T2 Software Functional Specification

1. Introduction

This document describes the software for Typhoon 2, or **T2**, 3Com's 2nd generation "smart" NIC. It's a 100 MBPS Ethernet adapter. T2 includes all of the Typhoon logic, which borrows some logic from previous 3Com Ethernet Adapter ASICs (Cyclone/Hurricane), such as parts of the PCI interface and the MAC logic. Previous functions that were external to the Typhoon ASIC are integrated into T2, including the 128k external SRAM, and the IPsec encryption engine. Locgic external to T2 includes the serial flash ROM, magnetics, and an optional external SRAM or coprocessor.

The on-board processing engine performs intelligent packet processing and scheduling while reducing the load on the host processor.

Typhoon supports several new or enhanced features that improve network management or lower CPU utilization:

Management Features

- MAC Statistics
- Power Management - Support for the four PCI power states.
- Enhanced Remote Wakeup support with the added capability to generate keep-alive or heartbeat packets.
- Personal Firewall - The ability to throttle or restrict bandwidth on some traffic transmitted or received by the NIC.
- I^2C Bus Interface - This is a supporting feature for off-line diagnostics.
- Off-line diagnostics - The ability to gather and report information about the health and state of the host even when the host is not operational.

Performance Enhancements

- TCP/IP checksum generation and verification.
- TCP segmentation.
- Acceleration of Dynamic Access 2.0 features.
- Improved Host Interface.
- IPSec

2. Functional Description of Features

2.1 Management Features

2.1.1 Statistics

The processor on the adapter will keep statistics in memory. The Host Driver may view these statistics by:

- Requesting Statistics transfer to host memory.
- Timed update of statistic transfers to host memory.

The sources of the statistics are a combination of MAC registers, status descriptors and software derived statistics from Filter results.

The adapter will keep the following statistics: (dRMON Statistics are not addressed in this section. See dRMON for details.)

Transmit Statistics

TxPackets	The number of packets transmitted successfully without critical error.
TxBytes	The number of bytes transmitted successfully without critical error (does not include framing).
TxDeferred	The number of times that packet transmission had to defer to network traffic before transmitting successfully.
TxLateCollisions	The number of times a transmitted packet has collided after the first 64 bytes were transmitted.
TxCollisions	The number of packets transmitted successfully after a single collision.
TxMultipleCollisions	The number of packets transmitted successfully after more than one collision.
TxCarrierLost	The number of packets transmitted that experienced a loss of carrier.
TxExcessiveCollisions	The number of packets tossed due to more than 16 collisions.
TxFifoUnderruns	The number of time the transmit FIFO has gone empty while transmitting before the packet was completed.
TxBroadcast	The number of packets transmitted with a destination address of all one’s.
TxMulticast	The number of packets transmitted with a destination address having the multicast bit set.
TxOverflows	The number of packets tossed by the transmitter due to lack of Queue or Buffer space.
TxFiltered	The number of packets tossed by the transmitter due to a Filter directive.
TxBroadcastBytes	The number of octets transmitted in packets with a destination address of all one’s.
TxMulticastBytes	The number of octets transmitted in packets with a destination address having the multicast bit set.
*TxJabbers	The number of packets transmitted that were aborted because they were continuously transmitting.
Not Supported	
TxSqeErrors	Not Applicable.
TxReclaimErrors	Not Applicable.

Receive Statistics

RxPacketsGood	The number of packets received without error.
*RxPacketsTotal	The number of packets received including error packets.
RxBytesGood	The number of bytes received without error (excluding framing).
*RxBytesTotal	The number of bytes received including error packets (excluding framing).
RxFifoOverruns	The number of packets lost because the receive FIFO was full.
*RxFilteredByMac	The number of packets that are filter at the MAC when filters are set.
BadSSD	The total number of packets received with bad start-of-stream delimiter. This is only valid for 100BaseTx and 100BaseFx.
*RxRunts	The number of packets received that were less than 64 bytes.
RxCrcErrors	The number of packets received that contained a bad CRC.
RxOversize	The number of packets received that were greater than MTU.
RxBroadcast	The number of packets received with a destination address of all one’s.
RxMulticast	The number of packets received with a destination address having the multicast bit set.

