynamic Linked Libraries (DLLs). The socket API interface will be referred to as socket DLL. DLL functions are loaded into the physical memory once an application calls a function located within a DLL. Once the OS maps the DLL into the application’s virtual memory space, the application can call any function the DLL exports. Other applications that need to access functions within the DLL can have the OS map the DLL into its virtual memory space and access the desired functions. Once the last application unmaps the DLL from its memory space, the OS can remove the DLL from the physical memory.

2.3.2 The Shared memory Interface

The shared memory interface is responsible for the transfer of socket parameters, network data structures and the network data between the host and the EBSA-285. For the polling protocol on the host side, the shared memory interface consists of the “mapmem” Windows Device Driver, Windows Service, and named pipes to connect the service to the socket DLL. The interrupt driven protocol consists of Windows 2000 plug and play device driver. Both the polling and the interrupt driven protocol on EBSA-285 are located in a Linux device driver. Though the interface on EBSA-285 Linux can be implemented directly in the Linux kernel source code, recompiling the kernel after any change would lengthen the development time. Furthermore, portability from one version of the Linux kernel to another version becomes extremely laborious.
The shared memory interface could be symmetrical or asymmetrical. Symmetrical interface means that the communication interface located on both host and EBSA-285 are identical. Typically, symmetrical protocols are implemented in terms of a state machine. The host EBSA-285 polling protocol, which is detailed in Bo Wu’s thesis[13], is an example of a symmetrical interface implemented on a state machine. In an asymmetrical interface, the communication interfaces located on the host and EBSA-285 are different. The interrupt driven interface would be an example of an asymmetrical interface since the host communicates with EBSA-285 by setting and clearing bits and EBSA-285 respond with interrupts.

Besides communicating with EBSA-285, the shared memory interface on the host side must also handle information to and from the socket DLL. Similarly, the EBSA-285 interface performs socket function calls on EBSA-285 Linux socket API functions while communicating with the host. The socket calls on EBSA-285 should not be the standard Linux socket API in application space, but the equivalent of the socket call inside the kernel space to prevent the context switching from the kernel space to the user space on EBSA-285.

2.3.3 StrongArm Linux

Eric Engstrom successfully ported the Linux kernel onto the EBSA-285 card recently with the assistance of the StrongArm Linux group and Jim Fischer. An “i386-cross-StrongArm” cross compiler compiles Linux version 2.4.0 source code on an i386 or later PC and emits executable images that are compatible with the StrongARM/EBSA-285 platform. In order to load the Linux kernel onto the EBSA-285, the BIOS on the EBSA-285 is loaded with a bootp client and TFTP client. The bootp client broadcast a packet to request an IP address for the NIC located on EBSA-285 (refer to Figure 2-3). The bootp server returns its
IP address, an IP address for the EBSA-285, and also the file name of the EBSA-285 Linux Kernel on the bootp server. The EBSA-285 then downloads the specific kernel image using the TFTP protocol. Once the kernel image arrives on the EBSA-285 memory, it can be started by the StrongARM.

Many enhancements were added to the Linux on StrongARM since its inception. Jim Fischer added the BASH shell, a native C compiler, the VI editor and other libraries and programs to the EBSA-285 Linux. In the future, we plan to modify the EBSA BIOS so that we can load a kernel image from a SCSI drive attached to the secondary PCI bus.

2.3.4 TCP/IP Stack and the NIC driver

Each layer of the TCP/IP stack performs a specific service to accomplish network transmission. For each layer, the data passed from the higher layer is attached to a header to create a protocol data unit (PDU). The entire header-data unit travels down to the lower layer and becomes the data for the lower layer. Figure 2-8 illustrates the layers of the TCP/IP stack in comparison with the ISO 7 layer model. [14]

![Figure 2-8 The comparison of the TCP/IP stack and the ISO seven layer model](image-url)
Once the fully created network packet exits from the physical layer, it is dispatched to the NIC and transmitted out to the network. Any packets arriving from the network are processed from the bottom of the TCP/IP stack.
Chapter 3  Host Socket Interface

The purpose of the socket functions is to provide an interface for application programs to access the TCP/IP stack. As mentioned in the previous chapter, the Windows socket functions, which are located in the winsock DLL, place the socket function parameters and other essential information into a data structure and than pass the structure to the TCP/IP stack. Our new socket API, which will be referred to as EBSAsock, also performs the similar type of task. It provides a set of socket functions with exactly the same function signature to the application. For example, the Winsock “socket” function has the parameters

\[
\text{SOCKET socket}(\text{int af, int type, int protocol});
\]

For the EBSAsock functions, we add an ‘E’ to the name of the socket name:

\[
\text{SOCKET socketE}(\text{int af, int type, int protocol});
\]

The winsock “connect” function, which has the form

\[
\text{int connect}(\text{SOCKET s, struct sockaddr *addr, int namelen});
\]

would be

\[
\text{int connectE}(\text{SOCKET s, struct sockaddr *addr, int namelen});
\]

So the names of the EBSAsock functions are just the regular socket functions plus the character ‘E’.

The main responsibility of each EBSAsock socket function is to generate the Host-EBSA data structure. The core of this data structure consists of a union of structures. Each structure in the union, which I will refer to as the function structures, contains parameter fields that correspond to a particular socket function.
The first responsibility of an EBSAsock function is to create a Host-EBSA data structure. Next, it copies the function parameters into the corresponding function structure inside the union. The Host-EBSA structure contains a type id field that identifies the function structure, and a process id field that identifies the current thread ID. The following listing is a snippet of the Host-EBSA data structure, which we named SOCKET_CALLS_TYPE. The full structure is located in Appendix A.

```c
typedef struct Host-EBSA {
    int type_id;
    int pid;   // process id
    RETURN return_val;
    union func_calls_info // union of function structures
    {
        SOCKET_TYPE socket;
        BIND_TYPE bind;
        .
        .
    }func_calls; // union of function structures.
}SOCKET_CALLS_TYPE // Host-EBSA structure.
```

Once the data structure has been created and initialized, the structure is next passed to the "funcHandle" function. This function is the interface between the EBSAsock socket call functions and the Host-EBSA share memory communication interface. The details of this function for the polling protocol simulation are presented in the next chapter. Chapter seven looks at this function for the interrupt driven protocol.
Chapter 4 The Polling Protocol Implementation and Simulation

The first objective at the CiNIC project was to create a data path that allows the transmission of socket call related information from the host to the co-host. Because a polling protocol is easier to design and implement than an interrupt driven protocol, we choose to start with the polling protocol. By the spring of 2000, we achieved the transfer of data from Windows NT 4.0 on the host to the shared memory on EBSA-285 with the polling protocol. However, since Linux — and thus the TCP/IP stack — was not ported on to the EBSA-285 at that time, we could not perform the actual operation of utilizing the communication protocol, shared memory and TCP/IP stack on EBSA-285 to perform the network connection. As a result, Bo and I simulated our communication protocol on the PC. Bo performed the simulation on a Linux workstation and I implemented mine on a Windows NT workstation. This chapter presents a general overview of the polling protocol and the details of the NT simulation, plus the inefficiencies and deficiencies of this protocol. For an in-depth look into the polling protocol, please refer to Chapter 5 of Bo Wu’s thesis [13]. Chapter 7 of Bo’s thesis describes the simulation of the protocol on Linux.

4.1 General Overview of the Host-EBSA 285 Polling Protocol

The polling protocol consists of the protocol state machine communicating through the share memory. Each process contains a state machine and *shared memory header*. Since there are exactly two processes in the polling protocol between the host and the EBSA-285 –
i.e., the host shared memory interface and EBSA-285 shared memory interface – there are two copies of the state machine and two blocks of shared memory header.

4.1.1 Shared Memory Header

The shared memory headers consist of various registers located at the beginning of shared memory on the EBSA-285. The purpose of the shared memory registers are to facilitate the transfer of an EBSAsock data structure and the objects that are pointed to by the socket function parameter, e.g. sockaddr pointer in bindE function call or buf* in the sendE function, between the host and the EBSA-285. The EBSAsock data structure and the objects pointed by the function parameters comprise the data package for host-EBSA-285 transfer.

The polling protocol utilizes only the status and data size registers in the shared memory header. The status register provides bits for handshaking between the host and the EBSA-285, and the data size register indicates the size of the data transfer. We did not remove the other registers in the shared memory header, because we anticipated that they might be used in future implementations.

4.1.2 Overview of the State Machine

As mentioned in the previous chapter, the shared polling consists of two symmetrical state machines on both the host and the EBSA-285. So an overview on the operation of the state machine applies to either side. This overview is written from the perspective of one of the local side state machine, which could be on either the host or the EBSA-285. The remote side state machine refers to the state machine that is interacting with the local side state machine.

