PA 1243244

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

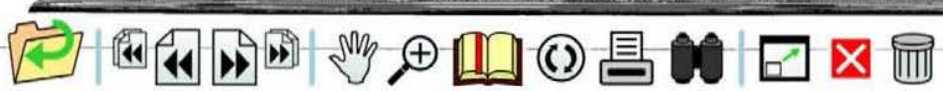
November 03, 2004

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: *60/061,809*
FILING DATE: *October 14, 1997*

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

M. K. HAWKINS
Certifying Officer



ALA00138383

DELL Ex.1031.001

A/Prov

PTO/SB/16 (11/95) (Modified 1997 Legalsoft)
Approved for use through 01/31/98. OMB 0651-0037
Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

(Large Entity)

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53 (b)(2).

Docket Number	ALA-001	Type a plus sign (+) inside this box	+

INVENTOR(s)/APPLICANT(s)

LAST NAME	FIRST NAME	MIDDLE INITIAL	RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY)
Boucher	Laurence	B.	Saratoga, California
Blightman	Stephen	E. J.	San Jose, California
Craft	Peter	K.	San Francisco, California
Higgin	David	A.	Saratoga, California

TITLE OF THE INVENTION (280 characters max)

INTELLIGENT NETWORK INTERFACE CARD AND SYSTEM FOR PROTOCOL PROCESSING

CORRESPONDENCE ADDRESS

Mark Lauer
6850 Regional Street,
Suite 250
Dublin

Tel: (510)556-3500
Fax: (510 803-8189

STATE	CA	ZIP CODE	94568	COUNTRY	USA

ENCLOSED APPLICATION PARTS *(check all that apply)*

- [X] Specification — *Number of Pages*: 130
- [X] Drawing(s) — *Number of Sheets*: included in Specificatio
- [X] Other (specify): Drawings are included within Specification

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT *(check one)*

- [X] A check or money order is enclosed to cover the filing fees
- [ ] The Commissioner is hereby authorized to charge filing fees and credit Deposit Account Number:

FILING FEE AMOUNT: $150.00

The invention was made by an agency of the United States Government or under a contract with an agency of the United States

- [X] No.
- [ ] Yes, the name of the U.S. Government agency and the Government contract number _____

Respectfully submitted,

SIGNATURE _____ Date October 14 1997

TYPED or PRINTED NAME Mark Lauer REGISTRATION NO. *(if appropriate)* 36,578

- [X] Additional inventors are being named on separately numbered sheets attached hereto

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

SEND TO: Box Provisional Application, Assistant Commissioner for Patents, Washington, DC 20231

P19LARGE/REV03

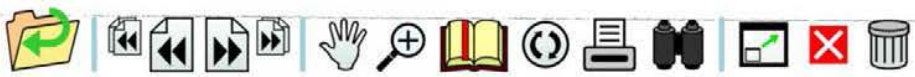
PROVISIONAL APPLICATION FOR PATENT COVER SHEET

(Large Entity)

INVENTOR(s)/APPLICANT(s)			
LAST NAME	FIRST NAME	MIDDLE INITIAL	RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY)
Philbrick	Clive	M.	San Jose, California
Starr	Daryl	D.	Milpitas, California

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

SEND TO: Box Provisional Application, Assistant Commissioner for Patents, Washington, DC 20231



CERTIFICATE OF MAILING BY "EXPRESS MAIL" (37 CFR 1.10) Applicant(s): **Laurence B. Boucher et al.**			Docket No. **ALA-001**
Serial No.	Filing Date	Examiner	Group Art Unit

Invention: **INTELLIGENT NETWORK INTERFACE CARD AND SYSTEM FOR PROTOCOL PROCESSING**

I hereby certify that this **PROVISIONAL PATENT APPLICATION, COVER SHEET & CHECK FOR $150.00**
(Identify type of correspondence)

is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 in an envelope addressed to: The Assistant Commissioner for Patents, Washington, D.C. 20231 on

October 14, 1997
(Date)

60061809.101497

Mark Lauer
(Typed or Printed Name of Person Mailing Correspondence)

(Signature of Person Mailing Correspondence)

EH756230105US
("Express Mail" Mailing Label Number)

Note: Each paper must have its own certificate of mailing.

P06A/REV02

INTELLIGENT NETWORK INTERFACE CARD

AND SYSTEM FOR PROTOCOL PROCESSING

Provisional Patent Application Under 35 U.S.C. § 111 (b)

Inventors: Laurence B. Boucher
Stephen E. J. Blightman
Peter K. Craft
David A. Higgin
Clive M. Philbrick
Daryl D. Starr

Assignee: Alacritech Corporation

1 Background of the Invention

Network processing as it exists today is a costly and inefficient use of system resources. A 200 MHz Pentium-Pro is typically consumed simply processing network data from a 100Mb/second-network connection. The reasons that this processing is so costly are described here.

1.1 Too Many Data Moves

When network packet arrives at a typical network interface card (NIC), the NIC moves the data into pre-allocated network buffers in system main memory. From there the data is read into the CPU cache so that it can be checksummed (assuming of course that the protocol in use requires checksums. Some, like IPX, do not.). Once the data has been fully processed by the protocol stack, it can then be moved into its final destination in memory. Since the CPU is moving the data, and must read the destination cache line in before it can fill it and write it back out, this involves at a minimum 2 more trips across the system memory bus. In short, the best one can hope for is that the data will get moved across the system memory bus 4 times before it arrives in its final destination. It can, and does, get worse. If the data happens to get invalidated from system cache after it has been checksummed, then it must get pulled back across the memory bus before it can be moved to its final destination. Finally, on some systems, including Windows NT 4.0, the data gets copied yet another time while being moved up the protocol stack. In NT 4.0, this occurs between the miniport driver interface and the protocol driver interface. This can add up to a whopping 8 trips across the system memory bus (the 4 trips described above, plus the move to replenish the cache, plus 3 more to copy from the miniport to the protocol driver). That's enough to bring even today's advanced memory busses to their knees.

1.2 Too Much Processing by the CPU

In all but the original move from the NIC to system memory, the system CPU is responsible for moving the data. This is particularly expensive because while the CPU is moving this data it can do nothing else. While moving the data the CPU is typically stalled waiting for the relatively slow memory to satisfy its read and write requests. A CPU, which can execute an instruction every 5 nanoseconds, must now wait as long as several hundred nanoseconds for the memory controller to respond before it can begin its next instruction. Even today's advanced pipelining technology doesn't help in these situations because that relies on the CPU being able to do useful work while it waits for the memory controller to respond. If the only thing the CPU has to look forward to for the next several hundred instructions is more data moves, then the CPU ultimately gets reduced to the speed of the memory controller.

Moving all this data with the CPU slows the system down even after the data has been moved. Since both the source and destination cache lines must be pulled into the CPU cache when the data is moved, more than 3k of instructions and or data resident in the CPU cache must be flushed or invalidated for every 1500 byte frame. This is of course assuming a combined instruction and data second level cache, as is the case with the Pentium processors. After the data has been moved, the former resident of the cache will likely need to be pulled back in, stalling the CPU even when we are not performing network processing. Ideally a system would never have to bring network frames into the CPU cache, instead reserving that precious commodity for instructions and data that are referenced repeatedly and frequently.

But the data movement is not the only drain on the CPU. There is also a fair amount of processing that must be done by the protocol stack software. The most obvious expense is calculating the checksum for each TCP segment (or UDP datagram). Beyond this, however, there is other processing to be done as well. The TCP connection object must be located when a given TCP segment arrives, IP header checksums must be calculated, there are buffer and memory management issues, and finally there is also the significant expense of interrupt processing which we will discuss in the following section.

1.3 Too Many Interrupts

A 64k SMB request (write or read-reply) is typically made up of 44 TCP segments when running over Ethernet (1500 byte MTU). Each of these segments may result in an interrupt to the CPU. Furthermore, since TCP must acknowledge all of this incoming data, it's possible to get another 44 transmit-complete interrupts as a result of sending out the TCP acknowledgements. While this is possible, it is not terribly likely. Delayed ACK timers allow us to acknowledge more than one segment at a time. And delays in interrupt processing may mean that we are able to process more than one incoming network frame per interrupt. Nevertheless, even if we assume 4 incoming frames per input, and an acknowledgement for every 2 segments (as is typical per the ACK-every-other-segment property of TCP), we are still left with 33 interrupts per 64k SMB request.

Interrupts tend to be very costly to the system. Often when a system is interrupted, important information must be flushed or invalidated from the system cache so that the interrupt routine instructions, and needed data can be pulled into the cache. Since the

CPU will return to its prior location after the interrupt, it is likely that the information flushed from the cache will immediately need to be pulled back into the cache.

What's more, interrupts force a pipeline flush in today's advanced processors. While the processor pipeline is an extremely efficient way of improving CPU performance, it can be expensive to get going after it has been flushed.

Finally, each of these interrupts results in expensive register accesses across the peripheral bus (PCI). This is discussed more in the following section.

1.4 Inefficient Use of the Peripheral Bus (PCI)

We noted earlier that when the CPU has to access system memory, it may be stalled for several hundred nanoseconds. When it has to read from PCI, it may be stalled for many microseconds. This happens every time the CPU takes an interrupt from a standard NIC. The first thing the CPU must do when it receives one of these interrupts is to read the NIC Interrupt Status Register (ISR) from PCI to determine the cause of the interrupt. The most troubling thing about this is that since interrupt lines are shared on PC-based systems, we may have to perform this expensive PCI read even when the interrupt is not meant for us!

There are other peripheral bus inefficiencies as well. Typical NICs operate using descriptor rings. When a frame arrives, the NIC reads a receive descriptor from system memory to determine where to place the data. Once the data has been moved to main memory, the descriptor is then written back out to system memory with status about the received frame. Transmit operates in a similar fashion. The CPU must notify that NIC that it has a new transmit. The NIC will read the descriptor to locate the data, read the data itself, and then write the descriptor back with status about the send. Typically on transmits the NIC will then read the next expected descriptor to see if any more data needs to be sent. In short, each receive or transmit frame results in 3 or 4 separate PCI reads or writes (not counting the status register read).

2 Summary of the Invention

Alacritech was formed with the idea that the network processing described above could be offloaded onto a cost-effective Intelligent Network Interface Card (INIC). With the Alacritech INIC, we address each of the above problems, resulting in the following advancements:

1. The vast majority of the data is moved directly from the INIC into its final destination. A single trip across the system memory bus.
2. There is no header processing, little data copying, and no checksumming required by the CPU. Because of this, the data is never moved into the CPU cache, allowing the system to keep important instructions and data resident in the CPU cache.
3. Interrupts are reduced to as little as 4 interrupts per 64k SMB read and 2 per 64k SMB write.
4. There are no CPU reads over PCI and there are fewer PCI operations per receive or transmit transaction.

In the remainder of this document we will describe how we accomplish the above.

2.1 Perform Transport Level Processing on the INIC

In order to keep the system CPU from having to process the packet headers or checksum the packet, we must perform this task on the INIC. This is a daunting task. There are more than 20,000 lines of C code that make up the FreeBSD TCP/IP protocol stack. Clearly this is more code than could be efficiently handled by a competitively priced network card. Furthermore, as we've noted above, the TCP/IP protocol stack is complicated enough to consume a 200 MHz Pentium-Pro. Clearly in order to perform this function on an inexpensive card, we need special network processing hardware as opposed to simply using a general purpose CPU.

2.1.1 Only Support TCP/IP

In this section we introduce the notion of a "context". A context is required to keep track of information that spans many, possibly discontiguous, pieces of information. When processing TCP/IP data, there are actually two contexts that must be maintained. The first context is required to reassemble IP fragments. It holds information about the status of the IP reassembly as well as any checksum information being calculated across the IP datagram (UDP or TCP). This context is identified by the IP_ID of the datagram as well as the source and destination IP addresses. The second context is required to handle the sliding window protocol of TCP. It holds information about which segments have been sent or received, and which segments have been acknowledged, and is identified by the IP source and destination addresses and TCP source and destination ports.

If we were to choose to handle both contexts in hardware, we would have to potentially keep track of many pieces of information. One such example is a case in which a single 64k SMB write is broken down into 44 1500 byte TCP segments, which are in turn broken down into 131 576 byte IP fragments, all of which can come in any order (though the maximum window size is likely to restrict the number of outstanding segments considerably).

Fortunately, TCP performs a Maximum Segment Size negotiation at connection establishment time, which should prevent IP fragmentation in nearly all TCP connections. The only time that we should end up with fragmented TCP connections is when there is a router in the middle of a connection which must fragment the segments to support a smaller MTU. The only networks that use a smaller MTU than Ethernet are serial line interfaces such as SLIP and PPP. At the moment, the fastest of these connections only run at 128k (ISDN) so even if we had 256 of these connections, we would still only need to support 34Mb/sec, or a little over three 10bT connections worth of data. This is not enough to justify any performance enhancements that the INIC offers. If this becomes an issue at some point, we may decide to implement the MTU discovery algorithm, which should prevent TCP fragmentation on all connections (unless an ICMP redirect changes the connection route while the connection is established).

With this in mind, it seems a worthy sacrifice to not attempt to handle fragmented TCP segments on the INIC.

UDP is another matter. Since UDP does not support the notion of a Maximum Segment Size, it is the responsibility of IP to break down a UDP datagram into MTU sized

packets. Thus, fragmented UDP datagrams are very common. The most common UDP application running today is NFSV2 over UDP. While this is also the most common version of NFS running today, the current version of Solaris being sold by Sun Microsystems runs NFSV3 over TCP by default. We can expect to see the NFSV2/UDP traffic start to decrease over the coming years.

In summary, we will only offer assistance to non-fragmented TCP connections on the INIC.

2.1.2 Don't handle TCP "exceptions"

As noted above, we won't provide support for fragmented TCP segments on the INIC. We have also opted to not handle TCP connection and breakdown. Here is a list of other TCP "exceptions" which we have elected to not handle on the INIC:

Fragmented Segments –Discussed above.

Retransmission Timeout – Occurs when we do not get an acknowledgement for previously sent data within the expected time period.

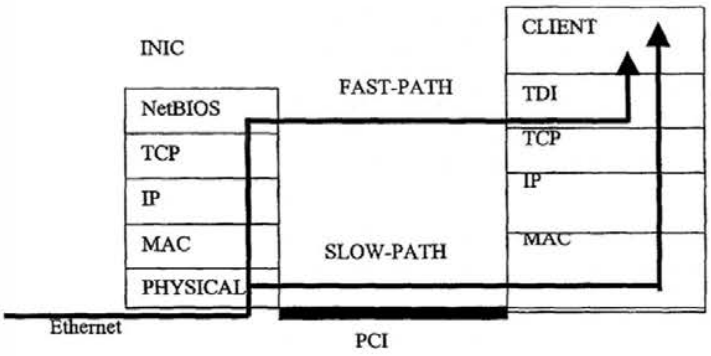
Out of order segments – Occurs when we receive a segment with a sequence number other than the next expected sequence number.

FIN segment – Signals the close of the connection.

Since we have now eliminated support for so many different code paths, it might seem hardly worth the trouble to provide any assistance by the card at all. This is not the case. According to W. Richard Stevens and Gary Write in their book "TCP/IP Illustrated Volume 2", TCP operates without experiencing any exceptions between 97 and 100 percent of the time in local area networks. As network, router, and switch reliability improve this number is likely to only improve with time.

2.1.3 Two modes of operation

So the next question is what to do about the network packets that do not fit our criteria. The answer is to use two modes of operation: One in which the network frames are processed on the INIC through TCP and one in which the card operates like a typical dumb NIC. We call these two modes fast-path, and slow-path. In the slow-path case, network frames are handed to the system at the MAC layer and passed up through the host protocol stack like any other network frame. In the fast path case, network data is given to the host after the headers have been processed and stripped.



The transmit case works in much the same fashion. In slow-path mode the packets are given to the INIC with all of the headers attached. The INIC simply sends these packets out as if it were a dumb NIC. In fast-path mode, the host gives raw data to the INIC which it must carve into MSS sized segments, add headers to the data, perform checksums on the segment, and then send it out on the wire.

2.1.4 The TCB cache

Consider a situation in which a TCP connection is being handled by the card and a fragmented TCP segment for that connection arrives. In this situation, it will be necessary for the card to turn control of this connection over to the host.