RxOverflow	The number of packets received that were lost due to no buffer space available.
RxFiltered	The number of packets tossed by the receiver due to a Filter directive.
RxAlignmentErrors	The number of packets received that were not octet aligned.
RxDribbleBits	The number of packets received that contained dribble bits. Software selectable as forward or reject on error.

Statistic Sources

The following statistics are read from MAC registers:
TxPackets, TxBytes, TxDeferred, TxLateCollisions, TxCollisions, TxMultipleCollisions, TxCarrierLost, RxPacketsGood, *RxPacketsTotal, RxBytesGood, *RxBytesTotal, RxFifoOverruns, RxFilteredByMac, BadSSD

The following statistics are created from MAC Receive descriptors:
*RxRunts, RxCrcErrors, RxOversize, RxDribbleBits, RxAlignmentErrors

The following statistics are created from software interrupts from the MAC:
TxExcessiveCollisions, TxFifoUnderruns, *TxJabbers

The following statistics are created from software events or counts:
TxBroadcast, TxMulticast, TxOverflows, TxFiltered, RxBroadcast, RxMulticast, RxOverflow, RxFiltered

Other Statistics that can be derived from software, such as the outcome of packet classification, may be added as needed.

*May or may not be supported by MAC.

2.1.2 Power Management

Power Management supports four device states, D0 through D3. These states indicate the functional state of the adapter. Each state has a corresponding PCI bus state B0 though B3. Each bus state indicates the amount of power and functionality of the PCI bus. D0 & B0 are a fully functioning system which we will call the highest state. D3 & B3 are a fully powered down state, where the adapter may not have power, we will call this the lowest state. The transition of states D0 through D3 is performed through the driver. The transfer from B0 though B3 are PCI changes made to the PCI bus. A requirement for changing these states is that the adapter's device state must never be higher than the bus state. This is to say that before the PCI will move from D0 to and other state less than D0, the driver will be told to move the device state to a state that will be less than or equal to the impending bus state. Likewise when moving back toward D0, the bus state will move back toward B0 before the device state.

Typhoon supports all four device states, but D1, D2 and D3 are identical from a software functionality point of view:

<u>D0</u>

- The Adapter is ON, running and fully functional.

<u>D1, D2, D3 Hot & Cold</u>

- Power may be removed, if power is removed, it is expected that the host will assert RESET upon powering up. -B3 only
- No PCI bus transmission - B3
- No PCI bus reception - B3
- Wakeup events are supported as long as power has not been removed
- Sleep events (Enhanced Remote Wakeup) are supported as long as power has not been removed
- Changes in Power Management State are supported as long as power has not been removed
- The processor will reduce its clock rate to conserve power
- Promiscuous mode packet filtering is not supported.

<u>D3 Disabled – ARM Clock Stoped</u>

- PCI configuration state is preserved if power has not been removed
- No PCI bus transmission -B1
- No PCI bus reception - B1
- Wakeup Events are supported
- Changes in Power Management State are supported if power has not been removed
- No Wakeup or Sleep events supported
- MAC will be held in reset

Before the bus state is changed from B0 to B1-3, the driver will be told what state we are moving to and what to support while asleep, i.e. wake-up events. Before acknowledging this information, the driver will pass this on to the adapter and the adapter will prepare to go to sleep. Once ready, the adapter will notify the driver and the driver can acknowledge that we are in the appropriate driver state.

2.1.3 Enhanced Remote Wakeup

Wakeup Events

While the adapter is in any non-D0 state that does include Power to the ARM, it has the ability to tell the host to wake up from its sleep mode based on a network event. A wake-up event is the network indication that we should request the hardware and software external to the adapter to put the system into the power state BO (working) by asserting the PME# signal on the PCI bus. Before being placed in a sleep mode, the driver will tell the adapter what conditions to look for as a wake-up event.

The following wake up events will be supported:

- Change in Network Link State.
- Receipt of Network Wakeup Frame.
- Receipt of Magic Packet.
- Sleep Timer: Typhoon will wake the system at a user-defined time in the future.
- Receipt of an ARP request directed to this host.
- Receipt of an ICMP Echo Request directed to this host.

Sleep Events (Deep Sleep)

It may be necessary during adapter sleep to occasionally send out network packets to the keep route tables and windows browse masters updated. This is required so that when the system wakes up the network will be better prepared to begin forwarding packets or to keep devices updated so that the packet required to wake the system will be delivered.