---

1 This section presents a general overview of the polling protocol’s shared memory header data structure. See section 5.2 in Bo Wu’s thesis [13] for a more in-depth discussion of this data structure.

2 This section presents a general overview of the state machine that defines the operation of the polling protocol. Refer to section 5.3 in Bo Wu’s thesis [13] for more detailed information on the state machine design. (Please note that the terminology used in this section might be different from the terminology used by Bo Wu in his thesis)
The state machine is composed of three main sections: the initialization, the data transfer, and the termination sections. When the system is initially powered up, the contents of the shared memory headers are indeterminate. The initialization section ensures that both the host and EBSA-285 arrive at a known state from an unknown state during the system boot up.

The data transfer section of the state machine provides the handshaking mechanism that the host and the EBSA-285 use to coordinate the transfer of data packages from one side to another. This section is further divided into the “read states” subsection, the “write states” subsection and the idle state. The data transfer section begins at the idle state. The idle state loops and tests two conditions. One of the conditions tests if the local side has any data packages ready for transfer. If it does, the local side state machine enters the write subsection states.

The steps perform in the write subsection states:

1. In the case of the Host side, this occurs when one or more EBSAsock functions have a data package ready for transmission to the EBSA-side. On the EBSA-285 Linux, this occurs when returned values or received data from a TCP/IP socket call are placed in a data package.

2. Once a data package is placed in the shared memory and the size register is updated, the local side notifies the remote side that a data package is ready for reading.

3. The local side then returns to the idle state and remains there until one of the two (previously mentioned) idle-state conditions occurs.
The other condition tested in the idle state examines if the remote side has placed a data package in the shared memory. If so, the local side enters the “read subsection” states. On the host, it reads information that the EBSA-285 has generated during a socket call. On the EBSA-285, it reads the EBSAsock socket function parameters and the data that the host has provided for transmission over the network.

The data transfer section of the state machine also ensures that neither side will monopolize the communication by allowing fair sharing of the shared memory among the host and EBSA-285. If the local side state machine exits the write subsection state and subsequently detects that it has more data packages ready for transfer, it must wait to allow the remote side to have a chance to write to the shared memory before it can reenter the write subsection.

The termination stage allows the host and EBSA-285 communication mechanism to finish transferring the last data package before the system shuts down. Once the state machine enters the termination stage, it must travel through the initialization stage again in order to become available for data transfer.

4.2 Data transfer across the PCI Bus with Mapmem

To access the shared memory through the PCI bus, Windows NT/2000 either interface directly with EBSA-285 or through the Intel 21554 drawbridge as in the CiNIC architecture. The Intel 21554 evaluation board and the EBSA-285 board are standard PCI boards that can be plugged in to any available PCI slot on the host PCI bus. Windows NT/2000 communicates with a PCI device via the PCI device driver for that particular device.

In general, the device driver contains modules or data structures that cannot be placed outside of the driver, such as the interrupt service routine (ISR), DMA functions or ISR
queues. Also, any module and/or data structure that causes more than the minimum number of context switches between the user and kernel space or that interacts directly with a device should reside in a device driver. Otherwise, the module should be placed in user space since user space provides more security and features.

The polling protocol interacts with the shared memory that is located in the kernel space as well as the DLL in the application space. This creates a dilemma on whether to place the shared memory polling protocol in a kernel space driver or in a user space application. Since the polling protocol does not include interrupts, DMA, or features that require it to exist within a kernel-layer device driver, we have written it in the user space. On the other hand, there must be a way for the polling protocol to access the shared memory from the user space. To do this, we utilized the “Mapmem” device driver [10], which allows a user space application to access the shared memory.

The Mapmem device driver that ships with the Windows NT 4.0 Device Driver Kit [15] allows a process to map a device’s RAM into the process’s address space. In this case, it translates the PCI relative address of the shared memory on the EBSA-285 into an application program’s virtual address space. Thus, an application on NT can directly access the shared memory on the EBSA-285.
4.3 Windows Architecture with the Polling Protocol

With the Mapmem driver, we were able to implement the polling protocol as a Windows Service. We called this service “PCICom.” The first task of the PCICom service is to retrieve the shared memory address. Figure 4-1 presents the overall CiNIC architecture with the polling protocol and Windows Host.
Figure 4-2 presents the software architecture of the CiNIC architecture, which is similar to Figure 2-7 with the additions of the details in the Windows polling protocol design.

**Figure 4-2 Software Architecture of the CiNIC board with the Polling Protocol**

### 4.3.1 The Named Pipes and Communication from the DLL to PCICom Service

A *named pipe* is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients. All instances of a named pipe share the same pipe name, but each instance has its own buffers and handles, and provides a separate conduit for client-server communication. Any process can access named pipes, subject to security checks. This makes named pipes an easy form of communication between related or unrelated processes. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network [16].

A named pipe is employed as the communication medium between the application program with the EBSAsock DLL and PCICom Service. One of the first responsibilities of the PCICom service is to create a thread to serve as the named pipe server. Through the DLL,
any application can request the named pipe server to create a dedicated pipe from the application to the PCICom service. Figure 4-3 illustrate the process of creating a dedicate pipe. The client pipe thread represents the application. A dedicated pipe between the client thread and the PCICom service ensures that only the intended client receives the data.

Each thread in a user’s application communicates with the pipe service via its own dedicated pipe. If a user application spawns a new thread, another dedicated pipe is created for that new thread. A thread local storage space in the DLL monitors whether a thread holds a dedicate pipe. Immediately after a new thread attaches to the DLL, a thread local storage region is created and initialized to FALSE for the particular thread. Before a thread transfers a data package, it checks the thread local storage for the existence of a dedicated pipe. If the thread local storage value is FALSE, a dedicate pipe is created and the value is switched to TRUE. If the value is TRUE, the thread simply transfers the data package via the already created dedicated named pipe. The dedicated pipe terminates upon the termination of the thread.
Step 1:

The client thread opens up a connection with the Pipe Server. Once connected, the client transmits its thread ID to the server.

Step 2:

The client thread terminates the connection with the Named Pipe server and creates its own named pipe with its thread ID as the name.

In the meantime, the Name Pipe server creates an Application pipe thread.

Step 3:

The Application pipe thread connects with the client thread by the name of the client thread, which is the name of the dedicated pipe for data transfer between the PCICom server and the application.

Figure 4-3 Steps of creating a dedicate pipe between the PCICom Service and a thread
4.3.2 EBSAsock DLL

As mentioned in Chapter three, once the Host-EBSA data structure has been occupied in the EBSAsock functions, it is passed to the “funcHandle” function. Figure 4-4 lists the actions performed within the “funcHandle” function.

<table>
<thead>
<tr>
<th>“funcHandle” function</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Call function to Serialize the data package in a buffer.</td>
</tr>
<tr>
<td>- Check if Pipe for the thread exists. If not, create a dedicate pipe.</td>
</tr>
<tr>
<td>- Send the buffer to the PClCom Service through the pipe.</td>
</tr>
<tr>
<td>- Wait for the result of the socket function to arrive from the pipe and place it in the buffer</td>
</tr>
<tr>
<td>- Call function to deserialze the data package from the buffer.</td>
</tr>
</tbody>
</table>

**Figure 4-4 Operations perform by the funcHandle function inside the EBSAsock DLL**

First, it calls the data package copying functions. These functions, referred as the serializing functions, serialize the data package by placing the EBSAsock data structure and the objects “points to” by pointer parameters of the EBSAsock functions. The serializing functions are organized with an array of structures with function pointers. The type ID in the Host-EBSA data structure indexes into the array of structures. There is one serializing function for each socket function with distinctive parameters. For example, the bind function call with the parameters

```c
bindE(SOCKET , struct sockaddr , int)
```

would share the same serializing function as

```c
connectE(SOCKET, struct sockaddr , int).
```

However, the

```c
acceptE(SOCKET, struct sockaddr , int*)
```

function would have its own serializing function.
After calling the serializing functions, funcHandle checks the value of the thread local storage to determine if a pipe to the PCICom service has already been created. If not, the thread would request a dedicated pipe connection from the pipe server, as illustrated in Figure 4-3. Once a pipe exists, the data package in the serialized buffer is sent to the PCICom service via the pipe.

After sending the data package, funcHandle waits for the return of the EBSAsock socket call from the EBSA-285 via the PCICom service and the dedicated pipe. Since the pipe passes data via a buffer, the deserializing functions parse the parameters and data from the buffer and place them in a) the EBSAsock data structure and b) the objects “pointed to” by the parameters with pointer types. The functions with same signature share the same deserializing functions.