This introduces the notion of a Transmit Control Block (TCB) cache. A TCB is a structure that contains the entire context associated with a connection. This includes the source and destination IP addresses and source and destination TCP ports that define the connection. It also contains information about the connection itself such as the current send and receive sequence numbers, and the first-hop MAC address, etc. The complete set of TCBs exists in host memory, but a subset of these may be "owned" by the card at any given time. This subset is the TCB cache. The INIC can own up to 256 TCBs at any given time.

TCBs are initialized by the host during TCP connection setup. Once the connection has achieved a "steady-state" of operation, its associated TCB can then be turned over to the INIC, putting us into fast-path mode. From this point on, the INIC owns the connection until either a FIN arrives signaling that the connection is being closed, or until an

exception occurs which the INIC is not designed to handle (such as an out of order segment). When any of these conditions occur, the INIC will then flush the TCB back to host memory, and issue a message to the host telling it that it has relinquished control of the connection, thus putting the connection back into slow-path mode. From this point on, the INIC simply hands incoming segments that are destined for this TCB off to the host with all of the headers intact.

Note that when a connection is owned by the INIC, the host is not allowed to reference the corresponding TCB in host memory as it will contain invalid information about the state of the connection.

2.1.5 TCP hardware assistance

When a frame is received by the INIC, it must verify it completely before it even determines whether it belongs to one of its TCBs or not. This includes all header validation (is it IP, IPV4 or V6, is the IP header checksum correct, is the TCP checksum correct, etc). Once this is done it must compare the source and destination IP address and the source and destination TCP port with those in each of its TCBs to determine if it is associated with one of its TCBs. This is an expensive process. To expedite this, we have added several features in hardware to assist us. The header is fully parsed by hardware and its type is summarized in a single status word. The checksum is also verified automatically in hardware, and a hash key is created out of the IP addresses and TCP ports to expedite TCB lookup. For full details on these and other hardware optimizations, refer to the INIC Hardware Specification sections (Heading 8).

With the aid of these and other hardware features, much of the work associated with TCP is done essentially for free. Since the card will automatically calculate the checksum for TCP segments, we can pass this on to the host, even when the segment is for a TCB that the INIC does not own.

2.1.6 TCP Summary

By moving TCP processing down to the INIC we have offloaded the host of a large amount of work. The host no longer has to pull the data into its cache to calculate the TCP checksum. It does not have to process the packet headers, and it does not have to generate TCP ACKs. We have achieved most of the goals outlined above, but we are not done yet.

2.2 Transport Layer Interface

This section defines the INIC's relation to the hosts transport layer interface (Called TDI or Transport Driver Interface in Windows NT). For full details on this interface, refer to the Alacritech TCP (ATCP) driver specification (Heading 4).

2.2.1 Receive

Simply implementing TCP on the INIC does not allow us to achieve our goal of landing the data in its final destination. Somehow the host has to tell the INIC where to put the data. This is a problem in that the host can not do this without knowing what the data

actually *is*. Fortunately, NT has provided a mechanism by which a transport driver can "indicate" a small amount of data to a client above it while telling it that it has more data to come. The client, having then received enough of the data to know what it is, is then responsible for allocating a block of memory and passing the memory address or addresses back down to the transport driver, which is in turn responsible for moving the data into the provided location.

We will make use of this feature by providing a small amount of any received data to the host, with a notification that we have more data pending. When this small amount of data is passed up to the client, and it returns with the address in which to put the remainder of the data, our host transport driver will pass that address to the INIC which will DMA the remainder of the data into its final destination.

Clearly there are circumstances in which this does not make sense. When a small amount of data (500 bytes for example), with a push flag set indicating that the data must be delivered to the client immediately, it does not make sense to deliver some of the data directly while waiting for the list of addresses to DMA the rest. Under these circumstances, it makes more sense to deliver the 500 bytes directly to the host, and allow the host to copy it into its final destination. While various ranges are feasible, it is currently preferred that anything less than a segment's (1500 bytes) worth of data will be delivered directly to the host, while anything more will be delivered as a small piece which may be128 bytes, while waiting until receiving the destination memory address before moving the rest.

The trick then is knowing when the data should be delivered to the client or not. As we've noted, a push flag indicates that the data should be delivered to the client immediately, but this alone is not sufficient. Fortunately, in the case of NetBIOS transactions (such as SMB), we are explicitly told the length of the session message in the NetBIOS header itself. With this we can simply indicate a small amount of data to the host immediately upon receiving the first segment. The client will then allocate enough memory for the entire NetBIOS transaction, which we can then use to DMA the remainder of the data into as it arrives. In the case of a large (56k for example) NetBIOS session message, all but the first couple hundred bytes will be DMA'd to their final destination in memory.

But what about applications that do not reside above NetBIOS? In this case we can not rely on a session level protocol to tell us the length of the transaction. Under these circumstances we will buffer the data as it arrives until A) we have receive some predetermined number of bytes such as 8k, or B) some predetermined period of time passes between segments or C) we get a push flag. If after any of these conditions occur we will then indicate some or all of the data to the host depending on the amount of data buffered. If the data buffered is greater than about 1500 bytes we must then also wait for the memory address to be returned from the host so that we may then DMA the remainder of the data.

2.2.2 Transmit

The transmit case is much simpler. In this case the client (NetBIOS for example) issues a TDI Send with a list of memory addresses which contain data that it wishes to send along

with the length. The host can then pass this list of addresses and length off to the INIC. The INIC will then pull the data from its source location in host memory, as it needs it, until the complete TDI request is satisfied.

2.2.3 Affect on interrupts

Note that when we receive a large SMB transaction, for example, that there are two interactions between the INIC and the host. The first in which the INIC indicates a small amount of the transaction to the host, and the second in which the host provides the memory location(s) in which the INIC places the remainder of the data. This results in only two interrupts from the INIC. The first when it indicates the small amount of data and the second after it has finished filling in the host memory given to it. A drastic reduction from the 33/64k SMB request that we estimate at the beginning of this section.

On transmit, we actually only receive a single interrupt when the send command that has been given to the INIC completes.

2.2.4 Transport Layer Interface Summary

Having now established our interaction with Microsoft's TDI interface, we have achieved our goal of landing most of our data directly into its final destination in host memory. We have also managed to transmit all data from its original location on host memory. And finally, we have reduced our interrupts to 2 per 64k SMB read and 1 per 64k SMB write. The only thing that remains in our list of objectives is to design an efficient host (PCI) interface.

2.3 Host (PCI) Interface

In this section we define the host interface. For a more detailed description, refer to the "Host Interface Strategy for the Alacritech INIC" section (Heading 3).

2.3.1 Avoid PCI reads

One of our primary objectives in designing the host interface of the INIC was to eliminate PCI reads in either direction. PCI reads are particularly inefficient in that they completely stall the reader until the transaction completes. As we noted above, this could hold a CPU up for several microseconds, a thousand times the time typically required to execute a single instruction. PCI writes on the other hand, are usually buffered by the memory-bus⇔PCI-bridge allowing the writer to continue on with other instructions. This technique is known as "posting".

2.3.1.1 Memory-based status register

The only PCI read that is required by most NICs is the read of the interrupt status register. This register gives the host CPU information about what event has caused an interrupt (if any). In the design of our INIC we have elected to place this necessary status register into host memory. Thus, when an event occurs on the INIC, it writes the status register to an agreed upon location in host memory. The corresponding driver on the host reads this local register to determine the cause of the interrupt. The interrupt lines are

held high until the host clears the interrupt by writing to the INIC's Interrupt Clear Register. Shadow registers are maintained on the INIC to ensure that events are not lost.

2.3.1.2 Buffer Addresses are pushed to the INIC

Since it is imperative that our INIC operate as efficiently as possible, we must also avoid PCI reads from the INIC. We do this by pushing our receive buffer addresses to the INIC. As mentioned at the beginning of this section, most NICs work on a descriptor queue algorithm in which the NIC reads a descriptor from main memory in order to determine where to place the next frame. We will instead write receive buffer addresses to the INIC as receive buffers are filled. In order to avoid having to write to the INIC for every receive frame, we instead allow the host to pass off a pages worth (4k) of buffers in a single write.

2.3.2 Support small and large buffers on receive

In order to reduce further the number of writes to the INIC, and to reduce the amount of memory being used by the host, we support two different buffer sizes. A small buffer contains roughly 200 bytes of data payload, as well as extra fields containing status about the received data bringing the total size to 256 bytes. We can therefore pass 16 of these small buffers at a time to the INIC. Large buffers are 2k in size. They are used to contain any fast or slow-path data that does not fit in a small buffer. Note that when we have a large fast-path receive, a small buffer will be used to indicate a small piece of the data, while the remainder of the data will be DMA'd directly into memory. Large buffers are never passed to the host by themselves, instead they are always accompanied by a small buffer which contains status about the receive along with the large buffer address. By operating in the manner, the driver must only maintain and process the small buffer queue. Large buffers are returned to the host by virtue of being attached to small buffers. Since large buffers are 2k in size they are passed to the INIC 2 buffers at a time.

2.3.3 Command and response buffers

In addition to needing a manner by which the INIC can pass incoming data to us, we also need a manner by which we can instruct the INIC to send data. Plus, when the INIC indicates a small amount of data in a large fast-path receive, we need a method of passing back the address or addresses in which to put the remainder of the data. We accomplish both of these with the use of a command buffer. Sadly, the command buffer is the only place in which we must violate our rule of only pushing data across PCI. For the command buffer, we write the address of command buffer to the INIC. The INIC then reads the contents of the command buffer into its memory so that it can execute the desired command. Since a command may take a relatively long time to complete, it is unlikely that command buffers will complete in order. For this reason we also maintain a response buffer queue. Like the small and large receive buffers, a page worth of response buffers is passed to the INIC at a time. Response buffers are only 32 bytes, so we have to replenish the INIC's supply of them relatively infrequently. The response buffers only purpose is to indicate the completion of the designated command buffer, and to pass status about the completion.

2.4 Examples

In this section we will provide a couple of examples describing some of the differing data flows that we might see on the Alacritech INIC.

2.4.1 Fast-path 56k NetBIOS session message

Let's say a 56k NetBIOS session message is received on the INIC. The first segment will contain the NetBIOS header, which contains the total NetBIOS length. A small chunk of this first segment is provided to the host by filling in a small receive buffer, modifying the interrupt status register on the host, and raising the appropriate interrupt line. Upon receiving the interrupt, the host will read the ISR, clear it by writing back to the INIC's Interrupt Clear Register, and will then process its small receive buffer queue looking for receive buffers to be processed. Upon finding the small buffer, it will indicate the small amount of data up to the client to be processed by NetBIOS. It will also, if necessary, replenish the receive buffer pool on the INIC by passing off a pages worth of small buffers. Meanwhile, the NetBIOS client will allocate a memory pool large enough to hold the entire NetBIOS message, and will pass this address or set of addresses down to the transport driver. The transport driver will allocate an INIC command buffer, fill it in with the list of addresses, set the command type to tell the INIC that this is where to put the receive data, and then pass the command off to the INIC by writing to the command register. When the INIC receives the command buffer, it will DMA the remainder of the NetBIOS data, as it is received, into the memory address or addresses designated by the host. Once the entire NetBIOS transaction is complete, the INIC will complete the command by writing to the response buffer with the appropriate status and command buffer identifier.

In this example, we have two interrupts, and all but a couple hundred bytes are DMA'd directly to their final destination. On PCI we have two interrupt status register writes, two interrupt clear register writes, a command register write, a command read, and a response buffer write.

With a standard NIC this would result in an estimated 30 interrupts, 30 interrupt register reads, 30 interrupt clear writes, and 58 descriptor reads and writes. Plus the data will get moved anywhere from 4 to 8 times across the system memory bus.

2.4.2 Slow-path receive

If the INIC receives a frame that does not contain a TCP segment for one of its TCB's, it simply passes it to the host as if it were a dumb NIC. If the frame fits into a small buffer (~200 bytes or less), then it simply fills in the small buffer with the data and notifies the host. Otherwise it places the data in a large buffer, writes the address of the large buffer into a small buffer, and again notifies the host. The host, having received the interrupt and found the completed small buffer, checks to see if the data is contained in the small buffer, and if not, locates the large buffer. Having found the data, the host will then pass the frame upstream to be processed by the standard protocol stack. It must also replenish the INIC's small and large receive buffer pool if necessary.

With the INIC, this will result in one interrupt, one interrupt status register write and one interrupt clear register write as well as a possible small and or large receive buffer register write. The data will go through the normal path although if it is TCP data then the host will not have to perform the checksum.

With a standard NIC this will result in a single interrupt, an interrupt status register read, an interrupt clear register write, and a descriptor read and write. The data will get processed as it would by the INIC, except for a possible extra checksum.

2.4.3 Fast-path 400 byte send

In this example, lets assume that the client has a small amount of data to send. It will issue the TDI Send to the transport driver which will allocate a command buffer, fill it in with the address of the 400 byte send, and set the command to indicate that it is a transmit. It will then pass the command off to the INIC by writing to the command register. The INIC will then DMA the 400 bytes into its own memory, prepare a frame with the appropriate checksums and headers, and send the frame out on the wire. After it has received the acknowledgement it will then notify the host of the completion by writing to a response buffer.

With the INIC, this will result in one interrupt, one interrupt status register write, one interrupt clear register write, a command buffer register write a command buffer read, and a response buffer write. The data is DMA'd directly from the system memory.

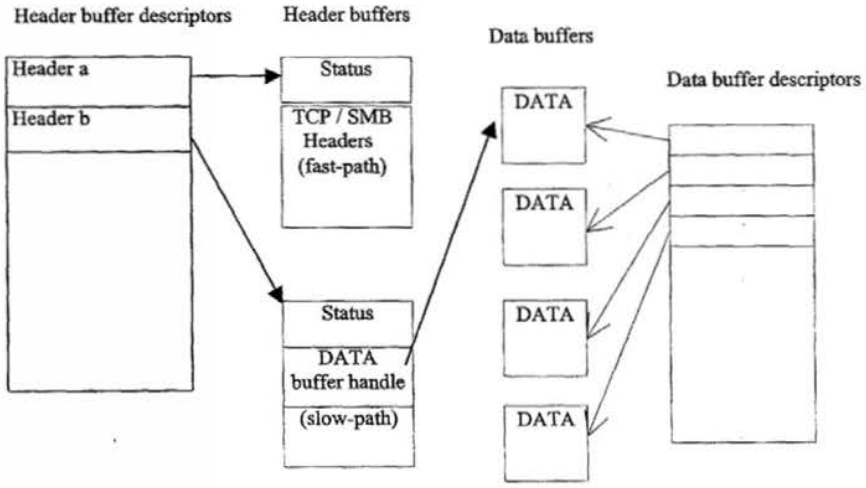
With a standard NIC this will result in a single interrupt, an interrupt status register read, an interrupt clear register write, and a descriptor read and write. The data would get moved across the system bus a minimum of 4 times. The resulting TCP ACK of the data, however, would add yet another interrupt, another interrupt status register read, interrupt clear register write, a descriptor read and write, and yet more processing by the host protocol stack.

3 Host Interface Strategy for the Alacritech INIC

This section describes the host interface strategy for the Alacritech Intelligent Network Interface Card (INIC). The goal of the Alacritech INIC is to not only process network data through TCP, but also to provide zero-copy support for the SMP upper-layer protocol. It achieves this by supporting two paths for sending and receiving data, the fast-path and the slow-path. The fast path data flow corresponds to connections that are maintained on the NIC, while slow-path traffic corresponds to network data for which the NIC does not have a connection. The fast-path flow works by passing a header to the host and subsequently holding further data for that connection on the card until the host responds via an INIC command with a set of buffers into which to place the accumulated data. In the slow-path data flow, the INIC will be operating as a "dumb" NIC, so that these packets are simply dumped into frame buffers on the host as they arrive. To do either path requires a pool of smaller buffers to be used for headers and a pool of data buffers for frames/data that are too large for the header buffer, with both pools being managed by the INIC. This section discusses how these two pools of data are managed as well as how buffers are associated with a given context.