The sleep events that will be supported in Typhoon are:

- MAC keep-alives – packets to keep Typhoon's MAC address alive in switches and multicast membership groups.
- ARP replies – Typhoon will answer ARP requests for its IP address.
- Ping replies – Typhoon will answer ICMP Echo Requests for its IP address
- Master Browser Host Announcements –Typhoon will announce its presence to the local Master Browser to maintain its position in the Network Neighborhood.
- DHCP Lease Renewal – Typhoon will maintain the DHCP lease on its IP address while asleep.
- Novell Watchdogs – Typhoon will respond to packets sent by a Novell server to maintain the client connection.

No encryption or decryption will be supported in sleep mode

2.1.4 Personal Firewall

This feature consists of 2 functions, broadcast throttling and DHCP prevention.

Broadcast Throttling

Broadcast throttling is implemented on both the transmit and receive data streams. The host can specify a transmit broadcast threshold as a percentage of the transmit bandwidth. The adapter monitors each data stream and discards any broadcast or multicast packets that exceed this limit.

The broadcast limit can be specified to an accuracy of 1%, Zero will discard all broadcast and multicast, 100 will discard none.

The adapter will implement this feature by calculating the broadcast usage over set time periods. For example, assume the adapter has an internal sample time period of 10ms, it is configured with a broadcast limit of 20%, and is transmitting at 100Mbs.

Total Tx in sample period:	1Mbit
Broadcast Tx limit in sample period:	200kbit

In this example the first 20kbits of broadcast or multicast data in any 10ms period is transmitted. Once this threshold is reached all broadcast or multicast packets are discarded until the next period is started.

DHCP Prevention

This feature prevents the PC from acting like a DHCP server. All receive packets are passed through a DHCP filter and are discarded if they are deemed to be destined for the DHCP process. DHCP packets are recognized as being UDP/IP packets to the well known DHCP port (67).

2.1.5 I^{2}C Serial Port Driver

An I^{2}C interface is supplied by Typhoon for the purpose of connecting support slave devices, such as the National Semiconductor LM78 Microprocessor System Hardware Monitor used for temperature, voltage and fan speed monitoring. Devices such as this are slave only devices that may be written and read by Typhoon for system diagnostic purposes. Software support will need to be added as devices are selected.

This Typhoon I^{2}C interface is a single master interface only, where Typhoon is the master. This is currently targeted for chip down applications.

2.1.6 Heartbeat

The Typhoon microcode provides a heartbeat signal to the host to indicate that the ARM9 is executing code. The ARM2Host Register 3 is continuously written with an incrementing 32 bit value by the ARM9. The rate of update of this register is on the order of 8ms per write, each write incrementing the value. The host can test for this register incrementing to determine if the ARM9 is not hung.

2.1.7 Off-line Diagnostics

Off-line Diagnostics refers to the ability to communicate with the adapter, from the network, for the purpose of running diagnostics. The intent of this communication is to create the ability to query the adapter about the state of the system. Off-line Diagnostics are not used to monitor an operational system. If the system is operational then communication between the network manager and the host can be used for system monitoring.

When the adapter boots it will load an image from non-volatile memory containing functionality to communicate over the network. Because the host may fail POST, the adapter must load this image at power-up rather than having the host download it to the adapter or having the host direct the adapter to load it from non-volatile memory. This will allow a diagnostic query/response even if the host POST fails.

When the adapter intercepts an Off-line Diagnostic request it can report the following:

- Report the current state of host/adapter communication
- Report last host value written to Port 80h
- Report diagnostic information for the adapter
- Report data from the host system memory (blue-screen)

Report the current state of host/adapter communication

The adapter can report the current state of host/adapter communication. When the diagnostic request is received, the adapter can send a ‘ping’ command to the host. If the host does not respond within a defined time period, then the adapter will report that the host is not communicating. The adapter could also determine that the host is not communicating if it is still waiting for the boot record from the host or if the host has not interrupted the adapter within a defined time period.