4.3.3 PCICom Service

A Windows service is a Win32 executable program without a user interface. Three Win32 processes control the operation of each Windows service application. The first process is the Service Control Manager (SCM). SCM controls the Win32 services. It orders the service to stop, start, pause, continue and so on. It also handles the communication between the services. The second process is the service itself, which this section will describe. A third process is called the Service Control Program (SCP). The SCP provides the user interface that allows a user to interact with the SCM and tell the SCM to perform operations on the services.

The PCICom service handles the host side of the Host-EBSA-285 communication protocol. Therefore, it must be up and running before any process begins network communication. When the service is started, it first retrieves the shared memory address from
the Mapmem driver. Next, it spawns a thread for the host-EBAS polling protocol. Finally, it spawns a thread to serve as the named pipe server.

The service contains a queue, a list, and one main global variable. The send data queue stores the EBSASock socket function call data packages in the order they arrive. The global variable “SendQCnt” tracks the size of the send data queue. The Wait List contains the threads that are waiting for socket function completion from the EBSA-285. These two data structures and the global variable facilitate all communication between the application pipe thread and the state machine. Figure 4-5 depicts the thread and data structures of the PCICom Service.

![Diagram of PCICom Service](image)

**Figure 4-5 The PCICom Service**

After an application pipe thread has been created, it attempts to make a pipe connection to the application (Please refer to Figure 4-3 step 3 for the details of this connection). Once the connection has been established, the application pipe thread enters a loop that lasts until the requesting thread decides to terminate the pipe connection.
Inside the loop, the application pipe thread first waits for the serialized data package to arrive from the application. When it arrives, the application pipe thread creates a structure called SOCKET_CALL_TYPE_P (stands for SOCKET_CALL_TYPE pointer). This structure will heretofore be referred to as the “Service Data Element.” The Service Data Element forms the basic element of the send data queue and the wait list. This data structure has the fields:

```c
typedef struct SC_P { //The Service Data Element
    HANDLE tevent; // store the thread signaling event
    char *element; // points to the buffer
    DWORD size; // store the size of the data in the buffer
    SC_P *next; // for the send queue and the wait list
} SOCKET_CALLS_TYPE_P; // The Service Data Element
```

The `tevent` field provides a mechanism for the application pipe thread that created the particular Service Data Element to be awakened after the socket call information has returned from the EBSA-285. The `element` field points to the data package in the buffer received from the dedicated pipe and later for the data package that arrived from the shared memory. The `size` field indicates the total size of the data package that the `element` field points to. Finally, the `next` field provides linking of the queue and the wait list.

After the Service Data Element has been created, its `element` field is set to the data package received from the dedicated pipe and the `size` field is updated to be the size of the data package in the buffer. The `event` field is filled with a Windows event set to the non-signaling state. Next, the structure is attached to the end of the Send Data Queue. The head of the Send Data Queue is a global variable of type SOCKET_CALL_TYPE_P*. There is also another global pointer at the end of the queue for the insertion into the queue. Then the global variable SendQCnt is incremented by one. Finally, the application pipe thread puts itself to sleep waiting for the event to signal.
Figure 4-6 illustrates the appearance and operations of the PCICom service when an application pipe thread puts itself to sleep.

As mentioned above, the host communication state machine polls on whether any host application has any data package ready for sending. In this case, it polls on the SendQCnt value. If the SendQCnt value is greater than or equal to one, and if the host side is permitted to send data to the EBSA-285, then the state machine: a) transfers the data package that is pointed to by the element field of the first Service Data Element at the front of the queue to the shared memory, b) updates the shared memory header, c) moves the Service Data Element to the wait list, and d) decrements the SendQCnt. The following figure displays...
the situation inside the service after the state machine has moved a data package to the shared memory.

Figure 4-7 The PCICom after the Host State machine copies the data package to the shared memory

```c
EBSA struct {
    thread_ID,
    socket parameters,
    other fields
};
```

objects “points to” by pointer parameters

Figure 4-8 Review of the Data Object
When the host state machine receives notification through the shared memory header that a data package from the EBSA-285 exists in the shared memory, it retrieves the thread ID field from the data package and compares it with the thread ID field inside the data package pointed to by the Service Data Elements in the wait list. When the shared memory thread ID matches the thread ID pointed to by the Service Data Element, the host state machine signals the event field in the particular Service Data Element to wake up the sleeping application pipe thread. Next, the application pipe thread copies the entire data package from the shared memory to the pipe buffer and sends the pipe buffer to the DLL. Figure 4-9 illustrate how the thread IDs are compared and the operations of the service for the data package return trip.
4.4 Simulation of the Polling Protocol on Windows NT

The simulation of the protocol enables us to envision the data path from the Host application to the EBSA-285 socket function. Since EBSA-285 with Linux was not operational at the time, we simulate the EBSA-285 operations with a Windows NT program. In order to simulate the shared memory on the Windows NT, we allocated a memory block using a Windows memory-mapped file. Both the PCICom service and the program that simulates the EBSA-285 can access the memory-mapped file. The program that simulates EBSA-285, known as Eside, contains the EBSA-285 part of the state machine mechanism that spawns threads to perform the actual socket call.

Instead of retrieving the shared memory address through the Mapmem driver, the PCICom service performs three Win32 function calls. These are the CreateFile, CreateFileMapping and MapViewOfFile. These three functions allow the service to create a shared memory space in the physical memory and map its address into the service’s virtual address space. The host state machine can access the emulating shared memory just like the shared memory on EBSA-285.

The Eside maps the emulated shared memory space by calling MapViewOfFile function. It communicates with the host by the EBSA side of the host-EBSA-285 communication protocol through the emulated shared memory. Figure 4-10 presents the elements of the Eside program.

When the Host side informs the EBSA-285 of a data package in the shared memory through the shared memory header, the Eside program spawns a new thread called the “function multiplexing module.” This thread copies the data package from the shared memory into a buffer and then passes the buffer on to the socket calling functions. There is a
socket calling function for each actual socket function. They perform socket calls with the information in the buffer. Figure 4-11 illustrates how the `recv()` socket call utilizes the data package fields in the buffer.

![Diagram](image)

**Figure 4-10 Operations of the EBSA-285 Emulation**

After the socket function call, the Function Multiplexing module enqueues the data package, which is located in the buffer, into the return queue. The Return QCnt value is incremented by one and the Function Multiplexing module terminates. The EBSA-285 state machine copies the data package to the shared memory and then dequeues and deletes the buffer.

```
Objects stored in the buffer created by the Function Multiplexing Module for the Recv function

int thread_ID
int type_ID
int return value
Socket s
Char *Buf
int len
int flag

The size of the Buf determines by the len field
recv(Socket s, char* buf, int len, int flag);
```

**Figure 4-11 Illustration of how the data return from the recv socket call is already serialized for the trip back to the host**
4.5 Assessment of the Polling Protocol

The polling protocol provides a simple approach for communication between the host and the EBSA-285. The protocol does not need any interrupt mechanisms or to deal with DMA operations. It also does not have to deal with thread and interrupt priorities. However, this protocol presents much inefficiency regarding the use of system resources.

In the polling protocol, the CPU continuously polls the status registers in the shared memory header. This decreases the CPU’s ability to perform other tasks during the time when the polling protocol is running. Also, if the OS preempts the state machine thread, the polling protocol on the host will sleep and the Host-EBSA-285 communication will stop. The communication will not resume until the state machine thread is swapped back in.

Another inefficiency of this protocol is the number of times the data package must be physically moved. In the DLL, the serialization of the data package is the first physical movement of the data. The data package is again moved when it passes through the dedicated pipe from the DLL to the buffer in the application pipe thread in the PCICom service. From there, the data package is moved to the shared memory. The EBSA side must move the data again to another buffer in order to free the shared memory for other process. Finally, there is an actual network data movement during the system calls.

In the returning path, the EBSA-285 state machine copies the data package to the shared memory. From there, it is copied to the pipe buffer in an application pipe thread. The data package then returns to the DLL through the pipe and then deserialized to the original buffer. So there are four copies in the returning trip. The number of data package copies must be decreased in order for the Windows-EBSA-285 communication efficiently.
Chapter 5 The Interrupt Driven Protocol

In the interrupt driven protocol, the EBSA-285 communicates with the host by interrupting the host. This allows the host’s CPU to concentrate on other tasks instead of diverting processing time to polling the shared memory. However, the host continues to communicate with the EBSA-285 through the shared memory registers. So the SA-110 still polls on a shared memory register through the polling protocol. This creates a non-symmetrical protocol in which the host operates with an interrupt driven protocol that differs from the EBSA-285’s polling operation.

Besides the difference in the communication mechanism, another significant change between the polling protocol and the interrupt driven protocol is the division of shared memory into multiple blocks. Previously, only one data package could exist in shared memory at any given time. Consequently, the EBSA-285 had to move the “current” data package out of shared memory and into a separate buffer before the host would be allowed to place another data package into the shared memory region. By dividing the memory into blocks, each block can hold a data package and the EBSA-285 can directly work on the shared memory.