3.1 Receive Interface

The varying requirements of the fast and slow paths and a desire to save PCI bandwidth are the driving forces behind the host interface that is described herein. As mentioned above, the fast-path flow puts a header into a header buffer that is then forwarded to the host. The host uses the header to determine what further data is following, allocates the necessary host buffers, and these are passed back to the INIC via a command to the INIC. The INIC then fills these buffers from data it was accumulating on the card and notifies the host by sending a response to the command. Alternatively, the fast-path may receive a header and data that is a complete request, but that is also too large for a header buffer. This results in a header and data buffer being passed to the host. This latter flow is identical to the slow-path flow, which also puts all the data into the header buffer or, if the header is too small, uses a large (2K) host buffer for all the data. This means that on the unsolicited receive path, the host will only see either a header buffer or a header and at most, one data buffer. Note that data is never split between a header and a data buffer. The diagram below illustrates both situations:



Since we want to fill in the header buffer with a single DMA, the header must be the last piece of data to be written to the host for any received transaction.

3.1.1 Receive Interface Details

3.1.2 Header Buffers

Header buffers in host memory are 256 bytes long, and are aligned on 256 byte boundaries. There will be a field in the header buffer indicating it has valid data. This field will initially be reset by the host before passing the buffer descriptor to the INIC. A

set of header buffers are passed from the host to the INIC by the host writing to the *Header Buffer Address Register* on the INIC. This register is defined as follows:

Bits 31-8 Physical address in host memory of the first of a set of contiguous header buffers
Bits 7-0 Number of header buffers passed.

In this way the host can, say, allocate 16 buffers in a 4K page, and pass all 16 buffers to the INIC with one register write. The INIC will maintain a queue of these header descriptors in the SmallHType queue in it's own local memory, adding to the end of the queue every time the host writes to the *Header Buffer Address Register*. Note that the single entry is added to the queue; the eventual dequeuer will use the count after extracting that entry.

The header buffers, will be used and returned to the host in the same order that they were given to the INIC. The valid field will be set by the INIC before returning the buffer to the host. In this way a PCI interrupt, with a single bit in the interrupt register, may be generated to indicate that there is a header buffer for the host to process. When servicing this interrupt, the host will look at its queue of header buffers, reading the valid field to determine how many header buffers are to be processed.

3.1.3 Receive Data Buffers

Receive data buffers in host memory are aligned to page boundaries, assumed here to be 2K bytes long and aligned on 4K page boundaries, 2 buffers per page. In order to pass receive data buffers to the INIC, the host must write to two registers on the INIC. The first register to be written is the *Data Buffer Handle Register*. The buffer handle is not significant to the INIC, but will be copied back to the host to return the buffer to the host. The second register written is the *Data Buffer Address Register*. This is the physical address of the data buffer. When both registers have been written, the INIC will add the contents of these two registers to FreeType queue of data buffer descriptors. Note that the INIC host driver sets the handle register first, then the address register. There needs to be some mechanism put in place to ensure the reading of these registers does not get out of sync with writing them. Effectively the INIC can read the address register first and save its contents, then read the handle register. It can then lock the register pair in some manner such that another write to the handle register is not permitted until the current contents have been saved. Both addresses extracted from the registers are to be written to the FreeType queue. The INIC will extract 2 entries each time when dequeuing.

Data buffers will be allocated and used by the INIC as needed. For each data buffer used by a slow-path transaction, the data buffer handle will be copied into a header buffer. Then the header buffer will be returned to the host.

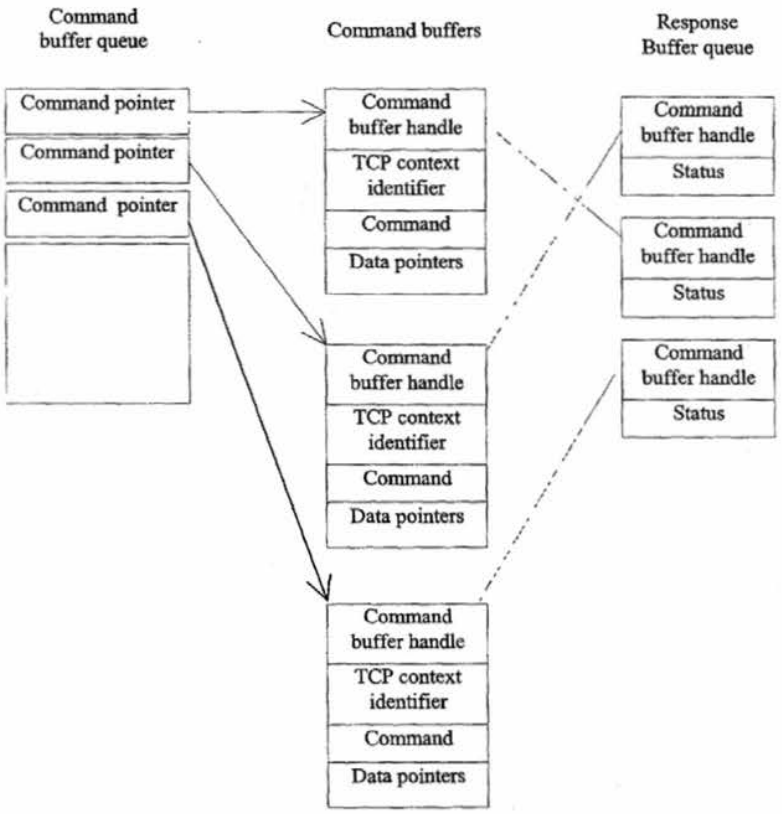
3.2 Transmit Interface

3.2.1 Transmit Interface Overview

The transmit interface, like the receive interface, has been designed to minimize the amount of PCI bandwidth and latencies. In order to transmit data, the host will transfer a command buffer to the INIC. This command buffer will include a command buffer

handle, a command field, possibly a TCP context identification, and a list of physical data pointers. The command buffer handle is defined to be the first word of the command buffer and is used by the host to identify the command. This word will be passed back to the host in a response buffer, since commands may complete out of order, and the host will need to know which command is complete. Commands will be used for many reasons, but primarily to cause the INIC to transmit data, or to pass a set of buffers to the INIC for input data on the fast-path as previously discussed.

Response buffers are physical buffers in host memory. They are used by the INIC in the same order as they were given to it by the host. This enables the host to know which response buffer(s) to next look at when the INIC signals a command completion.



3.2.2 Transmit Interface Details

3.2.2.1 Command Buffers

Command buffers in host memory are a multiple of 32 bytes, up to a maximum of 1K bytes, and are aligned on 32 byte boundaries. A command buffer is passed to the INIC by writing to one of 5 *Command Buffer Address Registers*. These registers are defined as follows:

Bits 31-5	Physical address in host memory of the command buffer.
Bits 4-0	Length of command buffer in bytes / 32 (i.e. number of multiples of 32 bytes)

This is the physical address of the command buffer. The register to which the command is written predetermines the XMT interface number, or if the command is for the RCV CPU; hence there will be 5 of them, 0 – 3 for XMT and 4 for RCV. When one of these registers has been written, the INIC will add the contents of the register to it's own internal queue of command buffer descriptors. The first word of all command buffers is defined to be the command buffer handle. It is the job of the utility CPU to extract a command from its local queue, DMA the command into a small INIC buffer (from the FreeSType queue), and queue that buffer into the Xmit#Type queue, where # is 0 – 3 depending on the interface, or the appropriate RCV queue. The receiving CPU will service the queues to perform the commands. When that CPU has completed a command, it extracts the command buffer handle and passes it back to the host via a response buffer.

3.2.2.2 Response Buffers

Response buffers in host memory are 32 bytes long and aligned on 32 byte boundaries. They are handled in a very similar fashion to header buffers. There will be a field in the response buffer indicating it has valid data. This field will initially be reset by the host before passing the buffer descriptor to the INIC. A set of response buffers are passed from the host to the INIC by the host writing to the *Response Buffer Address Register* on the INIC. This register is defined as follows:

Bits 31-8	Physical address in host memory of the first of a set of contiguous response buffers
Bits 7-0	Number of response buffers passed.

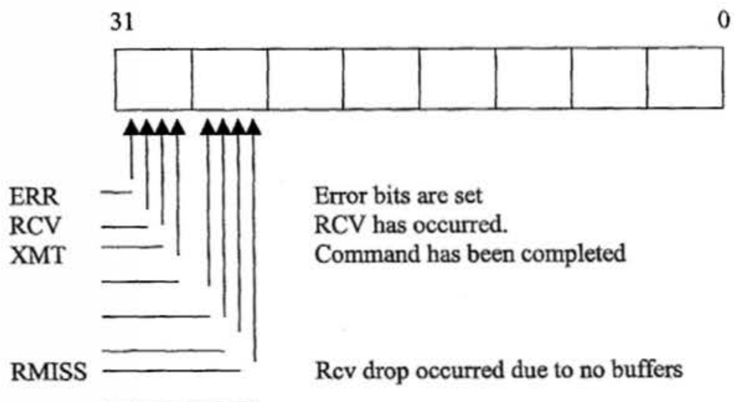
In this way the host can, say, allocate 128 buffers in a 4K page, and pass all 128 buffers to the INIC with one register write. The INIC will maintain a queue of these header descriptors in it's ResponseType queue, adding to the end of the queue every time the host writes to the *Response Buffer Address Register*. The INIC writes the extracted contents including the count, to the queue in exactly the same manner as for the header buffers.

The response buffers can be used and returned to the host in the same order that they were given to the INIC. The valid field will be set by the INIC before returning the buffer

to the host. In this way a PCI interrupt, with a single bit in the interrupt register, may be generated to indicate that there is a response buffer for the host to process. When servicing this interrupt, the host will look at its queue of response buffers, reading the valid field to determine how many response buffers are to be processed.

3.2.3 Interrupt Status Register / Interrupt Mask Register:

The following is the general format of this register:



The setting of any bits in the ISR will cause an interrupt, provided the corresponding bit in the Interrupt Mask Register is set. The default setting for the IMR is 0.

The INIC is configured so that the host should never need to directly read the ISR from the INIC. To support this, it is important for the host/INIC to arrange a buffer area in host memory into which the ISR is dumped. The address and size of that area ca be passed to the INIC via a command on the XMT interface. That command will also specify the setting for the IMR. Until the INIC receives this command, it will not DMA the ISR to host memory, and no events will cause an interrupt. The host could if necessary, read the ISR directly from the INIC in this case.

For the host to never have to actually read the register from the INIC itself, it is necessary for the INIC to update this host copy of the register whenever anything in it changes. The host will Ack (or deassert) events in the register by writing the register with 0's in appropriate bit fields. So that the host does not miss events, the following scheme has been developed:

The INIC keeps a local copy of the register whenever it DMAs it to the host i.e. after some event(s). Call this COPYA Then the INIC starts accumulating any new events not reflected in the host copy in a separate word. Call this NEWA. As the host clears bits by writing the register back with those bits set to zero, the INIC clears these bits in COPYA (or the host write-back goes directly to COPYA). If there are new events in NEWA, it

ORs them with COPYA, and DMAs this new ISR to the host. This new ISR then replaces COPYA, NEWA is cleared and the cycle then repeats.

3.2.4 Register Addresses

For the sake of simplicity, in this example the registers are at 4-byte increments from whatever the base address is. Hence:

ISR	0x0	Interrupt Status
IMR	0x4	Interrupt Mask
HBAR	0x8	Header Buffer Address
DBHR	0xC	Data Buffer Handle
DBAR	0x10	Data Buffer Address
CBAR0	0x14	Command Buffer Address XMT0
CBAR1	0x18	Command Buffer Address XMT1
CBAR2	0x1C	Command Buffer Address XMT2
CBAR3	0x20	Command Buffer Address XMT3
CBAR4	0x24	Command Buffer Address RCV
RBAR	0x28	Response Buffer Address

4 Alacritech TCP (ATCP) Design Specification

This section outlines the design specification for the Alacritech TCP (ATCP) transport driver. The ATCP driver consists of three components:

1. The bulk of the protocol stack is based on the FreeBSD TCP/IP protocol stack. This code performs the Ethernet, ARP, IP, ICMP, and (slow path) TCP processing for the driver.
2. At the top of the protocol stack we introduce an NT filter driver used to intercept TDI requests destined for the Microsoft TCP driver.
3. At the bottom of the protocol stack we include an NDIS protocol-driver interface which allows us to communicate with the INIC miniport NDIS driver beneath the ATCP driver.

This section covers each of these topics, as well as issues common to the entire ATCP driver.

4.1 Coding style

In order to ensure that our ATCP driver is written in a consistent manner, we have adopted a set of coding guidelines. These guidelines are introduced with the philosophy that we should write code in a Microsoft style since we are introducing an NT-based product. The guidelines below apply to all code that we introduce into our driver. Since a very large portion of our ATCP driver will be based on FreeBSD, and since we are somewhat time-constrained on our driver development, the ported FreeBSD code will be exempt from these guidelines.

1. Global symbols – All function names and global variables in the ATCP driver should begin with the "ATK" prefix (ATKSend() for instance).
2. Variable names – Microsoft seems to use capital letters to separate multi-word variable names instead of underscores (VariableName instead of variable_name). We should adhere to this style.
3. Structure pointers – Microsoft typedefs all of their structures. The structure types are always capitals and they typedef a pointer to the structure as "P"<name> as follows:

```
typedef  struct _FOO {
        INT     bar;
} FOO, *PFOO;
```

   We will adhere to this style.

4. Function calls – Microsoft separates function call arguments on separate lines:

```
X = foobar(
        argument1,
        argument2,
        );
```

   We will adhere to this style.

5. Comments – While Microsoft seems to alternatively use // and /* */ comment notation, we will exclusively use the /* */ notation.
6. Function comments – Microsoft includes comments with each function that describe the function, its arguments, and its return value. We will also include these comments, but will move them from within the function itself to just prior to the function for better readability.
7. Function arguments – Microsoft includes the keywords IN and OUT when defining function arguments. These keywords denote whether the function argument is used as an input parameter, or alternatively as a placeholder for an output parameter. We will include these keywords.
8. Function prototypes – We will include function prototypes in the most logical header file corresponding to the .c file. For example, the prototype for function foo() found in foo.c will be placed in foo.h.
9. Indentation – Microsoft code fairly consistently uses a tabstop of 4. We will do likewise.
10. Header file #ifndef – each header file should contain a #ifndef/#define/#endif which is used to prevent recursive header file includes. For example, foo.h would include:

```
#ifndef __FOO_H__
#define __FOO_H__
<foo.h contents..>
#endif /* __FOO_H__ */
```

    Note the __NAME_H__ format.
11. Each file must contain a comment at the beginning which includes the Id as follows:

```
/*
 * $Id$
 */
```

CVS (RCS) will expand this keyword to denote RCS revision, timestamps, author, etc.

4.2 SMP

This section describes the process by which we will make the ATCP driver SMP safe.

The basic rule for SMP kernel code is that any access to a memory variable must be protected by a lock that prevents a competing access by code running on another processor. Spinlocks are the normal locking method for code paths which do not take a long time to execute (and which do not sleep.)

In general each instance of a structure will include a spinlock, which must be acquired before members of that structure are accessed, and held while a function is accessing that instance of the structure. Structures which are logically grouped together may be protected by a single spinlock: for example, the 'in_pcb' structure, 'tcpcb' structure, and 'socket' structure which together constitute the administrative information for a TCP connection will probably be collectively managed by a single spinlock in the 'socket' structure.

In addition, every global data structure such as a list or hash table must also have a protecting spinlock which must be held while the structure is being accessed or modified. The NT DDK in fact provides a number of convenient primitives for SMP-safe list manipulation, and it is recommended that these be used for any new lists. Existing list manipulations in the FreeBSD code can probably be left as-is to minimize code disturbance, except of course that the necessary spinlock acquisition and release must be added around them.

Spinlocks should not be held for long periods of time, and most especially, must not be held during a sleep, since this will lead to deadlocks. There is a significant deficiency in the NT kernel support for SMP systems: it does not provide an operation which allows a spinlock to be exchanged atomically for a sleep lock. This would be a serious problem in a UNIX environment where much of the processing occurs in the context of the user process which initiated the operation. (The spinlock would have to be explicitly released, followed by a separate acquisition of the sleep lock: creating an unsafe window.)