Report last Host value written to Port 80h

If the adapter determines that the host is not communicating then it could report the last value written by the host to Port 80h. When the host performs POST is uses I/O Port 80h to write a test point code value. If the host hangs during POST because of a terminal error condition then Port 80h will indicate the point at which the host halted execution. The Port 80h value will have to be decoded by either the adapter or the external application requesting the diagnostic report. The current proposal is for the external application to decode it because the adapter has limited storage. The main set of POST codes need to be identified which are common to the different BIOS versions and then determine if the adapter will have to report the BIOS vendor/version in the diagnostic report in order for the external application to decode the value.

Report diagnostic information for the Adapter

The adapter reports its POST result.

Report data from the Host system memory

If the adapter determines that the host is not communicating then it could read pertinent information from host system memory. The adapter could check whether or not the system timer is running and also report contents from the video buffer (blue-screen). The external application would have to interpret the contents.

There are a number of outstanding issues related to Off-line Diagnostics. The network management application (EdgeMonitor?) which will send the diagnostic request and interpret the response needs to be defined. The protocol (SNMP, AMP over UDP?), request and response can then be defined. Other issues can then be addressed such as whether or not the adapter should send unsolicited diagnostic reports (SNMP traps). Unsolicited reports would require the network management application to configure parameters on the adapter. Required parameters would include the destination address for unsolicited reports, an enable/disable diagnostic mode parameter, and an identifier for indicating the type of information which should be reported (i.e., only report host/adapter communication failure). Required parameters such as the destination address would need to be stored in the adapter's NV Device so that they are maintained through a reset. Another issue which needs to be addressed is whether or not the adapter will always intercept the diagnostic request or only look for the diagnostic request when it has detected that the host is no longer communicating? If the adapter ignores the request because the host is operational then who responds, the TDI?

There are no current plans for the following Off-line Diagnostic support:

- Report logged data. The adapter could be used to store data logged by an application. Pertinent information could be stored in non-volatile memory if it is useful after a power-cycle (the adapter will restrict the frequency and type of information stored to non-volatile memory). No application has been identified, maybe EdgeMonitor?

- Report diagnostic information from the devices on the I^2C interface. No devices are on the adapter's I^2C interface.

2.2 Performance Enhancements

2.2.1 TCP/IP Checksum Generation and Verification

IP Checksum

The processor on the adapter will support an IP checksum insertion and verification for packets using IPv4 for the IP forms of Ethernet Type 0x0800, 802.2 and SNAP.

Reference Microsoft document "Task Offloading in NDIS 5.0 for Windows NT5.0" for more details on IP Checksum Offloading.

TCP Checksum

The processor on the adapter will support a TCP checksum insertion and verification for packets using TCP/IP. The processor will calculate the checksum using the hardware checksums available through the packet DMA process. The checksums from the DMA will be corrected by software for data not included in the actual checksum calculation.

For transmit packets, the TCP Pseudo Header Checksum should be in the TCP checksum field.

Reference Microsoft document "Task Offloading in NDIS 5.0 for Windows NT5.0" for more details on TCP Checksum Offloading.

NOTE: The adapter will not support the insertion or verification of TCP checksums on fragmented IP packets since it will not store the packet for fragmentation or re-assembly.

2.2.2 TCP Segmentation

TCP can pass a buffer, containing a "large datagram" bigger than the MTU of the medium and the MSS, to be transmitted. The goal is to reduce host CPU utilization and number of interrupts, and to improve performance and scalability.

A session MSS will be passed down to the adapter along with the datagram, telling Typhoon the segment size of the TCP payload for the packet transmission. The size of the datagram first datagram is specified in the IP Total Length field. From this value the MSS could be extracted through calculations, however, the MSS is pass to the driver through the `NDIS_PER_PACKET_INFO_FROM_PACKET` structure. The total Length of the packet and MSS will be sent to the adapter in a TCP option header.

The typhoon adapter will not support MSS less than 4 bytes. The driver should fail packets with an MSS less than 4 byte without sending to the adapter.

The large datagram will be passed to Typhoon as an oversized MAC frame, with template frame header, IP header and TCP header. TCP protocol can pass down only payload of size within the advertised window of the other side. Typhoon will handle the large datagram as a shorthand request to transmit a series of normal TCP datagrams, with limited protocol handling for each segment of transmitting. For the handling of large datagram sends interleaving small sends (regular sends), the adapter maintains an order-preserving queue for each session involving large sends.