In the interrupt driven protocol, the data transfer is divided into two distinct protocols – i.e., a writing protocol and a reading protocol – on the host side. After the host finishes writing to the shared memory, the EBSA can access the shared memory and read from it. This protocol is known as the Host Write EBSA Read, or simply as the “Host Write” protocol. If the EBSA-285 writes to the shared memory, the host reads what EBSA-285 has
written. This protocol is called EBSA Write Host Read or simply as the “Host Read” protocol. Both protocols operate independently from each other and concurrently. On the host, these two protocols share the same interrupt mechanism.

The EBSAsock functions and data structure remain the same in the interrupt driven protocol. The EBSAsock functions continue to fill in the EBSAsock data structure. However, the “funcHandl” function in the EBSAsock DLL accesses a Windows Device Driver instead of connecting to the PCICom service through a named pipe. No changes were required to initialize and terminate the protocols.

5.1 Interrupt on Windows 2000

The interrupt system for a PC consists of the interrupt signals from PCI, ISA and other bus devices that converge at a master interrupt controller. These signals may be attached directly to the master interrupt controller, or they may be routed through other interrupt management devices – e.g., a Programmable Interrupt Router and/or a “slave interrupt controller”. Regardless of how these interrupt signals reach the master interrupt controller, the master interrupt controller is the only device that asserts the processor’s interrupt request line(s). The processor then queries the master interrupt controller to obtain the interrupt vector value that corresponds to the device that is currently asserting its interrupt signal [17]. In the Windows OS, the interrupt dispatcher, a module of the kernel that acts as a switchboard between the interrupt value and the interrupt service routine, handles the interrupt requests [18].

Once the processor receives the interrupt vector value, it uses the vector value to raise the processor’s interrupt request level (IRQL). It is important to understand three terms that are related to IRQL.
1. **Preemptable** means that a context switch occurs after the current running thread’s quantum is exhausted or when a higher priority thread is available for execution. The current thread is placed in a ready queue[19].

2. **Dispatchable** indicates that if the current thread is blocked or is waiting for a signal, the current thread will be placed in a wait queue[20].

3. **Interruptible** indicates an interrupt request with the higher priority than the present CPU priority will be serviced by the CPU first. [19]

The following figure demonstrates the IRQL of a processor:

![Figure 5-1 The IRQL levels of a processor operating on Windows NT/2000](image)

The IRQL ranks execution units by priority. Ordinary execution threads execute at the lowest level. The system dispatcher – i.e., the Windows scheduler – operates at the dispatch level. The hardware interrupts operate at IRQL levels 11 through 26, and system interrupts occupy levels 27 through 31.

After the interrupt dispatcher raises its IRQL, the interrupt dispatcher locates the routine that will handle a particular IRQL through an interrupt dispatch table (IDT). This routine, called the handling routine, calls the interrupt service routines (ISR) that are
associated with the particular IRQL. The operations of the interrupt dispatcher and a map of
the handling routine associates with an IRQL inside the IDT are presented in Figure 5-2.

![Figure 5-2 Operations of the Windows Interrupt and IDT](image)

The ISR belongs to a particular device driver. Its association with a certain IRQL is
determined by the Windows OS. Since an interrupt signal can be shared by multiple PCI
functions on the PCI bus, the first thing the ISR must do is determine which device is
generating the interrupt. If the ISR determines that its device is asserting its interrupt request
line, the ISR tells the device (in some device-specific manner) to deassert its interrupt request
line. After the ISR has determined the reason why its device asserted its interrupt request
line, the ISR registers the name of a deferred ISR completion callback routine – i.e., a
“DpcForIsr” routine – with the Windows kernel and then returns to the handling routine. If
the interrupt doesn’t belong to the ISR’s device, the ISR ignores the interrupt and simply
returns to the handling routine. The handling routine sequentially calls all of the ISRs that
have the same IRQL so that each ISR has a chance to service the interrupt request. After that, the interrupt dispatcher lowers the IRQL for the processor to the original level.

It is important for the ISR to be as short as possible because a high level IRQL masks out all lower IRQL interrupts. That is why only the most critical operations should be performed inside the ISR. All other interrupt related operations for a driver should be performed within the DpcForIsr routine. The DpcForIsr routine operates at the lower “dispatch level” where other hardware interrupts (IRQL 11-26) are not blocked. Note, however, that the DpcForIsr, like the ISR, is not dispatchable or preemptible. Therefore, the ISR and its corresponding DpcForIsr routine cannot be context switched out (e.g., they cannot be put to sleep, nor can they be synchronized with other processes). Furthermore, paging cannot work when the IRQL is greater than or equal to “dispatch level” since paging is performed on Asynchronous Procedure Call (APC) level (level 2). APC also perform various tasks in the NT kernel, such as communication between device drivers.

5.2 The Shared Memory

Each partition of the shared memory contains one data package. The size of the shared memory should be the “common” message size plus the EBSAsock structure overhead. According to the network traffic analysis by Jonathan Kay and Joseph Pasquale [2], 90% of LAN traces are UDP messages and 10% are TCP messages. The size distribution of the UDP messages is bimodal: 86% are less than 200 bytes long while 9% are approximately 8 K bytes long. 86% of the TCP messages sizes are less than 200 bytes long.

5.2.1 The shared memory blocks

The shared memory blocks are divided into two different sizes, 8 K bytes and 256 bytes, to conform to the results of Kay’s and Pasquale’s research [2]. We originally defined
the small block to be 128 bytes for the data packages of EBSAsock socket calls such as bindE and connectE. These data packages contain a sockaddr structure in addition to the EBSAsock structure that contains the socket parameters and communication information. The total size of these data packages would be around 80-100 bytes – well within the 128-byte limit. Since most of the network messages are less than 200 bytes, it is prudent to create block size that can fit these messages.

The number of blocks is based on the total available shared memory size. The total shared memory size for the EBSA-285 when directly plugged onto the host PCI bus is 2 M Bytes. This memory is divided into 256 8 kilobytes (KB) blocks. One 8 KB block, which we denote as a big block, can be further divide into 32 blocks containing 256 bytes each, which I denote as a small block. The first four big blocks are subdivided into small blocks, so there are a total of 128 small blocks and 252 big 8K blocks. Note, however, that the first small block is reserved for the shared memory header; so in actuality there are only 127 small blocks available for data transfer.

5.2.2 Shared memory headers

Since both the host read and write protocols can simultaneously communicate across the shared memory, two memory headers – one for each protocol – exist to serve the communication and data transfer. Figure 5-3 shows the registers and their byte size for both headers.
The Host Write EBSA Read header supports the host write protocol. This header contains three fields, Host Write Host Status Register, the Type field, and the Host Write Number field (which indicates the shared memory block). The host possesses the write permission to these fields but EBSA-285 can only read them. The “Host Write Host status register” provides the bits for the host to write (and EBSA to read) during the host write handshaking sequence. The EBSA-285, as mentioned previously, communicates with the host via the interrupt mechanism, which replaces an EBSA write-Host Read status register. The “block type” register specifies the block type, where a 0 signifies a small block and a 1 represents a big block. The “Host Write Number” represents the block number, which supports up to 4 thousand blocks.

The EBSA Write Host Read header facilitates communication for the EBSA write protocol. It mirrors the Host Write EBSA Read header with the similar registers. The “EBSA Write Host status register” provides the bits for the host to handshake with EBSA while EBSA signals the host through the device interrupt pin during the EBSA write protocol. The EBSA informs the host of the location of the data package through the block type register and the EBSA Write block number register.

![Figure 5-3 Registers of the shared memory header](image)
The current protocol does not support the transfer of data packages larger than 8 KB. However, the next generation protocol will use the DMA engine on the EBSA-285 to transfer huge data packages. So this protocol becomes an intermediate communication scheme until the next generation protocol is created.

5.2.3 Shared Memory Bit Mask.

The host contains a rudimentary shared memory manager to control the shared memory. Two bit masks, one for each block size, indicate the availability of the blocks – i.e., a “1” indicates a block that is currently being used and a “0” indicates a block that is currently available for use. The bit masks, therefore, are initialized to zero since all the shared memory blocks started out empty.

Whenever a thread wishes to copy a data package to the shared memory, it passes the data package size to the shared memory manager’s “SharedMemoryAlloc” function to request a shared memory block. Since multiple threads can simultaneously request a block number, a spin lock, called the MemBlockLock, controls the access to the bit mask. The spin lock allows only one execution unit on a multiprocessor (SMP) system to access the controlled data object. Once an execution unit enters a spin lock, other units that desire to pass through the spin lock item must “spin” until it exits the spin lock. The spin lock permits proper sharing of data objects across a multi-processor system.