The NT approach is more asynchronous, however: IRPs are simply marked as 'PENDING' when an operation cannot be completed immediately. The calling thread does NOT sleep at that point: it returns, and may go on with other processing. Pending IRPs are later completed, not by waking up the thread which initiated them, but by an 'IoCompleteRequest' call which typically runs at DISPATCH level in an arbitrary context.

Thus we have not in fact used sleep locks anywhere in the design of the ATCP driver, hoping the above issue will not arise.

4.3 Data flow overview

The ATCP driver supports two paths for sending and receiving data, the fast-path and the slow-path. The fast-path data flow corresponds to connections that are maintained on the INIC, while slow-path traffic corresponds to network data for which the INIC does not have a connection. In order to set some groundwork for the rest of this section, these two data paths are summarized here.

4.3.1 Fast-path input data flow

There are 2 different cases to consider:

1. NETBIOS traffic (identifiable by port number.)
2. Everything else.

4.3.1.1 NETBIOS input

As soon as the INIC has received a segment containing a NETBIOS header, it will forward it up to the TCP driver, along with the NETBIOS length from the header. (In principle the host could get this from the header itself, but since the INIC has already done the decode, it seem reasonable to just pass it.)

From the TDI spec, the amount of data in the buffer actually sent must be at least 128 bytes. For small SMBs, all of the received SMB should be forwarded; it will be absorbed directly by the TDI client without any further MDL exchange. Experiments tracing the TDI data flow show that the NETBIOS client directly absorbs up to 1460 bytes: the amount of payload data in a single Ethernet frame. Thus the initial system specifies that the INIC will indicate anything up to a complete segment to the ATCP driver. [See note (1)]

Once the INIC has passed up an indication with an NETBIOS length greater than the amount of data in the packet it passed, it will continue to accumulate further incoming data in DRAM on the INIC. Overflow of INIC DRAM buffers will be avoided by using a receive window on the INIC at this point, which can be 8K.

On receiving the indicated packet, the ATCP driver will call the receive handler registered by the TDI client for the connection, passing the actual size of the data in the packet from the INIC as "bytes indicated" and the NETBIOS length as "bytes available." [See note (2)].

In the "large data input" case, where "bytes available" exceeds the packet length, the TDI client will then provide an MDL, associated with an IRP, which must be completed when this MDL is filled. (This IRP/MDL may come back either in the response to TCP's call of the receive handler, or as an explicit TDI_RECEIVE request.)

The ATCP driver will build a "receive request" from the MDL information, and pass this to the INIC. This request will contain:

- The TCP context identifier.
- Size and offset information.
- A list of physical addresses corresponding to the MDL pages.

- A context field to allow the ATCP driver to identify the request on completion.
- "Piggybacked" window update information (this will be discussed in section 6.1.3.)

Note: the ATCP driver must copy any remaining data (which was not taken by the receive handler) from the segment indicated by the INIC to the start of the MDL, and must adjust the size & offset information in the request passed to the INIC to account for this.

The INIC will fill the given page(s) with incoming data up to the requested amount, and respond to the ATCP driver when this is done [see note (3)]. If the MDL is large, the INIC may open up its advertised receive window for improved throughput while filling the MDL.

On receiving the response from the INIC, the ATCP driver will complete the IRP associated with this MDL, to tell the TDI client that the data is available.

At this point the cycle of events is complete, and the ATCP driver is now waiting for the next header indication.

4.3.1.2 Other TCP input.

In the general case we do not have a higher-level protocol header to enable us to predict that more data is coming. So on non-NETBIOS connections, the INIC will just accumulate incoming data in INIC DRAM up to a quantity of 8K in this example. Again, a maximum advertised window size, which may be 16K, will be used to prevent overflow of INIC DRAM buffers.

When the prescribed amount has been accumulated, or when a PSH flag is seen, the INIC will indicate a small packet which may be 128 bytes of the data to the ATCP driver, along with the total length of the data accumulated in INIC DRAM.

On receiving the indicated packet, the ATCP driver will call the receive handler registered by the TDI client for the connection, passing the actual size of the data in the packet from the INIC as "bytes indicated" and the total INIC-buffer length as "bytes available."

As in the NETBIOS case, if "bytes available" exceeds "bytes indicated", the TDI client will provide an IRP with an MDL. The ATCP driver will pass the MDL to the INIC to be filled, as before. The INIC will reply to the ATCP driver, which in turn will complete the IRP to the TDI client.

Using an MDL from the client avoids a copy step. However, if we can only buffer 8K and delay indicating to the ATCP driver until we have done so, a question arises regarding further segments coming in, since INIC DRAM is a scarce resource. We do not want to ACK with a zero-size window advertisement: this would cause the transmitting end to go into persist state, which is bad for throughput. If the transmitting end is also our INIC, this results in having to implement the persist timer on the INIC, which we do not wish to do. Instead for large transfers (i.e. no PSH flag seen) we will not send an ACK

until the host has provided the MDL, and also, to avoid stopping the transmitting end, we will use a receive window of twice the amount we will buffer before calling the host. Since the host comes back with the MDL quite quickly (measured at < 100 microseconds), we do not expect to experience significant overruns.

4.3.1.3 INIC Receive window updates

If the INIC "owns" an MDL provided by the TDI client (sent by ATCP as a receive request), it will treat this as a "promise" by the TDI client to accept the data placed in it, and may therefore ACK incoming data as it is filling the pages.

However, for small requests, there will be no MDL returned by the TDI client: it absorbs all of the data directly in the receive callback function. We need to update the INIC's view of data which has been accepted, so that it can update its receive window. In order to be able to do this, the ATCP driver will accumulate a count of data which has been accepted by the TDI client receive callback function for a connection.

From the INIC's point of view, though, segments sent up to the ATCP driver are just "thrown over the wall"; there is no explicit reply path. We will therefore "piggyback" the update on requests sent out to the INIC. Whenever the ATCP driver has outgoing data for that connection, it will place this count in a field in the send request (and then clear the counter.) Any receive request (passing a receive MDL to the INIC) may also be used to transport window update info in the same way.

Note: we will probably also need to design a message path whereby the ATCP driver can explicitly send an update of this "bytes consumed" information (either when it exceeds a preset threshold or if there are no requests going out to the INIC for more than a given time interval), to allow for possible scenarios in which the data stream is entirely one-way.

4.3.1.4 Notes

1) The PSH flag can help to identify small SMB requests that fit into one segment.
2) Actually, the observed "bytes available" from the NT TCP driver to its client's callback in this case is always 1460. The NETBIOS-aware TDI client presumably calculates the size of the MDL it will return from the NETBIOS header. So strictly speaking we do not need the NETBIOS header length at this point: just an indication that this is a header for a "large" size. However, we *do* need an actual "bytes available" value for the non-NETBIOS case, so we may as well pass it.
3) We observe that the PSH flag is set in the segment completing each NETBIOS transfer. The INIC can use this to determine when the current transfer is complete and the MDL should be returned. It can, at least in a debug mode, sanity check the amount of received data against what is expected, though.

4.3.2 Fast-path output data flow

The fast-path output data flow is similar to the input data-flow, but simpler. In this case the TDI client will provide a MDL to the ATCP driver along with an IRP to be completed when the data is sent. The ATCP driver will then give a request (corresponding to the MDL) to the INIC. This request will contain:

- The TCP context identifier.
- Size and offset information.
- A list of physical addresses corresponding to the MDL pages.
- A context field to allow the ATCP driver to identify the request on completion.
- "Piggybacked" window update information (as discussed in section 6.1.3.)

The INIC will copy the data from the given physical location(s) as it sends the corresponding network frames onto the network. When all of the data is sent, the INIC will notify the host of the completion, and the ATCP driver will complete the IRP.

Note that there may be multiple output requests pending at any given time, since SMB allows multiple SMB requests to be simultaneously outstanding.

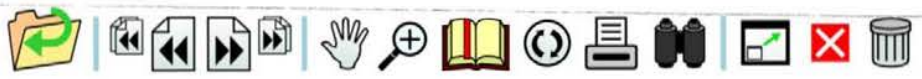
4.3.3 Slow-path data flow

For data for which there is no connection being maintained on the INIC, we will have to perform all of the TCP, IP, and Ethernet processing ourselves. To accomplish this we will port the FreeBSD protocol stack.
In this mode, the INIC will be operating as a "dumb NIC"; the packets which pass over the NDIS interface will just contain MAC-layer frames.

The MBUFs in the incoming direction will in fact be managing NDIS-allocated packets. In the outgoing direction, we need protocol-allocated MBUFs in which to assemble the data and headers. The MFREE macro must be cognizant of the various types of MBUFs, and "do the right thing" for each type. (See more extensive discussion of MBUFs in section XXX.)

We will retain a (modified) socket structure for each connection, containing the socket buffer fields expected by the FreeBSD code. The TCP code that operates on socket buffers (adding/removing MBUFs to & from queues, indicating acknowledged & received data etc) will remain essentially unchanged from the FreeBSD base (though most of the socket functions & macros used to do this will need to be modified; these are the functions in kern/uipc_socket2.c)

The upper socket layer (kern/uipc_socket.c), where the overlying OS moves data in and out of socket buffers, must be entirely re-implemented to work in TDI terms. Thus, instead of sosend(), there will be a function that copies data from the MDL provided in a TDI_SEND call into socket buffer MBUFs. Instead of soreceive(), there will be a handler that calls the TDI client receive callback function, and also copies data from socket buffer



MBUFs into any MDL provided by the TDI client (either explicitly with the callback response or as a separate TDI_RECEIVE call.)

We must note that there is a semantic difference between TDI_SEND and a write() on a BSD socket. The latter may complete back to its caller as soon as the data has been copied into the socket buffer. The completion of a TDI_SEND, however, implies that the data has actually been sent on the connection. Thus we will need to keep the TDI_SEND IRPs (and associated MDLs) in a queue on the socket until the TCP code indicates that the data from them has been ACK'd.

4.3.4 Data Path Notes

1. There might be input data on a connection object for which there is no receive handler function registered. This has not been observed, but we can probably just ASSERT for a missing handler for the moment. If it should happen, however, we must assume that the TDI client will be doing TDI_RECEIVE calls on the connection. If we can't make a callup at the time that the indication from the INIC appears, we can queue the data and handle it when a TDI_RECEIVE does appear.
2. NT has a notion of "canceling" IRPs. It is possible for us to get a "cancel" on an IRP corresponding to an MDL which has been "handed" to the INIC by a send or receive request. We can handle this by being able to force the context back off the INIC, since IRPs will only get cancelled when the connection is being aborted.

4.4 Context Passing Between ATCP and INIC

4.4.1 From ATCP to INIC

There is a synchronization problem that must be addressed here. The ATCP driver will make a decision on a given connection that this connection should now be passed to the INIC. It builds and sends a command identifying this connection to the INIC.

Before doing so, it must ensure that no slow-path outgoing data is outstanding. This is not difficult; it simply pends and queues any new TDI_SEND requests and waits for any unacknowledged slow path output data to be acknowledged before initiating the context pass operation.

The problem arises with incoming slow-path data. If we attempt to do the context-pass in a single command handshake, there is a window during which the ATCP driver has send the context command, but the INIC has not yet seen this (or has not yet completed setting up its context.) During this time, slow-path input data frames could arrive and be fed into the slow-path ATCP processing code. Should that happen, the context information which the ATCP driver passed to the INIC is no longer correct. We can simply abort the outward pass of the context in this event, but it seems better to have a reliable handshake.

Therefore, the command to pass context from ATCP driver to INIC will be split into two halves, and there will be a two-exchange handshake.

The initial command from ATCP to INIC expresses an "intention" to hand out the context. It will include the source and destination IP addresses and ports, which will allow the INIC to establish a "provisional" context. Once it has this "provisional" context in place, the INIC will not send any more slow-path input frames for that src/dest IP/port combination (it will queue them, if any are received.)

When the ATCP driver receives the response to this initial "intent" command, it knows that the INIC will send no more slow-path input. The ATCP driver then waits for any remaining unconsumed slow-path input data for this connection to be consumed by the client. (Generally speaking there will be none, since the ATCP driver will not initiate a context pass while there is unconsumed slow-path input data; the handshake is simply to close the crossover window.)

Once any such data has been consumed, we know things are in a quiescent state. The ATCP driver can then send the second, "commit" command to hand out the context, with confidence that the TCB values it is handing out (sequence numbers etc) are reliable.

Note 1: it is conceivable that there might be situations in which the ATCP driver decides, after having sent the original "intention" command, that the context is not to be passed after all. (E.g. the local client issues a close.) So we must allow for the possibility that the second command may be a "abort", which should cause the INIC to deallocate and clear up its "provisional" context.

Note 2: to simplify the logic, the ATCP driver will guarantee that only one context may be in process of being handed out at a time: in other words, it will never issue another initial "intention" command until it has completed the second half of the handshake for the first one.

4.4.2 From INIC to ATCP

There are two possible cases for this: a context transfer may be initiated either by the ATCP driver or by the INIC.

However the machinery will be very similar in the two cases. If the ATCP driver wishes to cause context to be flushed from INIC to host, it will send a "flush" message to the INIC specifying the context number to be flushed. Once the INIC receives this, it will proceed with the same steps as for the case where the flush is initiated by the INIC itself:

- The INIC will send an error response to any current outstanding receive request it is working on (corresponding to an MDL into which data is being placed.) Before sending the response, it updates the receive command "length" field to reflect the amount of data which has actually been placed in the MDL buffers at the time of the flush.
- Likewise it will send an error response for any current send request, again reporting the amount of data actually sent from the request.
- The INIC will DMA the TCB for the context back to the host. (Note: part of the information provided with a context must be the address of the TCB in the host.)

- The INIC will send a "flush" indication to the host (very preferably via the regular input path as a special type of frame) identifying the context which is being flushed. Sending this indication via the regular input path ensures that it will arrive before any following slow-path frames.

At this point, the INIC is no longer doing fast-path processing, and any further incoming frames for the connection will simply be sent to the host as raw frames for the slow input path.

The ATCP driver may not be able to complete the cleanup operations needed to resume normal slow path processing immediately on receipt of the "flush frame", since there may be outstanding send and receive requests to which it has not yet received a response.

If this is the case, the ATCP driver must set a "pend incoming TCP frames" flag in its per-connection context. The effect of this is to change the behavior of tcp_input(). This runs as a function call in the context of ip_input(), and normally returns only when incoming frames have been processed as far as possible (queued on the socket receive buffer or out-of-sequence reassembly queue.) However, if there is a flush pending and we have not yet completed resynchronization, we cannot do TCP processing and must instead queue input frames for TCP on a "holding queue" for the connection, to be picked up later when context flush is complete and normal slow path processing resumes. (This is why we want to send the "flush" indication via the normal input path: so that we can ensure it is seen before any following frames of slow-path input.)

Next we need to wait for any outstanding "send" requests to be errored off:

- The INIC maintains its context for the connection in a "zombie" state. As "send" requests for this connection come out of the INIC queue, it sends error responses for them back to the ATCP driver. (It is apparently difficult for the INIC to identify all command requests for a given context; simpler for it to just continue processing them in order, detecting ones that are for a "zombie" context as they appear.)
- The ATCP driver has a count of the number of outstanding requests it has sent to the INIC. As error responses for these are received, it decrements this count, and when it reaches zero, the ATCP driver sends a "flush complete" message to the INIC.
- When the INIC receives the "flush complete" message, it dismantles its "zombie" context. From the INIC perspective, the flush is now completed.
- When the ATCP driver has received error responses for all outstanding requests, it has all the information needed to complete its cleanup. This involves completing any IRPs corresponding to requests which have entirely completed and adjusting fields in partially-completed requests so that send and receive of slow path data will resume at the right point in the byte streams.
- Once all this cleanup is complete, the ATCP driver will loop pulling any "pended" TCP input frames off the "pending queue" mentioned above and feeding them into the normal TCP input processing. Once all input frames on this queue have been cleared off, the "pend incoming TCP frames" flag can be cleared for the connection, and we are back to normal slow-path processing.

4.5 FreeBSD Porting Specification

The largest portion of the ATCP driver is either derived, or directly taken from the FreeBSD TCP/IP protocol stack. This section defines the issues associated with porting this code, the FreeBSD code itself, and the modifications required for it to suit our needs.