NOTE: The receive ACK, retransmission, and window maintenance will not be handled by Typhoon.

Typhoon takes the incoming datagram and carves it up into fragments based on the MSS passed down. All packets but the last will have a payload exactly MSS bytes in size. The last may be smaller.

The header of outgoing packets is derived from the template header on the incoming datagram as follows:

1) Unless otherwise specified, fields are simply copied from the template to the outgoing packet without change.
2) The MAC header is used exactly.
3) The IPV4 header is copied exactly, except for IP total length, which will be computed correctly for each datagram transmitted, and the IP identification (ID) field, which is advanced for each packet, with the first packet's ID copied from the template.
4) The IP header checksum is computed for each datagram transmitted.
5) The TCP header is copied exactly, except the sequence number (SEQ), which will be advanced correctly for each datagram transmitted, with the first SEQ copied from the template and succeeding ones advanced by MSS bytes.
6) The FIN and PUSH bits are set to zero for all but the last packet. These bit will be set to the value in the template on the last packet in the datagram transmitted.
7) The TCP checksum is computed for each datagram transmitted.
8) IP and TCP Options will be allowed. The adapter will not alter their value.

Reference Microsoft document "Task Offloading in NDIS 5.0 for Windows NT5.0" for more details on TCP Segmentation Offloading.

2.2.3

2.2.4 IEEE 802.1p/Q Class of Service Priorities

Transmit priority levels will be supported by the adapter. The current plans are to support 8 priority levels internal to Typhoon. Packets will be transmitted over the wire in priority order, not in the order received from the host. The prioritizing of the packets is specified by the host and contained in the Priority field in the 802.1 Q tagged frame.

In addition to the tagging service on a per-packet basis, Typhoon supports COS (Class of Service) policy negotiation for establishing COS interoperability with remote hosts. Typhoon maintains a table of remote host addresses with which Typhoon has a bilateral understanding of COS tagging. COS tagging is performed only for those packets to or from those hosts which appear in the table.

Typhoon initiates the COS negotiation when an outbound packet with a tagging request is first received that is destined to a remote host which is not listed in the COS table. The negotiation is performed via ICMP Ping messages (Details TBD).

Typhoon forwards all traffic to the remote host without tagging prior to the completion of COS negotiation.

2.2.5 IPSec

Typhoon will assist the host in performing various security services for traffic at the IP layer, in the IPv4 environment.

Typhoon provides support for IPSec as a BITS (Bump-in-the-stack) implementation

Typhoon's support for IPSec conforms to the following draft standards:

draft-ietf-ipsec-arch-sec-02.txt	Obsoletes RFC 1825
draft-ietf-ipsec-auth-header-03.txt	November 1997
draft-ietf-ipsec-esp-v2-02.txt	November 1997

2.2.5.1 Off-loading IPSec processing to Typhoon

Typhoon provides support only for calculation-intensive part of IPSec, such as ICV (Integrity Check Value) calculation and encryption/decryption. Typhoon expects the host itself to handle all other protocol related tasks, such as key management, Security Policy and Security Association establishment, PMTU discovery, and traffic filtering prior to enforcing various security policies.

Due to its limited buffer capacity, incoming fragmented IP datagrams will not be re-assembled on the adapter unless they are less than 2 fragments. No IPSec processing will be done for the other general cases. However, Typhoon makes no assumption about the fragmentation status of the outbound traffic that is to be processed by the Ipsec logic, which may be fragmented (in tunnel mode) if the host chooses to handle them. Correspondingly it makes no assumption about the fragmentation status of the inbound packets which appear after the Ipsec processing.

To request for Typhoon's assistance in IPSec processing, the requester must provide place holders for all AH headers, ESP headers, and all authentication data, properly embedded within the datagram per IPSec specification. Typhoon also expects all explicit DES padding except in the case of a large TCP datagram that is to be segmented by the adapter.

For outgoing traffic for which IPSec processing is requested, Typhoon expects all the Security Associations have been resolved by the host, which include the key management, Security Association establishment and creation, and the traffic classification to determine the proper Security Association. Typhoon will supply the IV (Initialization Vector, randomly selected per packet. Typhoon will not attempt to extend the CBC across the packets), calculate the Integrity Check Value (ICV), and encrypt data for ESP or BITW request. The SA selector matching and filtering, the assignment of the SPI value, the maintenance of anti-replay sequence number and windows, and all necessary DES padding are the responsibility of the requester.