Once a thread enters the spin lock, it begins to scan for a “0” bit within the bit mask that corresponds to the desired memory block size (i.e., either 256 bytes or 8KB). Once it encounters a zero, it switches the bit value to one and returns the following items of information to the requesting thread: a) the address of the shared memory block, b) the block number, and c) the block type. When the host desires to free a particular memory block, it
passes the block number to the shared memory manager’s “SharedMemoryDealloc” function. This function reverts the block’s bit position in the bit mask from one to zero. Naturally, the bit mask is protected by the MemBlockLock spin lock.

5.3 The Host Write Protocol

As presented in Figure 5-4, a thread that desires to transmit data packages to EBSA-285 must first acquire a memory block within the shared memory region. Once a thread acquires a shared memory block, it can write to this block at its own leisure. However, it can only access the block it is authorized to and not other blocks. This feature allows multiple threads to write to the shared memory independently, which creates simultaneous access of the shared memory from the host.

5.3.1 Contention for Access to the Shared Memory Header

After a thread finishes copying its data package to the shared memory, it notifies the EBSA-285 through the Host Write shared memory header so the EBSA-285 can control the
shared memory block. Since multiple threads can concurrently access the shared memory header, any access to shared memory header becomes a critical section. Therefore, an atomic thread synchronization object must be used to control the shared memory header.

A spin lock around the critical section was the obvious choice to safeguard the shared memory header. It provides symmetrical multiprocessor support (SMP) while other thread synchronization objects support only one processor. Note, however, that all of the “spinning” threads have an equal chance of entering the critical section. So if many threads attempt to gain control of the shared memory header, a given thread may potentially wait for a long time because the spin lock by itself doesn’t provide a “first come, first served” provision.

In order to ensure that the order of arrival equates to the higher priority to access the critical section, a queue for the contending threads was implemented. The order of accessing of the critical section is thus determined by the Write Queue. Also, a semaphore is used in conjunction with the Write Queue to synchronize the critical section. However, the enqueuing operation itself becomes a contention point for the threads. In addition, a semaphore doesn’t have multiple processor support. As a result, a spin lock must also be used to protect the functions that access to the Write Queue and the semaphore. Figure 5-5 presents the flow chart on the synchronization of the threads to access the shared memory header, and the handshaking between host and EBSA-285. This flow chart contains within the 3rd block of Figure 5-4.
5.3.2 Handshaking of the Host Write Protocol.

Once a thread has the exclusive access to the Host Write shared memory header on the host side, it can handshake with the EBSA-285 to allow the co-host to control the shared memory block. As previously mentioned, the host communicates with the co-host by setting bits on the Host Write Host Status Register, and the EBSA responds with interrupts on the host.

Figure 5-5 Flow chart of the synchronization of the Critical Section.
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Host updates the block type and block number on the Host Write Header</td>
</tr>
<tr>
<td>2</td>
<td>Sets WRITE_FINISH bit</td>
</tr>
<tr>
<td>3</td>
<td>EBSA retrieves the block number and type from the Host Write Header. EBSA now has the control of the particular shared memory block</td>
</tr>
<tr>
<td>4</td>
<td>EBSA Signals H_WRITE_FINISH_ACK</td>
</tr>
<tr>
<td>5</td>
<td>Clears WRITE_FINISH bit</td>
</tr>
<tr>
<td>6</td>
<td>EBSA Signals H_WRITE_CLEAR_ACK</td>
</tr>
<tr>
<td>7</td>
<td>Upon reception of the WRITE_CLEAR_ACK, the Host thread can exist the critical section</td>
</tr>
</tbody>
</table>

**Figure 5-6 Handshaking of the Host Write Protocol**

In Figure 5-6, the signals from the host side are the bits in the Host Write Host Status Register, and the signals from the EBSA-285 are interrupt signals. The host side of Figure 5-6 represents the step “Handshaking With EBSA-285” in Figure 5-5. The EBSA-285 side of Figure 5-6 represents the responsibilities of EBSA-285 in the Host Write Protocol.

Since the Host communicates with EBSA-285 through the polling mechanism, handshaking responses from EBSA-285 must be provided when the host sets a bit and when the host clears a bit. Otherwise, race conditions might develop. For example, if a thread on the host clears the WRITE_FINISH bit and then exits the critical section, another host thread could enter the critical section and set the particular bit before the co-host polls the Host Write Host Status Register.
5.4 Host Read EBSA Write Protocol

Once the EBSA-285 is in control of a shared memory block, it performs the appropriate socket function call based on the information provided by the data package inside the shared memory block. After the socket call, the EBSA-285 stores the result(s) in the same shared memory block and then returns control of the memory block back to the host.

5.4.1 Handshaking of the Host Read protocol

The Host
(Performed Inside DpcForIsr)

Step 1. EBSA updates the block type and number on the Host Read Header

Step 2. EBSA Signal E_WRITE_FINISH

Step 3. The DPCforISR copy the block type and number field of the Host Read Header and signal the host thread that is responsible for the particular shared memory block.

Step 4. Sets HOST_READ_PROGRESS bit

Step 5. EBSA Signal H_READ_ACK

Waits for HOST_READ_PROGRESS bit to be clear. End of the Host Read Protocol

The EBSA-285

Step 1. EBSA updates the block type and number on the Host Read Header

Step 2. EBSA Signal E_WRITE_FINISH

Figure 5-7 Handshaking of the Host Read Protocol

Since the co-host communicates with the host via the interrupt mechanism, the handshaking responses of the host are performed inside the DpcForIsr callback function. The responses involve setting and clearing the HOST_READ_PROGRESS bit on the EBSA Write Host Status Register of the Host Read Header. Figure 5-7 demonstrates the handshaking mechanism the host read protocol.
5.4.2 Reading and deallocating the shared memory block.

Figure 5-8 Host thread operations after a thread had been “awaken”

Once control exits the Host Write critical section, the host thread will typically sleep until the DpcForIsr callback function wakes it up. Note, however, that the host thread might not have a chance to sleep if the EBSA-285 can perform the socket function call in the time frame between the host thread exiting the critical section and it falling asleep. This situation could occur if: a) the host is executing a large number of threads, and b) those threads manage to occupy a sufficient number of execution time slots (a.k.a., “jiffies”). Regardless of whether the host thread actually sleeps, the host thread now copies the data package from its shared memory block to the host memory space. Finally, the thread calls the shared memory manager to return the particular block back to the blocks of free memory, as explained in section 5.2.3. The Figure 5-8 points out the steps after a host thread is “awaken” by the DpcForIsr.
Chapter 6 Implementation and Simulation of the Interrupt Driven Protocol

The previous chapter discusses the design of the interrupt driven protocol, this chapter explores the implementation and the simulation of the interrupt driven protocol. The WinHost device driver contains the implementation of the interrupt driven protocol on Windows 2000 that facilitates the communication between the host and the co-host. As presented in the previous chapter, the “sockFunct” function inside the EBSAsock DLL allows the calling thread to access the WinHost device driver. The device driver itself provides the mechanism to synchronize various threads when transferring data to and from the EBSA-285 shared memory. The term “device” refers to a hardware component operating on a system bus, such as a PCI device or a USB device. The “driver” represents a software module that serves as an intermediary communication channel between the OS and the device.

6.1 Overview of Window 2000 Device Driver

This section presents an overview of the Windows 2000 Device Driver components most relevant to the WinHost driver. For further studies about the Window 2000 device driver, please refer to “Microsoft Windows Driver Model” by Oney[21], and “Writing Kernel Mode Device Drivers for Windows 2000/WDM” [19].

6.1.1 Basic terms

The Windows 2000 kernel components that support the Winhost device driver implementation are the Plug and Play (PNP) manager, the I/O manager, and the Hardware
Abstraction Layer (HAL). During system start up, the PNP manager scans the buses and ports for new hardware devices that were inserted into the system since the previous system started up. If it detects a new device, it searches through a list of drivers in the system that matches the device. The drivers in the system must first be manually installed. There are various methods of installing a driver into the system, depending on the type of device driver. The WinHost device driver is installed through the device manager console.

The I/O manager performs the actual loading of the driver once the PNP manager locates a matching driver for a detected device. After the driver is loaded, the I/O manager manages the operation of the device. It provides the APIs that allow the driver to communicate and synchronize with other drivers and interface with the applications. The section of the I/O manager that provides the interface between the user application and the device driver is known as the I/O system service.