4.5.1 Porting philosophy

FreeBSD TCP/IP (current version referred to as Net/3) is a general purpose TCP/IP driver. It contains code to handle a variety of interface types and many different kinds of protocols. To meet this requirement the code is often written in a sometimes confusing, over-complex manner. General-purpose structures are overlaid with other interface-specific structures so that different interface types can coexist using the same general-purpose code. For our purposes much of this complexity is unnecessary since we are only supporting a single interface type and a few specific protocols. It is therefore tempting to modify the code and data structures in an effort to make it more readable, and perhaps a bit more efficient. There are, however, some problems with doing this. First, the more we modify the original FreeBSD, the more changes we will have to make. This is especially true with regard to data structures. If we collapse two data structures into one we might improve the cleanliness of the code a bit, but we will then have to modify every reference to that data structure in the entire protocol stack. Another problem with attempting to "clean up" the code is that we might later discover that we need something that we had previously thrown away. Finally, while we might gain a small performance advantage in cleaning up the FreeBSD code, the FreeBSD TCP code will mostly only run in the slow-path connections, which are not our primary focus. Our priority is to get the slow-path code functional and reliable as quickly as possible.

For the reasons above we have adopted the philosophy that we should initially keep the data structures and code at close to the original FreeBSD implementation as possible. The code will be modified for the following reasons:

5. As required for NT interaction – Obviously we can't expect to simply "drop-in" the FreeBSD code as is. The interface of this code to the NT system will require some significant code modifications. This will mostly occur at the topmost and bottommost portions of the protocol stack, as well as the "ioctl" sections of the code. Modifications for SMP issues are also needed.

6. Unnecessary code can be removed – While we will keep the code as close to the original FreeBSD as possible, we will nonetheless remove code that will never be used (UDP is a good example of this).

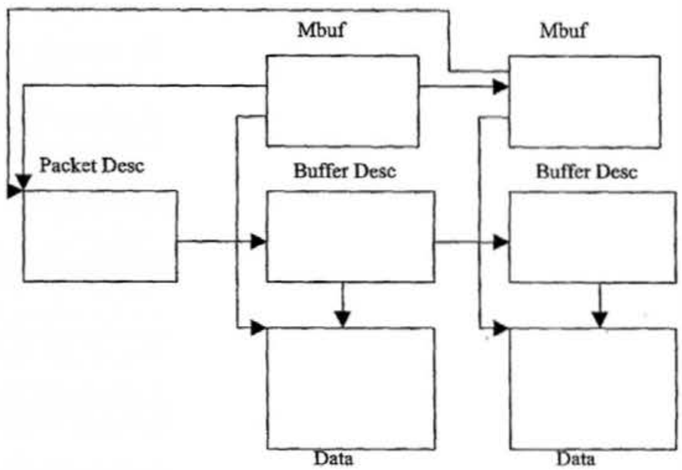
4.5.2 Unix ⇔ NT conversion

The FreeBSD TCP/IP protocol stack makes use of many Unix system services. These include bcopy to copy memory, malloc to allocate memory, timestamp functions, etc. These will not be itemized in detail since the conversion to the corresponding NT calls is a fairly trivial and mechanical operation.

An area which will need non-trivial support redesign is MBUFs.

4.5.2.1 Network buffers

Under FreeBSD, network buffers are mapped using mbufs. Under NT network buffers are mapped using a combination of packet descriptors and buffer descriptors (the buffer descriptors are really MDLs). There are a couple of problems with the Microsoft method. First it does not provide the necessary fields which allow us to easily strip off protocol headers. Second, converting all of the FreeBSD protocol code to speak in terms of buffer descriptors is an unnecessary amount of overhead. Instead, in our port we will allocate our own mbuf structures and remap the NT packets as follows:



The mbuf structure will provide the standard fields provided in the FreeBSD mbuf including the data pointer, which points to the current location of the data, data length fields and flags. In addition each mbuf will point to the packet descriptor which is associated with the data being mapped. Once an NT packet is mapped, our transport driver should never have to refer to the packet or buffer descriptors for any information except when we are finished and are preparing to return the packet.

There are a couple of things to note here. We have designed our INIC such that a packet header should never be split across multiple buffers. Thus, we should never require the equivalent of the "m_pullup" routine included in Unix. Also note that there are circumstances in which we will be accepting data that will also be accepted by the Microsoft TCP/IP. One such example of this is ARP frames. We will need to build our own ARP cache by looking at ARP replies as they come off the network. Under these circumstances, it is absolutely imperative that we do not modify the data, or the packet and buffer descriptors. We will discuss this further in the following sections.

We will allocate a pool of mbuf headers at ATCP initialization time. It is important to remember that unlike other NICs, we can not simply drop data if we run out of the system resources required to manage/map the data. The reason for this is that we will be receiving data from the card that has already been acknowledged by TCP. Because of this it is essential that we never run out of mbuf headers. To solve this problem we will statically allocate mbuf headers for the maximum number of buffers that we will ever allow to be outstanding. By doing so, the card will run out of buffers in which to put the data before we will run out of mbufs, and as a result, the card will be forced to drop data at the link layer instead of us dropping it at the transport layer.
DhXXX: as we've discussed, I don't think this is really true anymore. The INIC won't ACK data until either it's gotten a window update from ATCP to tell it the data's been accepted, or it's got an MDL.
Thus it seems workable, though undesirable, if we can't accept a frame from the INIC & return an error to it saying it was not taken.

We will also require a pool of actual mbufs (not just headers). These mbufs are required in order to build transmit protocol headers for the slow-path data path, as well as other miscellaneous purposes such as for building ARP requests. We will allocate a pool of these at initialization time and we will add to this pool dynamically as needed. Unlike the mbuf headers described above, which will be used to map acknowledged TCP data coming from the card, the full mbufs will contain data that can be dropped if we can not get an mbuf.

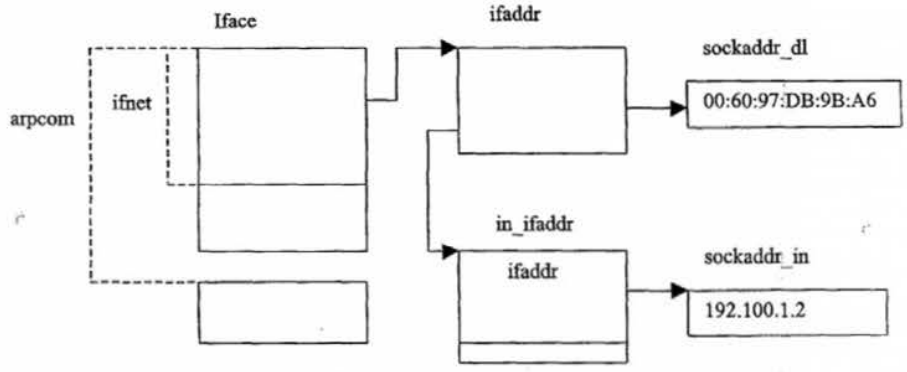
4.5.3 The code

In this section we describe each section of the FreeBSD TCP/IP port. These sections include Interface Initialization, ARP, Route, IP, ICMP, and TCP.

4.5.3.1 Interface initialization

4.5.3.1.1 Structures

There are a variety of structures, which represent a single interface in FreeBSD. These structures include:
ifnet, arpcom, ifaddr, in_ifaddr, sockaddr, sockaddr_in, and sockaddr_dl. The following illustration shows the relationship between all of these structures:



In this example we show a single interface with a MAC address of 00:60:97:DB:9B:A6 configured with an IP address of 192.100.1.2. As illustrated above, the in_ifaddr is actually an ifaddr structure with some extra fields tacked on to the end. Thus the ifaddr structure is used to represent both a MAC address and an IP address. Similarly the sockaddr structure is recast as a sockaddr_dl or a sockaddr_in depending on its address type. An interface can be configured to multiple IP addresses by simply chaining in_ifaddr structures after the in_ifaddr structure shown above.

As mentioned in the Porting Philosophy section, many of the above structures could likely be collapsed into fewer structures. In order to avoid making unnecessary modifications to FreeBSD, for the time being we will leave these structures mostly as is. We will however eliminate the fields from the structure that will never be used. These structure modifications are discussed below.

We also show above a structure called iface. This is a structure that we define. It contains the arpcom structure, which in turn contains the ifnet structure. It also contains fields that enable us to blend our FreeBSD implementation with NT NDIS requirements. One such example is the NDIS binding handle used to call down to NDIS with requests (such as send).

4.5.3.1.2 The functions

FreeBSD initializes the above structures in two phases. First when a network interface is found, the ifnet, arpcom, and first ifaddr structures are initialized first by the network layer driver, and then via a call to the if_attach routine. The subsequent in_ifaddr structure(s) are initialized when a user dynamically configures the interface. This occurs in the in_ioctl and the in_ifinit routines. Since NT allows dynamic configuration of a network interface we will continue to perform the interface initialization in two phases, but we will consolidate these two phases as described below:

4.5.3.1.2.1 IfInit

The IfInit routine will be called from the ATKProtocolBindAdapter function. The IfInit function will initialize the Iface structure and associated arpcom and ifnet structures. It will then allocate and initialize an ifaddr structure in which to contain link-level information about the interface, and a sockaddr_dl structure to contain the interface name and MAC address. Finally it will add a pointer to the ifaddr structure into the ifnet_addrs array (using the if_index field of the ifnet structure) contained in the extended device object. IfInit will then call IfConfig for each IP address that it finds in the registry entry for the interface.

4.5.3.1.2.2 IfConfig

IfConfig is called to configure an IP address for a given interface. It is passed a pointer to the ifnet structure for that interface along with all the information required to configure an IP address for that interface (such as IP address, netmask and broadcast info, etc). IfConfig will allocate an in_ifaddr structure to be used to configure the interface. It will chain it to the total chain of in_ifaddr structures contained in the extended device object, and will then configure the structure with the information given to it. After that it will add a static route for the newly configured network and then broadcast a gratuitous ARP request to notify others of our Mac/IP address and to detect duplicate IP addresses on the net.

4.5.3.2 ARP

We will port the FreeBSD ARP code to NT mostly as-is. For some reason, the FreeBSD ARP code is located in a file called if_ether.c. While the functionality of this file will remain the same, we will rename it to a more logical arp.c. The main structures used by ARP are the llinfo_arp structure and the rtentry structure (actually part of route). These structures will not be require major modifications. The functions that will require modification are defined here.

4.5.3.2.1 In_arpinput

This function is called to process an incoming ARP frame. An ARP frame can either be an ARP request or an ARP reply. ARP requests are broadcast, so we will see every ARP request on the network, while ARP replies are directed so we should only see ARP replies that are sent to us. This introduces the following possible cases for an incoming ARP frame:

1. ARP request trying to resolve our IP address – Under normal circumstances, ARP would reply to this ARP request with an ARP reply containing our MAC address. Since ARP requests will also be passed up to the Microsoft TCP/IP driver, we need not reply. Note however, that FreeBSD also creates or updates an ARP cache entry with the information derived from the ARP request. It does this in anticipation of the fact that any host that wishes to know our MAC address is likely to wish to talk to us soon. Since we will need to know his MAC address in order to talk back, we might as well add the ARP information now rather than issuing our own ARP request later.
2. ARP request trying to resolve someone else's IP address – Since ARP requests are broadcast, we see every one on the network. When we receive an ARP request of this type, we simply check to see if we have an entry for the host that sent the request in our ARP cache. If we do, we check to see if we still have the correct MAC address associated with that host. If it is incorrect, we update our ARP cache entry. Note that we do not create a new ARP cache entry in this case.
3. ARP reply – In this case we add the new ARP entry to our ARP cache. Having resolved the address, we check to see if there is any transmit requests pending for the resolve IP address, and if so, transmit them.

Given the above three possibilities, the only major change to the in_arpinput code is that we will remove the code which generates an ARP reply for ARP requests that are meant for our interface.

4.5.3.2.2 Arpintr

This is the FreeBSD code that delivers an incoming ARP frame to in_arpinput. We will be calling in_arpinput directly from our ProtocolReceiveDPC routine (discussed in the NDIS section below) so this function is not needed.

4.5.3.2.3 Arpwhohas

This is a single line function that serves only as a wrapper around arprequest. We will remove it and replace all calls to it with direct calls to arprequest.

4.5.3.2.4 Arprequest

This code simply allocates a mbuf, fills it in with an ARP header, and then passes it down to the ethernet output routine to be transmitted. For us, the code remains essentially the same except for the obvious changes related to how we allocate a network buffer, and how we send the filled in request.

4.5.3.2.5 Arp_ifinit

This is simply called when an interface is initialized to broadcast a gratuitous ARP request (described in the interface initialization section) and to set some ARP related fields in the ifaddr structure for the interface. We will simply move this functionality into the interface initialization code and remove this function.

4.5.3.2.6 Arptimer

This is a timer-based function that is called every 5 minutes to walk through the ARP table looking for entries that have timed out. Although the time-out period for FreeBSD is 20 minutes, RFC 826 does not specify any timer requirements with regard to ARP so we can modify this value or delete the timer altogether to suit our needs. Either way the function won't require any major changes.

All other functions in if_ether.c will not require any major changes.

4.5.3.3 Route

On first thought, it might seem that we have no need for routing support since our ATCP driver will only receive IP datagrams who's destination IP address matches that of one of our own interfaces. Therefore, we will not "route" from one interface to another. Instead, the MICROSOFT TCP/IP driver will provide that service. We will, however, need to maintain an up-to-date routing table so that we know a) whether an outgoing connection belongs to one of our interfaces, b) to which interface it belongs, and c) what the first-hop IP address (gateway) is if the destination is not on the local network.

We discuss four aspects on the subject of routing in this section. They are as follows:

1. The mechanics of how routing information is stored
2. The manner in which routes are added or deleted from the route table.
3. When and how route information is retrieved from the route table.
4. Notification of route table changes to interested parties.

4.5.3.3.1 The route table

In FreeBSD, the route table is maintained using an algorithm known as PATRICIA (Practical Algorithm To Retrieve Information Coded in Alphanumeric). This is a complicated algorithm that is a bit costly to set up, but is very efficient to reference. Since the routing table should contain the same information for both NT and FreeBSD, and since the key used to search for an entry in the routing table will be the same for each (the destination IP address), we should be able to port the routing table software to NT without any major changes.

The software which implements the route table (via the PATRICIA algorithm) is located in the FreeBSD file, radix.c. This file will be ported directly to the ATCP driver with no significant changes required.

4.5.3.3.2 Adding and deleting routes

Routes can be added or deleted in a number of different ways. The kernel adds or deletes routes when the state of an interface changes or when an ICMP redirect is received. User space programs such as the RIP daemon, or the route command also modify the route table.

For kernel-based route changes, the changes can be made by a direct call to the routing software. The FreeBSD software that is responsible for the modification of route table entries is found in route.c. The primary routine for all route table changes is called rtrequest(). It takes as its arguments, the request type (ADD, RESOLVE, DELETE), the destination IP address for the route, the gateway for the route, the netmask for the route, the flags for the route, and a pointer to the route structure (struct rtentry) in which we will place the added or resolved route. Other routines in the route.c file include rtinit(), which is called during interface initialization time to add a static route to the network, rtredirect, which is called by ICMP when we receive a ICMP redirect, and an assortment of support routines used for the modification of route table entries. All of these routines found in route.c will be ported with no major modifications.

For user-space-based changes, we will have to be a bit more clever. In FreeBSD, route changes are sent down to the kernel from user-space applications via a special route socket. This code is found in the FreeBSD file, rtsock.c. Obviously this will not work for our ATCP driver. Instead the filter driver portion of our driver will intercept route changes destined for the Microsoft TCP driver and will apply those modifications to our own route table via the rtrequest routine described above. In order to do this, it will have to do some format translation to put the data into the format (sockaddr_in) expected by the rtrequest routine. Obviously, none of the code from rtsock.c will be ported to the ATCP driver. This same procedure will be used to intercept and process explicit ARP cache modifications.

4.5.3.3.3 Consulting the route table

In FreeBSD, the route table is consulted in ip_output when an IP datagram is being sent. In order to avoid a complete route table search for every outgoing datagram, the route is stored into the in_pcb for the connection. For subsequent calls to ip_output, the route entry is then simply checked to ensure validity. While we will keep this basic operation as is, we will require a slight modification to allow us to coexist with the Microsoft TCP

driver. When an active connection is being set up, our filter driver will have to determine whether the connection is going to be handled by one of the INIC interfaces. To do this, we will have to consult the route table from the filter driver portion of our driver. This is done via a call to the rtalloc1 function (found in route.c). If a valid route table entry is found, then we will take control of the connection and set a pointer to the rtentry structure returned by rtalloc1 in our in_pcb structure.