For incoming traffic for which IPSec processing is requested, the adapter will generate the ICV and decrypt them per Security Association identified by the triplet <protocol-type, destination-address, SPI>. The SA selector matching and filtering and the enforcement of anti-replay policy are the responsibility of the requester. Note also that in Phase I the adapter performs “Crypto-only” functions for IPSec. The parsed IPSec headers, if nested, may not match the order required by the Security Policy, that must be re-parsed and checked by the host to enforce the relevant Security Policies.

Limited auditing will be provided by Typhoon for traffic on which the on-board IPSec processing is performed. All auditing reports are limited to local node. There is no support for the receiver to transmit any message to the purported transmitter in response to the detection of an auditable event.

2.2.5.2 Combination of Security Associations

Typhoon supports the host both as an IPSec host and/or an IPSec Security Gateway.

For outbound traffic, Typhoon supports the following combination of AH and ESP headers:

1. Tunneling of clear-text plain packet.
2. Basic IPSec packet: This is a combination of AH and ESP in the following two modes:

<u>Transport</u>	<u>Tunnel</u>
1. [IP1] [AH] [upper]	4. [IP2] [AH] [IP1] [upper]
2. [IP1] [ESP] [upper]	5. [IP2] [ESP] [IP1] [upper]
3. [IP1] [AH] [ESP] [upper]	6. [IP2] [AH] [ESP] [upper]

3. Tunneling of Basic IPSec Packets (defined in 2. above) which have already undergone one level of IPSec processing.

For inbound traffic, Typhoon supports any combination of AH and ESP modes, as long as the Security Association of each triggered IPSec processing is unambiguously defined. However, only one level of nesting is supported per single round of IPSec processing on the adapter.

Additional nesting of Security Associations, for both the inbound and outbound traffic, can be accomplished by utilizing Typhoon as a BITW outboard crypto-processor to create the IPSec header calculation, prior to the transmitting of the IPSec packets.

2.2.5.3 Performance consideration

Typhoon utilizes the on-board hardware (Sidewinder) to do most of the IPSec processing. Sidewinder is capable of performing a combination of HASH and DES in one pass. For example, for outbound traffic in transport mode with an AH and an ESP without ICV, the IPSec processing requires only a single pass through the Sidewinder. However, for multiple level of nesting or combination of an AH and an ESP with ICV, multiple passes through the Sidewinder are required. The performance of the IPSec processing is strictly a function of the number of Sidewinder passes needed to complete the processing.

For properly aligned packets, Sidewinder can deliver around 200 mega-bps per pass at 80 mega-Hz clock.

Examples:
Packet of format [IP1] [ESP with ICV] [Upper] require 1 pass.
Packet of format [IP1] [AH] [ESP without ICV] [Upper] require 1 pass.
Packet of format [IP2] [ESP with ICV] [IP1] [AH] [ESP with ICV] [Upper] require 3 passes.

2.2.5.4 Conformance requirements

The following mandatory-to-implement algorithms are supported:

- HMAC with MD5 [C. Madson & R. Glenn, "The Use of HMAC-MD5-96 within ESP and AH", Internet Draft, 7/2/97.]

- HMAC with SHA-1 [C. Madson & R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH", Internet Draft, 7/2/97.]

- DES in CBC mode [C. Madson & N. Doraswamy, "The ESP DES-CBC Cipher Algorithm With Explicit IV", Internet Draft, 07/02/1997."]

2.2.5.5 IPSec datagram flow description

The passing of network packet through the TCP/IP stack, the NDIS driver, and Typhoon can be characterized as follows:

1. On initialization, the Typhoon NDIS driver will declare its IPSec capabilities through the NDIS_TASK_IPSEC structure as follows:

   ? AH+ESP combinations supported.

   ? IP options supported

   ? Query SPI supported

   ? For AH, MD5, SHA-1 are supported for both the Tunnel and Transport Mode, and for both Send and Receive. Only Ipv4 is supported (Ipv6 is not supported).

   ? For ESP, DES and Triple-DES are supported for both the Tunnel and Transport Mode, and for both Send and Receive. Only Ipv4 is supported (Ipv6 is not supported).