The HAL allows the driver to interact directly with the hardware itself. In all versions of Windows NT before Windows 2000, the only way for a driver to communicate with a device was through the HAL. However, with the Windows 2000 driver stack, which we will explain in section 6.1.2, device drivers normally communicate with the hardware device itself through one or more lower layer drivers. For instance, a driver that desires to communicates with devices on the PCI bus, e.g. the WinHost driver, must do so through the PCI bus driver. However, the WinHost driver uses a couple of deprecated HAL functions because of the limitations of the WinHost ISR to communicate with the lower layer driver due to its high IRQL value.
6.1.2 The Driver Stack

A driver stack is a layer of device drivers working together to achieve the transfer of data from the operating system to a particular device. A driver stack normally consists of a functional driver, bus driver and one or more filter drivers. The WinHost driver represents the functional driver and the PCI bus driver represents the bus driver. Currently, no filter drivers are needed for our device stack.

When the WinHost driver desires to communicate with the EBSA-285 configuration space, it transmits a signal to the PCI bus driver through the I/O manager. The PCI bus driver directly communicates with the 21554 or EBSA-285 based on requests from the function driver. The only supported HAL functions for the functional drivers are read/write functions to or from a particular memory space or I/O ports.

Figure 6-1 presents the kernel modules and the driver stack related to the WinHost driver. It displays how the WinHost driver can access the hardware through the PCI bus.

Figure 6-1  The Windows Driver Stack and Kernel Modules

1. WinHost normally interacts with the hardware through the PCI bus.
2. WinHost can access the memory and I/O space directly through HAL.
driver. If the WinHost driver desires to read or write from a memory or I/O space, it can directly access the HAL.

6.1.3 General Driver Routines

A windows driver supports several routines that are accessible from outside of the driver. The WinHost driver accessible routines are the DriverEntry, AddDevice, DispatchPnP, ISR, DpcForIsr, and the application accessible routines, which include the write() and ioctl() routines.

The DriverEntry routine is called when the driver is first loaded. It performs the driver wide initialization and exports all other driver routines that can be accessed from outside of the driver.

The PNP manager calls the AddDevice routine after either the 21554 or EBSA-285 is located on the PCI bus. The AddDevice routine creates the Functional Device Object (FDO), the data structure that is used to manage the device. It also attaches the FDO to the data structure that manages PCI bus driver, which is also known as the physical device object (PDO). For further details of PDO and FDO, please study the Windows Driver book by Oney [21]. The AddDevices routine also initializes the device variable and ensures that Win32 can access the device. The device variables can be regard as global variables for the device driver.

The PNP manager also guides the driver through a series of state transitions when starting and stopping the device. The DispatchPnp function provides the state machine for this purpose. Also, the WinHost driver obtains the EBSA-285 shared memory address from the SDRAM base address register (referring to section 2.2.2) inside this function.
The ISR and DpcForIsr handle the interrupt mechanism of the WinHost driver. Section 6.2 will explain the operations within each function and how these two functions communicate with each another. It will also demonstrate how the IPCForISR routine associates with each step of the Host Read and Write protocols. The non-interrupt driven section of the protocols, which includes the user space accessible functions write() and ioctl(), locates in section 6.3.

6.2 WinHost Interrupt Functions and Data Structure

The interrupt mechanism of WinHost driver consists of the ISR and ISRForDpc functions. These two functions communicate with each other by an interrupt buffer.

6.2.1 ISR

The first responsibility of the ISR is to detect whether the interrupt originated from a device driven by the WinHost driver (either the EBSA-285 or 21554 board). This is due to the fact that PCI interrupts are shareable among PCI devices. If the EBSA-285 directly connects to the primary PCI bus, the WinHost driver can determine whether the interrupt originate from it by reading the 32-bit doorbell register. The doorbell register is located at offset 60H of the EBSA-285 PCI configuration space. It is initially set to 0 during system start up. When an EBSA-285 module sets one or more bits in the doorbell register, the EBSA-285 will issue an interrupt signal to the host. Currently, the 21554 board cannot translate any interrupt signal from the secondary PCI bus to the host so the WinHost driver does not provide support to detect any interrupt originated from the 21554 board.

The first BAR address of the EBSA-285 PCI configuration space (Presented in Figure 4-6), CSR Memory Base Address, should provide the PCI based address of the EBSA-285 configuration space to the host according to the 21285 Core Logic for SA-110
Microprocessor Datasheet [22]. Unfortunately, the CSR Memory Base Address does not provide the correct configuration space base address for the host. Therefore, I resorted to the deprecated HAL function HalGetBusDataByOffset() to access the PCI configuration space of the EBSA-285.

If none of the doorbell register bits are set, the signaled interrupt is generated by another device sharing the same IRQL as the Winhost. The ISR would simply return a FALSE value to indicate that the signal is destine for another driver. Otherwise, the ISR would cancel the interrupt by writing a “1” to the doorbell register bit that is set. The HAL function that write to the EBSA-285 configuration space is called HalSetBusDataByOffset(), which is also a deprecated function not supported by Windows 2000.

Once the ISR cancels the interrupt, it saves the critical information to the interrupt queue buffer. The critical information always includes the doorbell register. A bit on the doorbell register corresponds to an interrupt signal from the EBSA-285. In our implementation of the Host Write Protocol, the EBSA-285 signals H_WRITE_FINISH_ACK and H_WRITE_CLEAR_ACK. In the Host Read Protocol, the EBSA-285 would signal E_WRITE_FINISH and H_READ_ACK. The following figure presents the EBSA-285 signals and its corresponding bit position in the doorbell register.

<table>
<thead>
<tr>
<th>Doorbell register</th>
<th>Signal</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0001</td>
<td>H_WRITE_FINISH_ACK</td>
<td>Host Write</td>
</tr>
<tr>
<td>0x0000 0002</td>
<td>H_WRITE_CLEAR_ACK</td>
<td>Host Write</td>
</tr>
<tr>
<td>0x0000 0100</td>
<td>E_WRITE_FINISH</td>
<td>Host Read</td>
</tr>
<tr>
<td>0x0000 0200</td>
<td>H_READ_ACK</td>
<td>Host Read</td>
</tr>
</tbody>
</table>
Next, if the ISR receives E_WRITE_FINISH signal, it would save the block type and the block number the EBSA Write header fields into the interrupt queue buffer. Finally, ISR must request for the execution of the DpcForIsr routine of the WinHost drive before returning TRUE to indicate that it had handled the interrupt.

6.2.2 Interrupt Queue Buffer

The Interrupt Queue Buffer allows the ISR to store critical interrupt information for processing by the ISRForDpc. An Interrupt Queue Buffer element consists of the following fields.

- **block_type**: The ISR stores the block type of the host read header for the ISR when the signal from EBSA-285 is E_WRITE_FINISH.

- **location**: The ISR stores the block number from the host read header when the signal is E_WRITE_FINISH.

- **doorbell reg**: Contains an exact copy of the doorbell register. Which indicates the signal that triggers the interrupt.

The interrupt queue buffer is a circular queue buffer. The ISR index into the buffer through the eight bit unsigned field InterruptQCnt. The DpcForIsr retrieves information from the buffer through the eight bit unsigned field DpcQCnt. These two fields are device variables initialized to zero inside the addDevice routine. Once the ISR has placed the critical information into the buffer element indexed by the InterruptQCnt field, it increments the InterruptQCnt.

The DpcForIsr function contains a loop that compares the difference between the DpcQCnt and the InterruptQCnt. When these two fields are not equal, the DpcForIsr retrieves the information from the interrupt queue buffer element index by the DpcQCnt and
perform a task based on the interrupt signal, which will be explained in section 6.2.1. Finally, the DpcQCnt is incremented before the end of the loop. Figure 6-2 illustrates the relationship of the interrupt queue buffer with both the DPC and the DpcForIsr.

In the interrupt signal handling, the DpcQCnt is incremented before the end of the loop. Figure 6-2 illustrates the relationship of the interrupt queue buffer with both the DPC and the DpcForIsr.

**Figure 6-2** The interactions of interrupt queue buffer with both ISR and DPCForIsr

### 6.2.3 DpcForIsr

As mentioned in the previous section, the DpcForIsr contains a loop that handles the interrupt signals. The loop contains a switch case statement for each signal from EBSA-285, which is based on the doorbell_reg field of the queue element. As mentioned in Table 6-1, there are four possible signals from the EBSA-285. These are the H_WRITE_FINISH_ACK, H_WRITE_CLEAR_ACK, E_WRITE_FINISH and H_READ_ACK signals.

The H_WRITE_FINISH_ACK signal is issued in the Host Write protocol. Upon receiving this signal, the DpcForIsr responds by clearing the HOST_WRITE bit on Host Write Host Status Register. This action corresponds to step 6 of Figure 5-6. Please refer to section 5.3 for a complete look into the Host Write protocol.
The H_WRITE_CLEAR_ACK signal is also issued in the Host Write protocol. Figure 6-3 illustrates the actions performed in the IsrForDpc upon the reception of this signal.