4.5.3.3.4 What to do when a route changes.

When a route table entry changes, there may be connections that have pointers to a stale route table entry. These connections will need to be notified of the new route. FreeBSD solves this by checking the validity of a route entry during every call to ip_output. If the entry is no longer valid, its reference to the stale route table entry is removed, and an attempt is made to allocate a new route to the destination. For our slow path, this will work fine. Unfortunately, since our IP processing is handled by the INIC for our fast path, this sanity check method will not be sufficient. Instead, we will need to perform a review of all of our fast path connections during every route table modification. If the route table change affects our connection, we will need to advise the INIC with a new first-hop address, or if the destination is no longer reachable, close the connection entirely.

4.5.3.4 ICMP

Like the ARP code above, we will need to process certain types of incoming ICMP frames. Of the 10 possible ICMP message types, there are only three that we need to support. These include ICMP_REDIRECT, ICMP_UNREACH, and ICMP_SOURCEQUENCH. Any FreeBSD code to deal with other types of ICMP traffic will be removed. Instead, we will simply return NDIS_STATUS_NOT_ACCEPTED for all but the above ICMP frame types. This section describes how we will handle these ICMP frames.

4.5.3.4.1 ICMP_REDIRECT

Under FreeBSD, an ICMP_REDIRECT causes two things to occur. First, it causes the route table to be updated with the route given in the redirect. Second, it results in a call back to TCP to cause TCP to flush the route entry attached to its associated in_pcb structures. By doing this, it forces ip_output to search for a new route. As mentioned in the Route section above, we will also require a call to a routine which will review all of the TCP fast-path connections, and update the route entries as needed (in this case because the route entry has been zeroed). The INIC will then be notified of the route changes.

4.5.3.4.2 ICMP_UNREACH

In both FreeBSD and Microsoft TCP, the ICMP_UNREACH results in no more than a simple statistic update. We will do the same.

4.5.3.4.3 ICMP_SOURCEQUENCH

A source quench is sent to cause a TCP sender to close its congestion window to a single segment, thereby putting the sender into slow-start mode. We will keep the FreeBSD code as-is for slow-path connections. For fast path connections we will send a notification to the card that the congestion window for the given connection has been reduced. The INIC will then be responsible for the slow-start algorithm.

4.5.3.5 IP

The FreeBSD IP code should require few modifications when porting to the ATCP driver. What few modifications will be required will be discussed in this section.

4.5.3.5.1 IP initialization

During initialization time, ip_init is called to initialize the array of protosw structures. These structures contain all the information needed by IP to be able to pass incoming data to the correct protocol above it. For example, when a UDP datagram arrives, IP locates the protosw entry corresponding to the UDP protocol type value (0x11) and calls the input routine specified in that protosw entry. We will keep the array of protosw structures intact, but since we are only handling the TCP and ICMP protocols above IP, we will strip the protosw array down substantially.

4.5.3.5.2 IP input

Following are the changes required for IP input (function ip_intr()).

4.5.3.5.2.1 No IP forwarding

Since we will only be handling datagrams for which we are the final destination, we should never be required to forward an IP datagram. All references to IP forwarding, and the ip_forward function itself, can be removed.

4.5.3.5.2.2 IP options

The only options supported by FreeBSD at this time include record route, strict and loose source and record route, and timestamp. For the timestamp option, FreeBSD only logs the current time into the IP header so that before it is forwarded. Since we will not be forwarding IP datagrams, this seems to be of little use to us. While FreeBSD supports the remaining options, NT essentially does nothing useful with them. For the moment, we will not bother dealing with IP options. They will be added in later if needed.

4.5.3.5.2.3 IP reassembly

There is a small problem with the FreeBSD IP reassembly code. The reassembly code reuses the IP header portion of the IP datagram to contain IP reassembly queue information. It can do this because it no longer requires the original IP header. This is an absolute no-no with the NDIS 4.0 method of handling network packets. The NT DDK explicitly states that we must not modify packets given to us by NDIS. This is not the only place in which the FreeBSD code modifies the contents of a network buffer. It also does this when performing endian conversions. At the moment we will leave this code as is and violate the DDK rules. We believe we can do this because we are going to ensure that no other transport driver looks at these frames. If this becomes a problem we will have to modify this code substantially by moving the IP reassembly fields into the mbuf header.

4.5.3.5.3 IP output

There are only two modifications required for IP output. The first is that since, for the moment, we are not dealing with IP options, there is no need for the code that inserts the IP options into the IP header. Second, we may discover that it is impossible for us to ever receive an output request that requires fragmentation. Since TCP performs Maximum Segment Size negotiation, we should theoretically never attempt to send a TCP segment larger than the MTU.

4.6 NDIS Protocol Driver

This section defines protocol driver portion of the ATCP driver. The protocol driver portion of the ATCP driver is defined by the set of routines registered with NDIS via a call to NdisRegisterProtocol. These routines are limited to those that are called (indirectly) by the INIC miniport driver beneath us. For example, we register a ProtocolReceivePacket routine so that when the INIC driver calls NdisMIndicateReceivePacket it will result in a call from NDIS to our driver. Strictly speaking, the protocol driver portion of our driver does not include the method by which our driver calls down to the miniport (for example, the method by which we send network packets). Nevertheless, we will describe that method here for lack of a better place to put it. That said, we cover the following topics in this section of the document:

1. Initialization
2. Receive
3. Transmit
4. Query/Set Information
5. Status indications
6. Reset
7. Halt

4.6.1 Initialization

The protocol driver initialization occurs in two phases. The first phase occurs when the ATCP DriverEntry routine calls ATKProtoSetup. The ATKProtoSetup routine performs the following:

1. Allocate resources – We attempt to allocate many of the required resources as soon as possible so that we are more likely to get the memory we want. This mostly applies to allocating and initializing our mbuf and mbuf header pools.
2. Register Protocol – We call NdisRegisterProtocol to register our set of protocol driver routines.
3. Locate and initialize bound NICs – We read the Linkage parameters of the registry to determine which NIC devices we are bound to. For each of these devices we allocate and initialize a IFACE structure (defined above). We then read the TCP parameters out of the registry for each bound device and set the corresponding fields in the IFACE structure.

After the underlying INIC devices have completed their initialization, NDIS will call our driver's ATKBindAdapter function for each underlying device. It will perform the following:

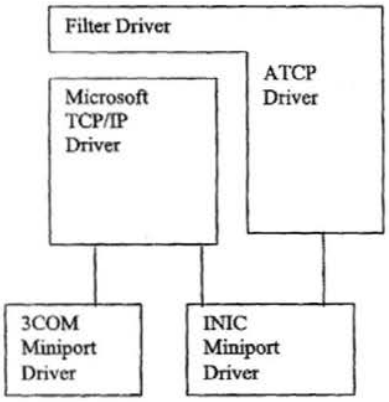
1. Open the device specified in the call the ATKBindAdapter
2. Find the IFACE structure that was created in ATKProtoSetup for this device.
3. Query the miniport for adapter information. This includes such things as link speed and MAC address. Save relevant information in the IFACE structure.
4. Perform the interface initialization as specified in section 4.5.3.1 Interface initialization

4.6.2 Receive

Receive is handled by the protocol driver routine ATKReceivePacket. Before we describe this routine, it is important to consider each possible receive type and how it will be handled.

4.6.2.1 Receive overview

Our INIC miniport driver will be bound to our transport driver as well as the generic Microsoft TCP driver (and possibly others). The ATCP driver will be bound exclusively to INIC devices, while the Microsoft TCP driver will be bound to INIC devices as well as other types of NICs. This is illustrated below:



By binding the driver in this fashion, we can choose to direct incoming network data to our own ATCP transport driver, the Microsoft TCP driver, or both. We do this by playing with the ethernet "type" field as follows.

To NDIS and the transport drivers above it, our card is going to be registered as a normal ethernet card. When a transport driver receives a packet from our driver, it will expect the data to start with an ethernet header, and consequently, expects the protocol type field to be in byte offset 12. If Microsoft TCP finds that the protocol type field is not equal to either IP, or ARP, it will not accept the packet. So, to deliver an incoming packet to our driver, we must simply map the data such that byte 12 contains a non-recognized ethernet type field. Note that we must choose a value that is greater than 1500 bytes so that the transport drivers do not confuse it with an 802.3 frame. We must also choose a value that will not be accepted by other transport driver such as Appletalk or IPX. Similarly, if we want to direct the data to Microsoft TCP, we can then simply leave the ethernet type field set to IP (or ARP). Note that since we will also see these frames we can choose to accept or not-accept them as necessary.

Incoming packets are delivered as follows:

Packets delivered to ATCP only (not accepted by MSTCP):

1. All TCP packets destined for one of our IP addresses. This includes both slow-path frames and fast-path frames. In the slow-path case, the TCP frames are given in there entirety (headers included). In the fast-path case, the ATKReceivePacket is given a header buffer that contains status information and data with no headers (except those above TCP). More on this later.

Packets delivered to Microsoft TCP only (not accepted by ATCP):

1. All non-TCP packets.

2. All packets that are not destined for one of our interfaces (packets that will be routed). Continuing the above example, if there is an IP address 144.48.252.4 associated with the 3com interface, and we receive a TCP connect with a destination IP address of 144.48.252.4, we will actually want to send that request up to the ATCP driver so that we create a fast-path connection for it. This means that we will need to know every IP address in the system and filter frames based on the destination IP address in a given TCP datagram. This can be done in the INIC miniport driver. Since it will be the ATCP driver that learns of dynamic IP address changes in the system, we will need a method to notify the INIC miniport of all the IP addresses in the system. More on this later.

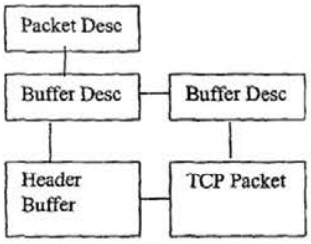
Packets delivered to both:

1. All ARP frames

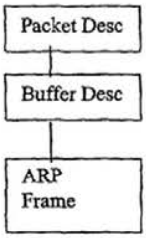
2. All ICMP frames

4.6.2.2 Two types of receive packets

There are several circumstances in which the INIC will need to indicate extra information about a receive packet to the ATCP driver. One such example is a fast path receive in which the ATCP driver will need to be notified of how much data the card has buffered. To accomplish this, the first (and sometimes only) buffer in a received packet will actually be an INIC header buffer. The header buffer contains status information about the receive packet, and may or may not contain network data as well. The ATCP driver will recognize a header buffer by mapping it to an ethernet frame and inspecting the type field found in byte 12. We will indicate all TCP frames destined for us in this fashion, while frames that are destined for both our driver and the Microsoft TCP driver (ARP, ICMP) will be indicated without a header buffer.



Example of incoming TCP pkt



Example of incoming ARP Frame

4.6.2.3 NDIS 4 ProtocolReceivePacket operation

NDIS has been designed such that all packets indicated via NdisMIndicateReceivePacket by an underlying miniport are delivered to the ProtocolReceivePacket routine for all protocol drivers bound to it. These protocol drivers can choose to accept or not accept the data. They can either accept the data by copying the data out of the packet indicated to it, or alternatively they can keep the packet and return it later via a call to NdisReturnPackets. By implementing it in this fashion, NDIS allows more than one protocol driver to accept a given packet. For this reason, when a packet is delivered to a protocol driver, the contents of the packet descriptor, buffer descriptors and data must all be treated as read-only. At the moment, we intend to violate this rule. We choose to violate this because much of the FreeBSD code modifies the packet headers as it examines them (mostly for endian conversion purposes). Rather than modify all of the FreeBSD code, we will instead ensure that no other transport driver accepts the data by making sure that the ethernet type field is unique to us (no one else will want it). Obviously this only works with data that is only delivered to our ATCP driver. For ARP and ICMP frames we will instead copy the data out of the packet into our own buffer and return the packet to NDIS directly. While this is less efficient than keeping the data and returning it later, ARP and ICMP traffic should be small enough, and infrequent enough, that it doesn't matter.

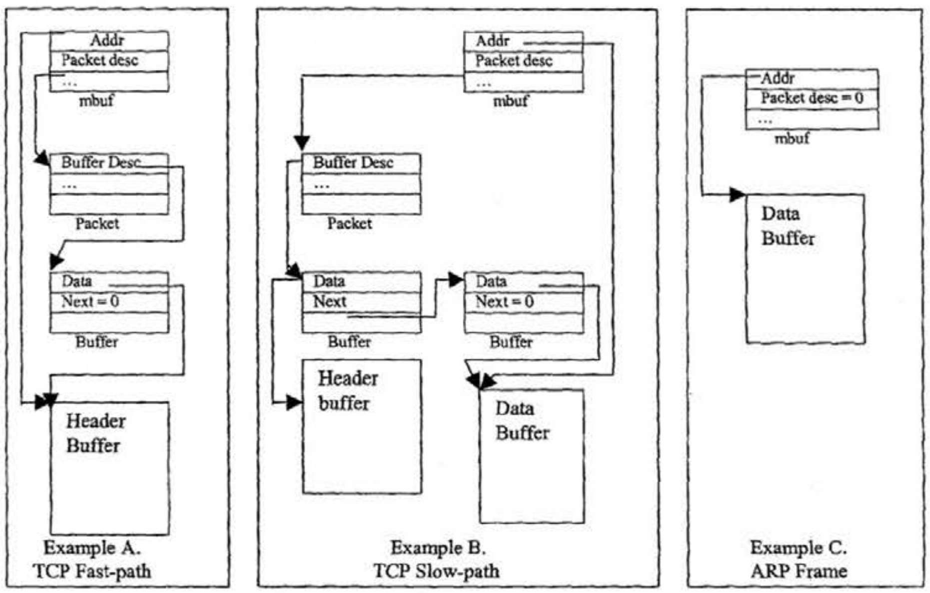
The DDK specifies that when a protocol driver chooses to keep a packet, it should return a value of 1 (or more) to NDIS in its ProtocolReceivePacket routine. The packet is then later returned to NDIS via the call to NdisReturnPackets. This can only happen after the ProtocolReceivePacket has returned control to NDIS. This requires that the call to NdisReturnPackets must occur in a different execution context. We can accomplish this by scheduling a DPC, scheduling a system thread, or scheduling a kernel thread of our own. For brevity in this section, we will assume it is a done through a DPC. In any case, we will require a queue of pending receive buffers on which to place and fetch receive packets.

After a receive packet is dequeued by the DPC it is then either passed to TCP directly for fast-path processing, or it is sent through the FreeBSD path for slow-path processing. Note that in the case of slow-path processing, we may be working on data that needs to be returned to NDIS (TCP data) or we may be working on our own copy of the data

(ARP and ICMP). When we finish with the data we will need to figure out whether or not to return the data to NDIS or not. This will be done via fields in the mbuf header used to map the data. When the mfreem routine is called to free a chain of mbufs, the fields in the mbuf will be checked and, if required, the packet descriptor pointed to by the mbuf will be returned to NDIS.

4.6.2.4 Mbuf ⇔ Packet mapping

As noted in the section on mbufs above, we will map incoming data to mbufs so that our FreeBSD port requires fewer modifications. Depending on the type of data received, this mapping will appear differently. Here are some examples:



In Example A, we show incoming data for a TCP fast-path connection. In this example, the TCP data is fully contained in the header buffer. The header buffer is mapped by the mbuf and sent upstream for fast-path TCP processing. In this case it is required that the header buffer be mapped and sent upstream because the fast-path TCP code will need information contained in the header buffer in order to perform the processing. When the mbuf in this example is freed, the mfreem routine will determine that the mbuf maps a packet that is owned by NDIS and will then free the mbuf header only and call NdisReturnPackets to free the data.