   ? Only CryptoOnly is supported
2. Typhoon also has a built-in hardware digital non-deterministic random number generator. When requested, Typhoon can provide a specified set of random numbers to the host, which can be a basis of various key generation and random number seeding.
3. Prior to the sending and receiving of IPSec network packets, the Stack must establish its Security Policy Database (SPD) and Security Association Database (SAD). Part of the SAD will be passed down to the NDIS Driver through OIDs, one for the creation of a SA (OID_TCP_TASK_IPSEC_ADD_SA) and one for the deletion of a previously created SA (OID_TCP_TASK_IPSEC_DELECT_SA). The OIDs will be used to maintain the SAD inside the Driver during the on-line operation.
4. The NDIS Driver strips off the Selector part of the SAD (retains only the Destination Address specification) and passes down to Typhoon only information related to the HASH and DES algorithms, which include the algorithm descriptor, the keys, and the padding convention. The adapter produces IPAD keys and OPAD keys in preparation for the HASH operation. For the adapter, a SA is identified either through the SA id on sending, or the triplet <destination address, SPI, ESP or AH> on receiving traffic. Note that as far as the adapter is concerned, the SPI is only meaningful for the inbound traffic. The creation and deletion of SA for Typhoon is typically carried out in-band. (For Phase I, the adapter is used as a outboard crypto-engin that does not perform IPSec selector functions. The identification of SA for the outbound traffic is also done through the SPI.)
5. Once the SAD is established on both the host side and the Typhoon side, the IPSec traffic is then allowed to flow. In the following example, we use a TCP datagram in Transport mode as an example.

6. On sending, the Stack prepares all the headers for IPSec based on its SPD and SAD. Since Typhoon is declared as CryptoOnly, all buffer space for AH, ESP, and trailing padding and HASH value is pre-allocated by the Stack. In addition to the IP header and TCP header, the Stack is also responsible for the following task:

   ? In AH, it fills the Next Header field.

   ? The TCP datagram is encapsulated as an ESP, with the TCP header embedded at offset 8+length-of-IV (8 bytes). The host is responsible for allocating space for all padding bytes in all cases.

7. The NDIS Driver accepts the outbound network packet (in clear text), figures out the mode and nesting requirement (Transport, Tunnel, or combination) based on the IP header, AH header, and ESP format, and for each AH or ESP, searches for the SA in its SA Database based on the following Selectors (This is performed per outbound packet):

   ? Source Address

   ? Destination Address

   ? Transport Protocol

   ? Source Port (in TCP header)

   ? Destination Port (in TCP header)

   ? AH or ESP

   ? TOS (Type of Service)

   ? Data Sensitivity Level (IP option, IPSO/CIPSO labels)

8. After identifying the proper Security Associations, one per AH or ESP, the Driver then performs the following tasks:

   ? It fills the Payload Length, SPI and Sequence Number Fields in the AH per identified SA.

   ? It fills the SPI, the Sequence Number, Pad Length, all the Padding bytes, and Next Header in ESP per identified SA.

   ? It formulates an *Option Descriptor* containing the identifier of the identified SA's in an array. The multi-fragment frame will then be placed in the Tx Descriptor Ring.

9. The adapter will retrieve the IPSec network packet from the Tx Descriptor Ring, walk down the packet to parse the various IP headers and to determine the nesting levels, and based on the information passed in via the *Option Descriptor*, identify all the SA's. Based on the SA's, the adapter formulates a request to Sidewinder with the following data:

   ? IV (Initialization Vector, 64 bits. This is for DES only.) The adapter provides a software random number generator for IV. It initially seeds the IV with a hardware-generated random number produced by the Sidewinder's Randomizer. It then saves the last 64-bit of the DES or HASH output and uses it as the IV for the succeeding round of DES. The adapter periodically re-seeds the generation of IV with new random number produced by the Sidewinder.

- ? DES key (64 bits).
- ? 2-key Triple-DES (128 bits) or 3-key Triple-DES (192 bits).
- ? HMAC IPAD/OPAD keys, four 32-bit words for MD5, five 32-bit words for SHA-1
- ? HASH offset and length.
- ? DES offset and length.
- ? Op code to indicate the algorithm.