**Figure 6-3 Operation inside IsrForDpc when receives H_WRITE_CLEAR_ACK**

Two arrays of Windows kernel events provide communication from the ISRForDPC and the threads that possess a shared memory block. These two arrays correspond to the two sizes of shared memory blocks and the size of each array equals to the number of blocks for each type. The declarations of these two arrays are:

KEVENT s_block_event[SMALL_BLOCK_NUMBER]; // small block events

KEVENT b_block_event[BIG_BLOCK_NUMBER]; // big block events
Once a thread acquires a shared memory block, it also becomes the owner of the particular kernel event. For example, if a thread possess small block 5, it also has the sole control over s_block_event[5].

After the EBSA-285 completes its operations on the shared memory block, it fills in the block number and the block type in the EBSA write header and notifies the host with the E_WRITE_FINISH signal. Upon receiving this signal, the IsrForDpc function retrieves the block number and the block type from the interrupt queue buffer. With these two pieces of information, it can determine which kernel event array entry to signal. After it signaled the event, the IsrForDpc sets the HOST_READ_PROGRESS bit on the EBSA Write Host Status Register. These operations correspond to Step 3 and 4 of Figure 5-7.

The H_READ_ACK signal is also issued in the Host Write protocol. The IsrForDpc responds by clearing the HOST_READ_PROGRESS bit on the EBSA Write Host Status Register. This action corresponds to Step 6 of Figure 5-7. Please refer to section 5.4 for a complete look into the Host Read Protocol.

6.3 Application Access Routines

The application access routines includes the write( ) function and the ioctl( ) function.

6.3.1 write ( )

The “funcHandle” function inside the EBSAsock DLL passes the data package and the size of the data package to the write( ) function. Upon entering, to the write( ) function allocates a shared memory block and copies the data package to the shared memory (Figure 5-4, steps 1 and 2). If the semaphore has not been taken, it acquires the semaphore and enters the critical section (See Figure 5-5). It then fills the Host Write block number and block type
and sets the HOST_WRITE bit (Step one of Figure 5-6) before returning the block number and type to the “funcHandle” function.

If the semaphore is already taken, the block type and block number is queue into the Write Queue. Finally, the block number and the block type are returned to the “funcHandle” function.

6.3.2 ioctl()

Next, the “funcHandle” function calls the ioctl( ) function with the data package, block number and the block type. With the block number and the block type, the ioctl( ) function waits upon the kernel event that associates with the particular block type and block number. After the signaling of the kernel event, the ioctl( ) function must reset the state of the event. Than it copies the data package from the shared memory block to the host memory space and deallocates the shared memory block itself. These steps are shown in section 5.4.2 and illustrated on Figure 5-8.

6.4 Simulation of the Interrupt Protocol

The simulation of the interrupt protocol is performed on the Windows 2000 and the EBSA-285 card plug directly on the host PCI bus.

The responsibility of the EBSA-285 co-host includes communicating with the host through the interrupt driven protocols and performs the actual socket call based on the data package from the host. For the polling protocol, the simulation was done entirely in the Windows host, as presented in chapter 4 section 4.4. However, the simulation of the interrupt driven protocol must originated from a program running on EBSA-285 board since only the EBSA-285 itself can generate an interrupt. Furthermore, the 21554 board cannot transfer the interrupt signal from the EBSA-285 to the host, the EBSA-285 must be plugged directly on
to the host PCI bus to interrupt the host. Since the Linux OS cannot co-exist with the Windows 2000 host, the EBSA-285 section of the interrupt protocol and the actual socket function calls are simulated on the Windows 2000 host. Figure 6-4 lays out the software architecture of the interrupt protocol simulation.

![Diagram of software architecture](image)

**Figure 6-4** The software architecture of Interrupt Protocol simulation

### 6.4.1 EBSA-285 interface

Both the Winhost driver and the EBSA-285 simulation program, IntPoll, access the shared memory. The interface between the Winhost, the shared memory and the interrupt had been explained in Chapter 5 and Chapter 6 (sections 6.1 through 6.3). The IntPoll program emulates the EBSA-285 side of the handshaking for the Host Read and the Host Write protocol (which presented in Figure 5-6 and Figure 5-7) and performs the actual socket call.

Because the IntPoll program performs the actual socket function call on Windows 2000, it must be located in the user space. Thus, it must first retrieve the virtual address to the
shared memory through the mapmem driver. Please see section 4.2 for the coverage of the mapmem driver.

![Flow chart of the EBSAInterrupt program](image)

Figure 6-5 Flow chart of the EBSAInterrupt program

The Mapmem driver allows the IntPoll program to access the shared memory but not the doorbell register. Only a program executing on the EBSA-285 can write to the doorbell register to generate an interrupt. Therefore, the IntPoll program must notify the module running on EBSA-285, the EBSAInterrupt program, to write to the doorbell register. A 32-bit field, call the IntSim register, located right after the shared memory header facilitates the communication from the IntPoll program to EBSA-285 doorbell registers. This field is initially set to zero. When the EBSA-285 side of the protocols needs to set a signal on the host side, the IntPoll program sets a bit that correspond to a particular signal in the IntSim register. The EBSAInterrupt consist of an infinite loop. The operation inside the loop are presented in Figure 6-5.
6.4.2 IntPoll Program

The EBSA-285 simulation program for the interrupt driven protocols on the Windows 2000 is similar to the simulation program for the polling protocol (Figure 4-10). However, there are a few differences in the IntPoll program. First of all, because the interrupt protocol is divided into the Host Read and the Host Write protocols, each protocol executes in its own thread. Secondly, the host read protocol thread retrieves only the block number and block type from the Host Write header. These two fields are passed on to the function multiplexing module. The specific shared memory address for a particular memory block is calculated and passed into the function that performs the actual socket call. The actual socket call parameters are directly accessed on the shared memory during the socket call and any function outputs are directly placed in the shared memory. Figure 6-6 steps through the operation of the IntPoll program up until the actual socket call.
Figure 6-7 Operation of the IntPoll simulation program after the socket call

After the socket call, the multiplexing module queues the block number and type into the return queue and increments the QCnt. The Host Read protocol constantly polls the QCnt. If QCnt > 1, the EBSA side of the Host Read protocol dequeues the block number and type from the Return Queue, place these two field in the EBSA Write share memory header and then decrements the QCnt. Figure 6-7 steps through the operations of the IntPoll program after the socket function call. For the Host Read protocol operations on the EBSA-285 side, please study Figure 5-7.
6.4.3 Testing the simulation

Currently, the EBSAsock functions we support are: socketE, bindE, connectE, acceptE, listenE, recvE, sendE, closesockE, gethostbynameE and getsocknameE functions. The socketE, the connectE and the sendE functions are sufficient to create a simple network client program to communicate with a network server program executing on a remote machine. This network client program, known as the “SendE” program, operates on a PC with Windows 2000 OS and has an EBSA-285 card attach to the primary PCI bus. Also, the WinHost driver, the EBSAsock DLL and the intpoll program that simulates the EBSA-285 operation executes on the host. Finally, the “EBSA interrupt” program executes on the EBSA-285 to coordinate with the intpoll program. Figure 6-8 diagrams the components of the test platform with the SendE program.
The sendE program accepts the host name and the port number of the server process as input. Once the SendE program begins to execute, it establishes a TCP connection with the server program. After the TCP connection has been created, the SendE program prompts the user to type in a text string. This string is transmitted through our protocol simulation to the server program and displayed on the server computer’s console.

Debugging statements were placed within each of the intpoll socket call functions and the EBSAsock functions. Figure 6-9 presents the screen capture of the sendE program. The sendE program accepts the server host name (volans) and its port number (53765) as command line arguments. The first line indicates that the EBSAsock DLL is attached. The next two lines show that the sendE program calls the socketE and the connectE functions, which create a TCP connection with the server program running on volans.

The line “type in some data” prompts the user to type in a string. The following line, “hello world”, originates from the person performing the test. After the person types in the entire string, the sendE function is called to transport the string across the network to the server. The rest of the program consists of the user typing in more lines for transmission from the sendE program to volans.
Figure 6-9  The screen dump of the sendE program, which includes the DLL output

Figure 6-10 shows output on the intpoll program while executing the socketE program on Figure 6-9. The Intpoll function calls (see Figure 6-10) correspond to the EBSAsock function calls on Figure 6-9. Figure 6-11 shows the interrupt signals issued by the EBSA interrupt program on the EBSA-285 while executing the socketE program on Figure 6-9. The signals are presented in Table 6-1.
Figure 6-10  The screen dump of the intpoll program when executing the sendE program

Figure 6-11  The screen dump of the interrupt signals by the EBSA Interrupt Program on the EBSA-285
Chapter 7  Conclusion and Future Work

Both the polling protocol and the interrupt driven protocol provide communication between the host and the co-host. However, the interrupt driven protocol requires only one movement of a data package from the host to the actual socket call on the co-host. The polling protocol requires several physical movements of the data package from the application on the host to the socket function call in the co-host. Also, with the interrupt driven protocol, multiple threads can write or read from the shared memory blocks concurrently with minimal serialization.