In Example B, we show incoming data for a TCP slow-path connection. In this example the mbuf points to the start of the TCP data directly instead of the header buffer. Since this buffer will be sent up for slow-path FreeBSD processing, we can not have the mbuf pointing to a header buffer (FreeBSD would get awfully confused). Again, when mfreem

is called to free the mbuf, it will discover the mapped packet, free the mbuf header, and call NDIS to free the packet and return the underlying buffers. Note that even though we do not directly map the header buffer with the mbuf we do not lose it because of the link from the packet descriptor. Note also that we could alternatively have the INIC miniport driver only pass us the TCP data buffer when it receives a slow-path receive. This would work fine except that we have determined that even in the case of slow-path connections we are going to attempt to offer some assistance to the host TCP driver (most likely by checksum processing only). In this case there may be some special fields that we need to pass up to the ATCP driver from the INIC driver. Leaving the header buffer connected seems the most logical way to do this.

Finally, in Example C, we show a received ARP frame. Recall that for incoming ARP and ICMP frames we are going to copy the incoming data out of the packet and return it directly to NDIS. In this case the mbuf simply points to our data, with no corresponding packet descriptor. When we free this mbuf, mfreem will discover this and free not only the mbuf header, but the data as well.

4.6.2.5 Other receive packets

We use this receive mechanism for other purposes besides the reception of network data. It is also used as a method of communication between the ATCP driver and the INIC. One such example is a TCP context flush from the INIC. When the INIC determines, for whatever reason, that it can no longer manage a TCP connection, it must flush that connection to the ATCP driver. It will do this by filling in a header buffer with appropriate status and delivering it to the INIC driver. The INIC driver will in turn deliver it to the protocol driver which will treat it essentially like a fast-path TCP connection by mapping the header buffer with an mbuf header and delivering it to TCP for fast-path processing. There are two advantages to communicating in this manner. First, it is already an established path, so no extra coding or testing is required. Second, since a context flush comes in, in the same manner as received frames, it will prevent us from getting a slow-path frame before the context has been flushed.

4.6.2.6 Summary

Having covered all of the various types of receive data, following are the steps that are taken by the ATKProtocolReceivePacket routine.

1. Map incoming data to an ethernet frame and check the type field.
2. If the type field contains our custom INIC type then it should be TCP
3. If the header buffer specifies a fast-path connection, allocate one or more mbufs headers to map the header and possibly data buffers. Set the packet descriptor field of the mbuf to point to the packet descriptor, set the mbuf flags appropriately, queue the mbuf, and return 1.
4. If the header buffer specifies a slow-path connection, allocate a single mbuf header to map the network data, set the mbuf fields to map the packet, queue the mbuf and return 1. Note that we design the INIC such that we will never get a TCP segment split across more than one buffer.

5. If the type field of the frame indicates ARP or ICMP
6. Allocate a mbuf with a data buffer. Copy the contents of the packet into the mbuf. Queue the mbuf, and return 0 (not accepted).
7. If the type field is not either the INIC type, ARP or ICMP, we don't want it. Return 0.

The receive processing will continue when the mbufs are dequeued. At the moment this is done by a routine called ATKProtocolReceiveDPC. It will do the following:

1. Dequeue a mbuf from the queue.
2. Inspect the mbuf flags. If the mbuf is meant for fast-path TCP, it will call the fast-path routine directly. Otherwise it will call the ethernet input routine for slow-path processing.

4.6.3 Transmit

In this section we discuss the ATCP transmit path.

4.6.3.1 NDIS 4 send operation

The NDIS 4 send operation works as follows. When a transport/protocol driver wishes to send one or more packets down to an NDIS 4 miniport driver, it calls NdisSendPackets with an array of packet descriptors to send. As soon as this routine is called, the transport/protocol driver relinquishes ownership of the packets until they are returned, one by one in any order, via a NDIS call to the ProtocolSendComplete routine. Since this routine is called asynchronously, our ATCP driver must save any required context into the packet descriptor header so that the appropriate resources can be freed. This is discussed further in the following sections.

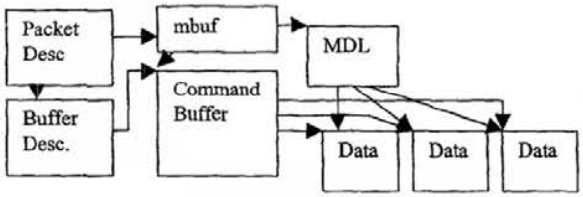
4.6.3.2 Types of "sends"

Like the Receive path described above, the transmit path is used not only to send network data, but is also used as a communication mechanism between the host and the INIC. Here are some examples of the types of sends performed by the ATCP driver.

4.6.3.2.1 Fast-path TCP send

When the ATCP driver receives a transmit request with an associated MDL, it will package up the MDL physical addresses into a command buffer, map the command buffer with a buffer and packet descriptor, and call NdisSendPackets with the corresponding packet. The underlying INIC driver will issue the command buffer to the INIC. When the corresponding response buffer is given back to the host, the INIC miniport will call NdisMSendComplete which will result in a call to the ATCP ProtocolSendComplete (ATKSendComplete) routine, at which point the resources associated with the send can be freed. We will allocate and use a mbuf to hold the command buffer. By doing this we can store the context necessary in order to clean up after the send completes. This context includes a pointer to the MDL and presumably some other connection context as well. The other advantage to using a mbuf to hold the command buffer is that it eliminates having another special set of code to allocate and return command buffer. We will store a pointer to the mbuf in the reserved section of the

packet descriptor so we can locate it when the send is complete. The following diagram illustrates the relationship between the client's MDL, the command buffer, and the buffer and packet descriptors.

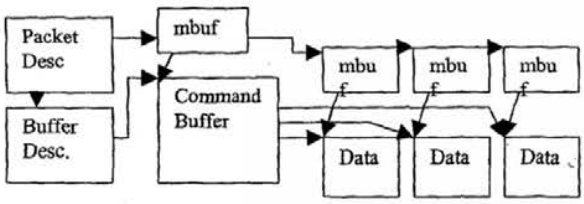


4.6.3.2.2 Fast-path TCP Receive

As described in section 4.3.1 above, the receive process typically occurs in two phases. First the INIC fills in a host receive buffer with a relatively small amount of data, but notifies the host of a large amount of pending data (either through a large amount of buffered data on the card, or through a large amount of expected NetBios data). This small amount of data is delivered to the client through the TDI interface. The client will then respond with a MDL in which the data should be placed. Like the Fast-path TCP send process, the receive portion of the ATCP driver will then fill in a command buffer with the MDL information from the client, map the buffer with packet and buffer descriptors and send it to the INIC via a call to NdisSendPackets. Again, when the response buffer is returned to the INIC miniport, the ATKSendComplete routine will be called and the receive will complete. This relationship between the MDL, command buffer and buffer and packet descriptors are the same as shown in the Fast-path send section above.

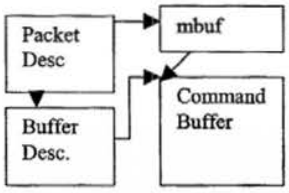
4.6.3.2.3 Slow-path (FreeBSD)

Slow-path sends pass through the FreeBSD stack until the ethernet header is prepended in ether_output and the packet is ready to be sent. At this point a command buffer will be filled with pointers to the ethernet frame, the command buffer will be mapped with a packet and buffer descriptor and NdisSendPackets will be called to hand the packet off to the miniport. In the illustration below we show the relationship between the mbufs, command buffer, and buffer and packet descriptors. Since we will use a mbuf to map the command buffer, we can simply link the data mbufs directly off of the command buffer mbuf. This will make the freeing of resources much simpler.



4.6.3.2.4 Non-data command buffer

The transmit path is also used to send non-data commands to the card. For example, the ATCP driver gives a context to the INIC by filling in a command buffer, mapping it with a packet and buffer descriptor, and calling NdisSendPackets.



4.6.3.3 ATKProtocolSendComplete

Given the above different types of sends, the ATKProtocolSendComplete routine will perform various types of actions when it is called from NDIS. First it must examine the reserved area of the packet descriptor to determine what type of request has completed. In the case of a slow-path completion, it can simply free the mbufs, command buffer, and descriptors and return. In the case of a fast-path completion, it will need to notify the TCP fast path routines of the completion so TCP can in turn complete the client's IRP. Similarly, when a non-data command buffer completes, TCP will again be notified that the command sent to the INIC has completed.

4.7 TDI Filter Driver

In a first embodiment of the product, the INIC handles only simple-case data transfer operations on a TCP connection. (These of course constitute the large majority of CPU cycles consumed by TCP processing in a conventional driver.)

There are many other complexities of the TCP protocol which must still be handled by host driver software: connection setup and breakdown, out-of-order data, nonstandard flags, etc.

The NT OS contains a fully functional TCP/IP driver, and one solution would be to enhance this so that it is able to detect our INIC and take advantage of it by "handing off" data-path processing where appropriate.

Unfortunately, we do not have access to NT source, let alone permission to modify NT. Thus the solution above, while a goal, cannot be done immediately. We instead provide our own custom driver software on the host for those parts of TCP processing which are not handled by the INIC.

This presents a challenge. The NT network driver framework does make provision for multiple types of protocol driver: but it does not easily allow for multiple instances of drivers handling the SAME protocol.

For example, there are no "hooks" into the Microsoft TCP/IP driver which would allow for routing of IP packets between our driver (handling our INICs) and the Microsoft driver (handling other NICs).

Our approach to this is to retain the Microsoft driver for all non-TCP network processing (even for traffic on our INICs), but to invisibly "steal" TCP traffic on our connections and handle it via our own (BSD-derived) driver. The Microsoft TCP/IP driver is unaware of TCP connections on interfaces we handle.

The network "bottom end" of this artifice is described earlier in the document. In this section we will discuss the "top end": the TDI interface to higher-level NT network client software.

We make use of an NT facility called a filter driver. NT allows a special type of driver ("filter driver") to attach itself "on top" of another driver in the system. The NT I/O manager then arranges that all requests directed to the attached driver are sent first to the filter driver; this arrangement is invisible to the rest of the system.

The filter driver may then either handle these requests itself, or pass them down to the underlying driver it is attached to. Provided the filter driver completely replicates the (externally visible) behavior of the underlying driver when it handles requests itself, the existence of the filter driver is invisible to higher-level software.

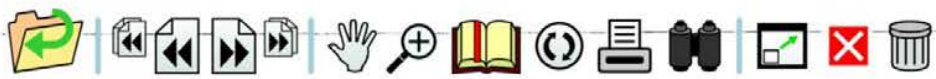
The filter driver attaches itself on top of the Microsoft TCP/IP driver; this gives us the basic mechanism whereby we can intercept requests for TCP operations and handle them in our driver instead of the Microsoft driver.

However, while the filter driver concept gives us a framework for what we want to achieve, there are some significant technical problems to be solved. The basic issue is that setting up a TCP connection involves a sequence of several requests from higher-level software, and it is not always possible to tell, for requests early in this sequence, whether the connection should be handled by our driver or by the Microsoft driver.

Thus for many requests, we store information about the request in case we need it later, but also allow the request to be passed down to the Microsoft TCP/IP driver in case the connection ultimately turns out to be one which that driver should handle.

Let us look at this in more detail, which will involve some examination of the TDI interface: the NT interface into the top end of NT network protocol drivers. Higher-level TDI client software which requires services from a protocol driver proceeds by creating various types of NT FILE_OBJECTs, and then making various DEVICE_IO_CONTROL requests on these FILE_OBJECTs.

There are two types of FILE_OBJECT of interest here. Local IP addresses that are represented by ADDRESS objects, and TCP connections that are represented by CONNECTION objects. The steps involved in setting up a TCP connection (from the "active", client, side) are:



(for a CONNECTION object)

1) Create an ADDRESS object.
2) Create a CONNECTION object.
3) Issue a TDI_ASSOCIATE_ADDRESS io-control to associate the CONNECTION object with the ADDRESS object.
4) Issue a TDI_CONNECT io-control on the CONNECTION object, specifying the remote address and port for the connection.

Initial thoughts were that handling this would be straightforward: we would tell, on the basis of the address given when creating the ADDRESS object, whether the connection is for one of our interfaces or not. After which, it would be easy to arrange for handling entirely by our code, or entirely by the Microsoft code: we would simply examine the ADDRESS object to see if it was "one of ours" or not.

There are two main difficulties, however.

First, when the CONNECTION object is created, no address is specified: it acquires a local address only later when the TDI_ASSOCIATE_ADDRESS is done. Also, when a CONNECTION object is created, the caller supplies an opaque "context cookie" which will be needed for later communications with that caller. Storage of this cookie is the responsibility of the protocol driver: it is not directly derivable just by examination of the CONNECTION object itself. If we simply passed the "create" call down to the Microsoft TCP/IP driver, we would have no way of obtaining this cookie later if it turns out that we need to handle the connection.

Therefore, for every CONNECTION object which is created we allocate a structure to keep track of information about it, and store this structure in a hash table keyed by the address of the CONNECTION object itself, so that we can locate it if we later need to process requests on this object. We refer to this as a "shadow" object: it replicates information about the object stored in the Microsoft driver. (We must, of course, also pass the create request down to the Microsoft driver too, to allow it to set up its own administrative information about the object.)

A second major difficulty arises with ADDRESS objects. These are often created with the TCP/IP "wildcard" address (all zeros); the actual local address is assigned only later during connection setup (by the protocol driver itself.) Of course, a "wildcard" address does not allow us to determine whether connections that will be associated with this ADDRESS object should be handled by our driver or by the Microsoft one. Also, as with CONNECTION objects, there is "opaque" data associated with ADDRESS objects that cannot be derived just from examination of the object itself. (In this case addresses of callback functions set on the object by TDI_SET_EVENT io-controls.)

Thus, as in the CONNECTION object case, we create a "shadow" object for each ADDRESS object which is created with a wildcard address. In this we store information (principally addresses of callback functions) which we will need if we are handling connections on CONNECTION objects associated with this ADDRESS object. We store similar information, of course, for any ADDRESS object which is explicitly for one of

our interface addresses; in this case we don't need to also pass the create request down to the Microsoft driver.

With this concept of "shadow" objects in place, let us revisit the steps involved in setting up a connection, and look at the processing required in our driver.

First, the TDI client makes a call to create the ADDRESS object. Assuming that this is a "wildcard" address, we create a "shadow" object before passing the call down to the Microsoft driver.

The next step (omitted in the earlier list for brevity) is normally that the client makes a number of TDI_SET_EVENT io-control calls to associate various callback functions with the ADDRESS object. These are functions that should be called to notify the TDI client when certain events (such arrival of data or disconnection requests etc) occur. We store these callback function pointers in our "shadow" address object, before passing the call down to the Microsoft driver.

Next, the TDI client makes a call to create a CONNECTION object. Again, we create our "shadow" of this object.

Next, the client issues the TDI_ASSOCIATE_ADDRESS io-control to bind the CONNECTION object to the ADDRESS object. We note the association in our "shadow" objects, and also pass the call down to the Microsoft driver.

Finally the TDI client issues a TDI_CONNECT io-control on the CONNECTION object, specifying the remote IP address (and port) for the desired connection. At this point, we examine our routing tables (see section XXX for details of routing) to determine if this connection should be handled by one of our interfaces, or by some other NIC. If it is ours, we mark the CONNECTION object as "one of ours" for future reference (using an opaque field which NT FILE_OBJECTS provide for driver use.) We then proceed with connection setup and handling in our driver, using information stored in our "shadow" objects. The Microsoft driver does not see the connection request or any subsequent traffic on the connection.

If the connection request is NOT for one of our interfaces, we pass it down to the Microsoft driver. Note carefully, however, that we can not simply discard our "shadow" objects at this point. The TDI interface allows re-use of CONNECTION objects: on termination of a connection, it is legal for the TDI client to dissociate the CONNECTION object from its current . Thus our "shadow" objects must be retained for the lifetime ADDRESS object, re-associate it with another, and use it for another connection of the NT FILE_OBJECTS: the subsequent connection could turn out to be via one of our interfaces!

4.7.1 Timers

4.7.1.1 Keepalive Timer

We don't want to implement keepalive timers on the INIC. It would in any case be a very poor use of resources to have an INIC context sitting idle for two hours.

4.7.1.2 Idle Timer

We will keep an idle timer in the ATCP driver for connections that are managed by the INIC (resetting it whenever we see activity on the connection), and cause a flush of context back to the host if this timer expires. We may want to make the threshold substantially lower than 2 hours, to reclaim INIC context slots for useful work sooner. May also want to make that dependent on the number of contexts which have actually been handed out: don't need to reclaim them if we haven't handed out the max.