The next generation communication protocol between the host and the co-host will utilize the DMA engine located on the EBSA-285. DMA data transactions will dramatically decrease the host processor utilization. However, a careful study must be performed to determine when to transmit data with the DMA engine versus the processor read/write of the shared memory. There are two goals for the study: 1) To ensure the maximum data throughput from the host to the co-host and vice versa and 2) to minimize the host processor utilization.

To complete the design of a protocol, a thorough study on error conditions must be performed. The host and the co-host must be capable of informing the other side when an error condition occurs. Also, both sides must coordinate to handle all error conditions. The reason that the error conditions have not been thoroughly studied is due to the constant changing of the protocol.
Finally, a board that can perform TCP/IP functionality will eventually supplant the EBSA-285 board. Currently, the CiNIC group members are looking into the Excalibur board by Altera. The board provides a PDL chip that can support an entire system. Hopefully, the TCP/IP stack and other network functionality can be programmed onto the chip itself. If that is the case, we will have a co-host that can out-perform the host CPU in TCP/IP stack processing.
Bibliography


Appendix A  EBSAsock structure declaration.

The declaration of the EBSAsock structure. The data structure is filled in the
EBSAsock functions and sent to the EBSA-285.

/*
* socketE.h
*
* Declares the EBSAsock data structure
*
*/

#ifndef SOCKETE
#define SOCKETE

#define NSIDE 1
#define LINE_SIZE 80
#define DWORD ULONG

/*
* definition of the socket calls,
* used as command name
* This socket calls are tie to Funct_calls
* table for function operation in NTside.h
*/
#define SOCKET_CALL 1
#define BIND_CALL 2
#define CONNECT_CALL 3
#define LISTEN_CALL 4
#define RECV_CALL 5
#define ACCEPT_CALL 6
#define SEND_CALL 7
#define CLOSE_CALL 8
#define GETHOSTBYNAME_CALL 9
#define GETSOCKNAME_CALL 10

// All the declaration must be done before this line
/*
* definitions of structures to hold the parameters
* of the socket calls
*/
typedef struct {
    int domain;
    int type;
    int protocol;
} SOCKET_TYPE; // for the function of socket()

typedef struct {
    int sockfd;
    struct sockaddr *my_addr;
    int addrlen;
} BIND_TYPE;    // for the function of bind()

typedef struct{
    int sockfd;
    struct sockaddr *my_addr;
    int addrlen;
} CONNECT_TYPE; // for the function of connect()

typedef struct{
    int sockfd;
    int backlog;
} LISTEN_TYPE; // for the function of listen()

typedef struct{
    int sockfd;
    struct sockaddr *my_addr;
    int addrlen;
} ACCEPT_TYPE; // for the function of accept()

typedef struct {
    int sockfd;
    int msg_offset; // void *msg;
    int len;
    unsigned int flags;
} SEND_TYPE; // for the function of sendto()

typedef struct {
    int sockfd;
    int msg_offset; // void *msg;
    int len;
    unsigned int flags;
} RECV_TYPE; // for the function of receive()

typedef struct{
    int sockfd;
} CLOSE_TYPE; // for the function of close()

typedef struct{
    char *name;
} GET_HOST_BY_NAME_TYPE;

typedef struct{
    int sockfd;
    struct sockaddr *my_addr;
    int addrlen;
} GET_SOCK_NAME_TYPE;
typedef int RETURN_TYPE; // for passing the return value
    // of the above function calls

#define SOCKET_CALL_TYPE_HEAD_SIZE 16

/*
 * definition of the structure to pass the socket call
 * information
 */
typedef struct{
    int type_id;
    DWORD pid;
    RETURN_TYPE return_val;
    union func_calls_info{
        SOCKET_TYPE socket;
        BIND_TYPE bind;
        CONNECT_TYPE connect;
        LISTEN_TYPE listen;
        ACCEPT_TYPE accept;
        SEND_TYPE send;
        RECV_TYPE recv;
        CLOSE_TYPE close;
        GET_HOST_BY_NAME_TYPE gethostbyname;
        GET_SOCK_NAME_TYPE getsockoptname;
    } func_calls;
} SOCKET_CALLS_TYPE; /* EBSAsock structure */

#endif
Appendix B  Declaration of shared memory header, shared memory blocks and communication signals between the host and EBSA-285

/*
 * ebsa285.h
*
 * Contains the signals for communication between the host and the
 * EBSA-285.
 * /
*/

#ifndef EBSA285
#define EBSA285

/* define the gcc types since part of the driver is a direct copy
 from gcc */
typedef UCHAR u8;
typedef USHORT u16;
typedef ULONG u32;

enum IOCTL_cmd {IOCTL_RETRIEVE_PCI_MEM, WAKEUP_WAITING, IOCTL_END};

/*
 * Interrupts from EBSA to host
 */
/* Host write, EBSA read occupies the first 8 bits of the
 * doorbell register
 */
#define H_WRITE_FINISH_ACK_ 0x00000001
#define H_WRITE_CLEAR_ACK_ 0x00000002

/* EBSA write, host read occupies the second 8 bits of the
 * doorbell register
 */
#define E_WRITE_FINISH_ 0x00000100
#define H_READ_ACK_ 0x00000200
#define SHARE_IOCTL 0x01000000
#define HOST_WRITE_FINISH 0x01

/*
 * Bits set and clear by the host on the status registers in
 * the share memory. Bits poll by the EBSA-285
 */

/* Host write EBSA read for location HOST_WRITE_STATUS */
#define HOST_WRITE_FINISH 0x01
/* Host write EBSA read for location HOST_WRITE_NUMBER_OF_BLOCKS */
#define SEND_BLOCK_SIZE 0x80

/* EBSA write Host read for location EBSA_WRITE_STATUS*/
#define HOST_READ_FINISH 0x01
/* EBSA write Host read for location EBSA_WRITE_E_STATUS_OFFSET - mirror of
* HOST_WRITE_NUMBER_OF_BLOCKS */
#define READ_BLOCK_SIZE 0x80

#define ONE_KILOBYTE 1024 /* define the size of a big and small blocks */

/* total size of the share memory */
#define TOTAL_SPACE 2097152 /* equals 8 K.. 8192 */
#define BIG_BLOCK_SIZE (8 * ONE_KILOBYTE) /* chosen because most non data function calls used less than 256 bytes*/
#define SMALL_BLOCK_SIZE 256

/* number of small blocks = (2* 8192/256) -1
* 8192/256 = Number of small blocks in a big block location
* 4 big blocks are subdivided into small blocks
* 1 block is use as the polling and information header. 2 times
SM_HEADER
*/
#define SMALL_BLOCK_NUMBER (2*(BIG_BLOCK_SIZE/SMALL_BLOCK_SIZE)-1)

/* Number of big blocks = total space / big block size
* 4 blocks are subtracted off, which is the number of big blocks use
* for small blocks */
#define BIG_BLOCK_NUMBER (TOTAL_SPACE/BIG_BLOCK_SIZE - 4)

/* the ceiling of BIG_BLOCK_NUMBER/8. EACH bit would be use to
* determine whether a block is used */
#define BIG_BLOCK_TABLE_SIZE (BIG_BLOCK_NUMBER/8 +
((BIG_BLOCK_NUMBER % 8)\n? 1 : 0))

/* the ceiling of SMALL_BLOCK_NUMBER/8. EACH bit would be use to
* determine whether a block is used */
#define SMALL_BLOCK_TABLE_SIZE (SMALL_BLOCK_NUMBER/8 +
((SMALL_BLOCK_NUMBER % 8)\n? 1 : 0))
// Define the number of Base Address Register are used in the PCI configuration space and the names of the used spaces

#define CURRENTLY_USED_BAR 3
#define CSR_MEMORY_BAR 0
#define CSR_IO_BAR 1
#define SDRAM_MEMORY_BAR 2

// Defining the offset of the PCI doorbell register
#define PCI_DOORBELL_REGISTER 0X60
#define BIG_BLOCK_BIT 0x80000000

typedef struct sharemem_header
{
    UCHAR status_reg;
    UCHAR block_type;
    USHORT location;
}SM_HEADER_TYPE;

/* Host or EBSA small and big block offset */
#define SMALL_SPACE_OFFSET 128
#define BIG_SPACE_OFFSET (2*BIG_BLOCK_SIZE)

typedef struct
{
    unsigned long mem_addr;
    unsigned long mem_size;
}EBSA285_TYPE;

#endif
Appendix C  The source code CD

The cdexplanation.txt file presents an overview of the directories in the CD. Please refer to that file for the source code contain in each directory.