5 Receive & Transmit Microcode Design

This section provides a general description of the design of the microcode that will execute on two of the sequencers of the Protocol Processor on the INIC. The overall philosophy of the INIC is discussed in other sections. This section will discuss the INIC microcode in detail.

5.1 Design Overview

As specified in other sections, the INIC supplies a set of 3 custom processors that will provide considerable hardware-assist to the microcode running thereon. The following lists the main hardware-assist features:

- header processing with specialized DMA engines to validate an input header and generate a context hash, move the header into fast memory and do header comparisons on a DRAM-based TCP control block.
- DRAM fifos for free buffer queues (large & small), receive-frame queues, event queues etc.
- header compare logic
- checksum generation
- multiple register contexts with register access controlled by simply setting a context register . The Protocol Processor will provide 512 SRAM-based registers to be shared among the 3 sequencers.
- automatic movement of input frames into DRAM buffers from the MAC Fifos.
- run receive processing on one sequencer and transmit processing on the other. This was chosen as opposed to letting both sequencers run receive and transmit. One of the main reasons for this is that the header-processing hardware can not be shared and interlocks would be needed to do this. Another reason is that interlocks would be needed on the resources used exclusively by receive and by transmit.
- The INIC will support up to 256 TCP connections (TCB's). A TCB is associated with an input frame when the frame's source and destination IP addresses and source and destination ports match that of the TCB. For speed of access, the TCB's will be

maintained in a hash table in NIC DRAM to save sequential searching. There will however, be an index in hash order in SRAM. Once a hash has been generated, the TCB will be cached in SRAM. There will be up to 8 cached TCBs in SRAM. These cache locations can be shared between both sequencers so that the sequencer with the heavier load will be able to use more cache buffers. There will also be 8 header buffers to be shared between the sequencers. Note that each header buffer is not statically linked to a specific TCB buffer. In fact the link is dynamic on a per-frame basis. The need for this dynamic linking will be explained in later sections. Suffice to say here that if there is a free header buffer, then somewhere there is also a free TCB SRAM buffer.

- There were 2 basic implementation options considered here. The first was single-stack and the second was a process model. The process model was chosen here because the custom processor design is providing zero-cost overhead for context switching through the use of a context base register, and because there will be more than enough process slots (or contexts) available for the peak load. It is also expected that all "local" variables will be held permanently in registers whilst an event is being processed.
- The features that provide this are:

  256 of the 512 SRAM-based registers will be used for the register contexts. This can be divided up into 16 contexts (or processes) of 16 registers each. Then 8 of these will be reserved for receive and 8 for transmit. A Little's Law analysis has shown that in order to support 512 byte frames at maximum arrival rate of 4 * 100 Mbits, requires more than 8 jobs to be in process in the NIC. However each job requires an SRAM buffer for a TCB context and at present, there are only 8 of these currently specified due to SRAM space limits. So more contexts (e.g. 32 * 8 regs each) do not seem worthwhile. Refer to Appendix A for more details of this analysis.

  A context switch simply involves reloading the context base register based on the context to be restarted, and jumping to the appropriate address for resumption.
- To better support the process model chosen, the code will lock an active TCB into an SRAM buffer while either sequencer is operating on it. This implies there will be no swapping to and from DRAM of a TCB once it is in SRAM and an operation is started on it. More specifically, the TCB will not be swapped after requesting that a DMA be performed for it. Instead, the system will switch to another active "process". Then it will resume the former process at the point directly after where the DMA was requested. This constitutes a zero-cost switch as mentioned above.
- individual TCB state machines will be run from within a "process". There will be a state machine for the receive side and one for the transmit side. The current TCB states will be stored in the SRAM TCB index table entry.
- The INIC will have 16 MB of DRAM. The current specification calls for dividing a large portion of this into 2K buffers and control allocation / deallocation of these buffers through one of the DRAM fifos mentioned above. These fifos will also be used to control small host buffers, large host buffers, command buffers and command response buffers.
- For events from one sequencer to the other (i.e. RCV ⇔ XMT), the current specification calls for using simple SRAM CIO buffers, one for each direction.
- Each sequencer handles its own timers independently of the others.
- Contexts will be passed to the INIC through the Transmit command and response buffers. INIC-initiated TCB releases will be handled through the Receive small

buffers. Host-initiated releases will use the Command buffers. There needs to be strict handling of the acquisition and release of contexts to avoid windows where for example, a frame is received on a context just after the context was passed to the INIC, but before the INIC has "accepted" it.

- T/TCP (Transaction TCP): the initial INIC will not handle T/TCP connections. This is because they are typically used for the HTTP protocol and the client for that protocol typically connects, sends a request and disconnects in one segment. The server sends the connect confirm, reply and disconnect in his first segment. Then the client confirms the disconnect. This is a total of 3 segments for the life of a context. Typical data lengths are on the order of 300 bytes from the client and 3K from the server. The INIC will provide as good an assist as seems necessary here by checksumming the frame and splitting headers and data. The latter is only likely when data is forwarded with a request such as when a filled-in form is sent by the client.

5.1.1 SRAM Requirements

The following are SRAM requirements for the Receive and Transmit engines:

TCB buffers	256 bytes * 16	4096
Header buffers	128 bytes * 16	2048
TCB hash index	16 bytes * 256	4096
Timers		128
DRAM Fifo queues	128 bytes * 16	2048
		~12K bytes

Depending upon the available space, the number of TCB buffers may be increased to 16.

5.1.2 General Philosophy

The basic plan is to have the host determine when a TCP connection is able to be handed to the INIC, setup the TCB and pass it to the card via a command in the Transmit queue. TCBs that the INIC owns can be handed back to the host via a request from the Receive or Transmit sequencers or from the host itself at any time.

When the INIC receives a frame, one of its immediate tasks is to determine if the frame is for a TCB that it controls. If not, the frame is passed to the host on a generic interface TCB. On transmit, the transmit request will specify a TCB hash number if the request is on a INIC-controlled TCB. Thus the initial state for the INIC will be transparent mode in which all received frames are directly passed through and all transmit requests will be simply thrown on the appropriate wire. This state is maintained until the host passes TCBs to the INIC to control. Note that frames received for which the INIC has no TCB (or it is with the host) will still have the TCP checksum verified if TCP/IP, and may split the TCPIP header off into a separate buffer.

5.1.3 Register Usage

There will be 512 registers available. The first 256 will be used for process contexts. The remaining 256 will be split between the 3 sequencers as follows:
257 – 320: 64 for RCV general processing / main loop.
321 - 384: 64 for XMT general processing / main loop.
385 – 512: 128 for 3rd sequencer use.

5.2 Receive Processing

5.2.1 Main Loop

The following is a summary of the main loop of Receive:

```
forever {
        while there are any Receive events  {
                if (a new event) {
                        if (no new context available)
                                ignore the event;
                }
                call appropriate event handler to service the event;
                this may make a waiting process runnable or set up
                a new process to be run (get free context, hddr buffer,
                TCB buffer, set the context up).
        }
        while any process contexts are runable {
                run them by jumping to the start/resume address;
                if (process complete)
                        free the context;
        }
}
```

5.2.2 Receive Events

The events that will be processed on a given context are:

- accept a context
- release a context command (from the host via Transmit)
- release a context request (from Transmit)
- receive a valid frame; this will actually become 2 events based on the received frame - receive an ACK, receive a segment
- receive an "invalid" frame i.e. one that causes the TCB to be flushed to the host
- a valid ACK needs to be sent (delayed ACK timer expiry).
- There are expected to be the following sources of events:
  1. Receive input queue: it is expected that hardware will automatically DMA arriving frames into frame buffers and queue an event into a RCV-event queue.
  2. Timer event queue: expiration of a timer will queue an event into this queue.
  3. Transmit sequencer queue: for requests from the transmit processor.

For the sake of brevity the following only discusses receive-frame processing .

5.2.3 Receive Details – Valid Context

The base for the receive processing done by the INIC on an existing context is the fast-path or "header prediction" code in the FreeBSD release. Thus the processing is divided into 3 parts: header validation and checksumming, TCP processing and subsequent SMB processing.

5.2.3.1 Header Validation

There is considerable hardware assist here. The first step in receive processing is to dma the frame header into an SRAM header buffer. It is useful for header validation to be implemented in conjunction with this dma by scanning the data as it flies by. The following tests need to be "passed":

- MAC header: destination address is our MAC address (not MC or BC too), the Ethertype is IP.
- IP header: header checksum is valid, header length = 5, IP length > header length, protocol = TCP, no fragmentation, destination IP is our IP address.
- TCP header: checksum is valid (incl. pseudo-header), header length = 5 or 8 (timestamp option), length is valid, dest port = SMB or FTP data, no FIN/SYN/URG/PSH/RST bits set, timestamp option is valid if present, segment is in sequence, the window size did not change, this is not a retransmission, it is a pure ACK or a pure receive segment, and most important, a valid context exists. The valid-context test is non-trivial in the amount of work involved to determine it. Also note that for pure ACKs, the window-size test will be relaxed. This is because initially the output PERSIST state is to be handled on the INIC.

Many but perhaps not all of these tests will be performed in hardware – depending upon the embodiment

5.2.3.2 TCP Processing

Once a frame has passed the header validation tests, processing splits based on whether the frame is a pure ACK or a pure received segment.

5.2.3.2.1 Pure RCV Packet

The design is to split off headers into a small header buffer and pass the aligned data in separate large buffers. Since a frame has been received, eventually some receiver process on the host will need to be informed. In the case of FTP, the frame is pure data and it is passed to the host immediately. This involves getting large buffers and dmaing the data into them, then setting the appropriate details in a small buffer that is used to notify the host. However for SMB, the INIC is performing reassembly of data when the frame consists of headers and data. So there may not yet be a complete SMB to pass to the host. In this case, a small buffer will be acquired and the header moved into it. If the received segment completes an SMB, then the procedures are pretty much as for FTP. If it does not, then the scheme is to at least move the received data (not the headers) to the host to free the INIC buffers and to save latency. The list of in-progress host buffers is maintained in the TCB and moved to the header buffer when the SMB is complete.

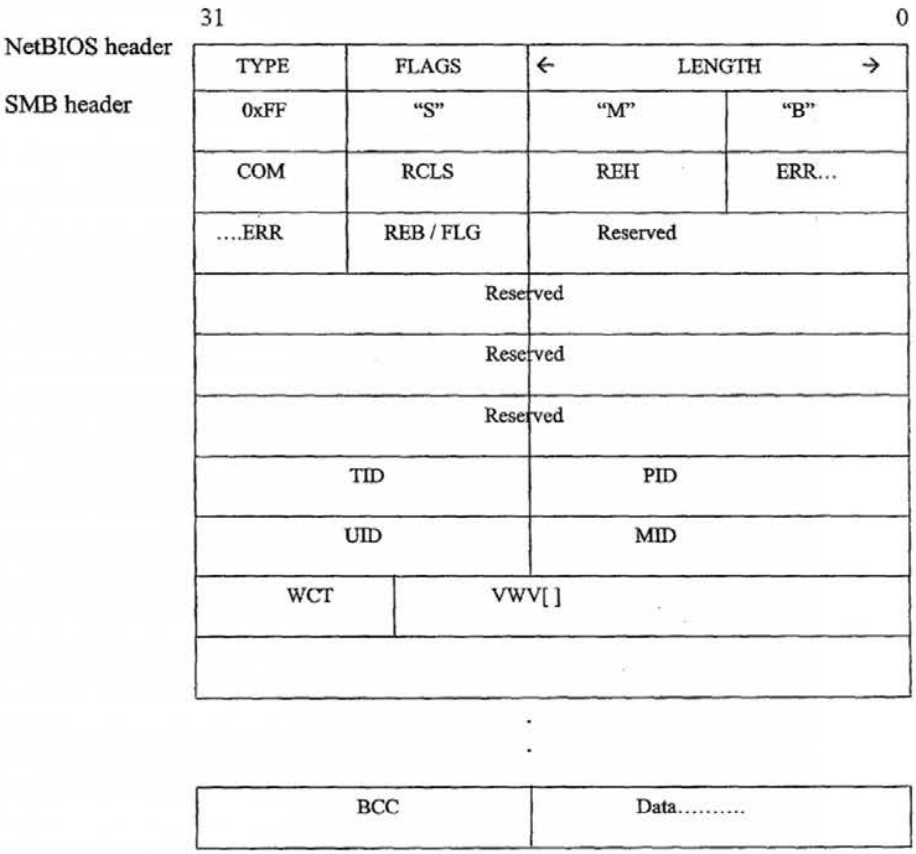
The final part of pure-receive processing is to fire off the delayed ACK timer for this segment.

5.2.3.2.2 Pure ACK

Pure ACK processing implies this TCB is the sender, so there may be transmit buffers that can be returned to the host. If so, send an event to the Transmit processor (or do the processing here). If there is more output available, send an event to the transmit processor. Then appropriate actions need to be taken with the retransmission timer.

5.2.3.3 SMB Processing

The following is the format of the SMB header of an SMB frame:



Notes (interesting fields):

LENGTH	17 bit Length of SMB message (0 – 128K)
COM	SMB command
WCT	Count (16 bit) of parameter words in VWV[]
VWV	Variable number of parameter words
BCC	Bytes of data following

The LENGTH field of the NetBIOS header will be used to determine when a complete SMB has been received and the header buffer with appropriate details can be posted to the host.
The interesting commands are the write commands: SMBwrite (0xB), SMBwriteBraw (0x1D), SMBwriteBmpx (0x1E), SMBwriteBs (0x1F), SMBwriteclose (0x2C), SMBwriteX (0x2F), SMBwriteunlock (0x14). These are interesting because they will have data to be aligned in host memory. The point to note about these commands is that they each have a different WCT field, so that the start offset of the data depends on the command type. SMB processing will thus need to be cognizant of these types.

5.2.4 Receive Details – No Valid Context

The design here is to provide as much assist as possible. Frames will be checksummed and the TCPIP headers may be split off.

5.2.5 Receive Notes

1. PRU_RCVD or the equivalent in Microsoft language: the host application has to tell the INIC when he has accepted the received data that has been queued. This is so that the INIC can update the receive window. It is an advantage for this mechanism to be efficient. This may be accomplished by piggybacking these on transmit requests (not necessarily for the same TCB).
2. Keepalive Timer: for a INIC-controlled TCB, the INIC will not maintain this timer. This leaves the host with the job of determining that the TCB is still active.
3. Timestamp option: it is useful to support this option in the fast path because the BSD implementation does. Also, it can be very helpful in getting a much better estimate of the round-trip time (RTT) which TCP needs to use.
4. Idle timer: the INIC will not maintain this timer (see Note 2 above).
5. Frame with no valid context: The INIC may split TCP/IP headers into a separate header buffer.

5.3 Transmit Processing

5.3.1 Main Loop.

The following is a summary of the main loop of Transmit:

```
forever {
        while there are any Transmit events  {
                if (a new event) {
                        if (no new context available)
                                ignore the event;
                }
                call appropriate event handler to service the event;
                this may make a waiting process runnable or set up
                a new process to be run (get free context, hddr buffer,
                TCB buffer, set the context up).
        }
        while any process contexts are runable {
                run them by jumping to the start/resume address;
                if (process complete)
                        free the context;
        }
}
```

5.3.2 Transmit Events

The events that will be processed on a given context and their sources are:

- accept a context (from the Host).
- release a context command (from the Host).
- release a context command (from Receive).
- valid send request and window > 0 (from host or RCV sequencer).
- valid send request and window = 0 (from host or RCV sequencer).
- send a window update (host has accepted data).
- persist timer expiration (persist timer).
- context-release event e.g. window shrank (XMT processing or retransmission timer).
- receive-release request ACK(from RCV sequencer).

5.3.3 Transmit Details – Valid Context

The following is an overview of the transmit flow:

The host posts a transmit request to the INIC by filling in a command buffer with appropriate data pointers etc and posting it to the INIC via the Command Buffer Address register. Note that there is one host command buffer queue, but there are 4 physical transmit lines. So each request needs to include an interface number as well as the context number. The INIC microcode will dma the command in and place it in 1 of 4 internal command queues which the transmit sequencer will work on. This is so that transmit processing can round-robin service these 4 queues to keep all 4 interfaces busy, and not