10. For AH processing, prior to issuing the request to Sidewinder, the adapter must save all the “mutable” fields, plug in the predictable value expected at the receiving side if predictable, and zero out the rest for the HASH processing. It restores those fields after the Sidewinder processing. The mutable fields are:
    - ? Destination Address (with loose or strict source routing), mutable but predictable
    - ? TOS
    - ? Flags
    - ? Fragment Offset
    - ? TTL
    - ? Header Checksum
    - ? Options (Each individual options must be check for mutability. If an option is classified as mutable, the entire option is zeroed for ICV computation purposes.)
11. For HASH, the Sidewinder outputs a Digest of equal length as the IPAD/OPAD keys. The adapter truncates the Digest to a 96-bit value before placing it into the packet.
12. Depending on the depth of nesting, the adapter may feed the same IPSec packet through Sidewinder several times. However, in most common cases, namely the ESP with Authentication data in Transport mode, one round through the Sidewinder will accomplish both the DES and HASH.
13. After the IPSec processing, the network packet is put on the wire like all other normal non-IPSec packet. Note that the adapter assumes that the final frame length is within the MTU of the local Ethernet. In other words, the Stack and the Driver is responsible for ensuring the final IPSec packet is within the MTU constraints.
14. For inbound traffic, IPSec processing is the reverse of the above logic, with the following exceptions:
    - ? The adapter does not have the ability to handle the general case of fragmented IP packet. If the Fragment Offset is non-zero, or the More Fragments bit is set, the packet is considered fragmented. If the number of fragments is not more than 2, then the adapter will attempt to put them together before applying the IPSec processing. All other cases are “punted” to the host. In other words, the host is expected to handle the re-assembly. The host must have a software implementation of IPSec so that the fragmented IP packets can be handled in the host.

? For non-fragmented network packet, the adapter uses the triplet <Destination Address, SPI, AH or ESP> within the packet to look up its local copy of SAD. Once the SA is found, the adapter will perform the HASH or DES accordingly. This look-up is performed per packet.

? The result of HASH and DES is returned directly via the response frame.

? The host is expected to verify the Sequence Number per SA. The NDIS driver is also expected to apply the Selector per SA to further filter out unwanted packets. Note that since the adapter is CryptoOnly and does not handle the IPSec Sequence Number, it can not regulate the traffic to prevent the Denial of Service attack.

3. Theory of Operation

3.1 Host Adapter Interface

Information is passed between the host and the adapter using shared data structures in the host's memory.

3.1.1 Packet Type

There are two types of information passed between the host and the adapter:

Network Packets	Contains either Transmit or Receive data framed for network transmission. Data in a network packet is formatted in the standard *network byte order*.
Control Packets	Contains meta-information, which is for initializing or setting of operation environment and for maintaining context and state that controls the packet handling. It is either a command, or a response to a previously issued command.

3.1.2 Data Format and Alignment Convention

The adapter card stores data in little endian format as does the Intel based. Therefore there is no need to convert the data structure in the control information package between the two. Control data must be aligned on a 32-bit word boundary on both platforms in order to avoid *endianess* conversion. However, since network byte order is specified as big endian, the adapter must convert any data item in the packet data into the correct byte order if it needs to manipulate it as integers.

For IP packet data, the IP header must also be aligned on a 32-bit word boundary in adapter memory so that all IP header structures can be accessed conveniently (after *endianess* conversion). However, since the MAC header is of size either 14 bytes or 22 bytes, it must be aligned on an *odd* 16-bit boundary so that the IP header can be correctly aligned. Other than the processing of VLAN tag or large TCP packets, the adapter is not expected to break up the packet or move around parts of it.

The control data is identified by a *command code*, followed by the Sequence number and one or more parameters. The *command code* indicates the kind of services requested by the command. The *Sequence number* is initialized by the issuer of the command and is carried by the response in responding to the original command. However, it is treated as opaque data to the receiver and must not be altered.

Control data is passed either in-band or out-of-band. In-band control data is synchronized to the packet data stream and must be handled in the order received relevant to the packet data. Its execution sequence within the packet data stream affects the function of the In-band control data. Due to the in-band nature of the control data, the processing duration should be reasonably short in order not to delay normal flow of packet data traffic.

3.2 Host Adapter Interface Ring Buffer

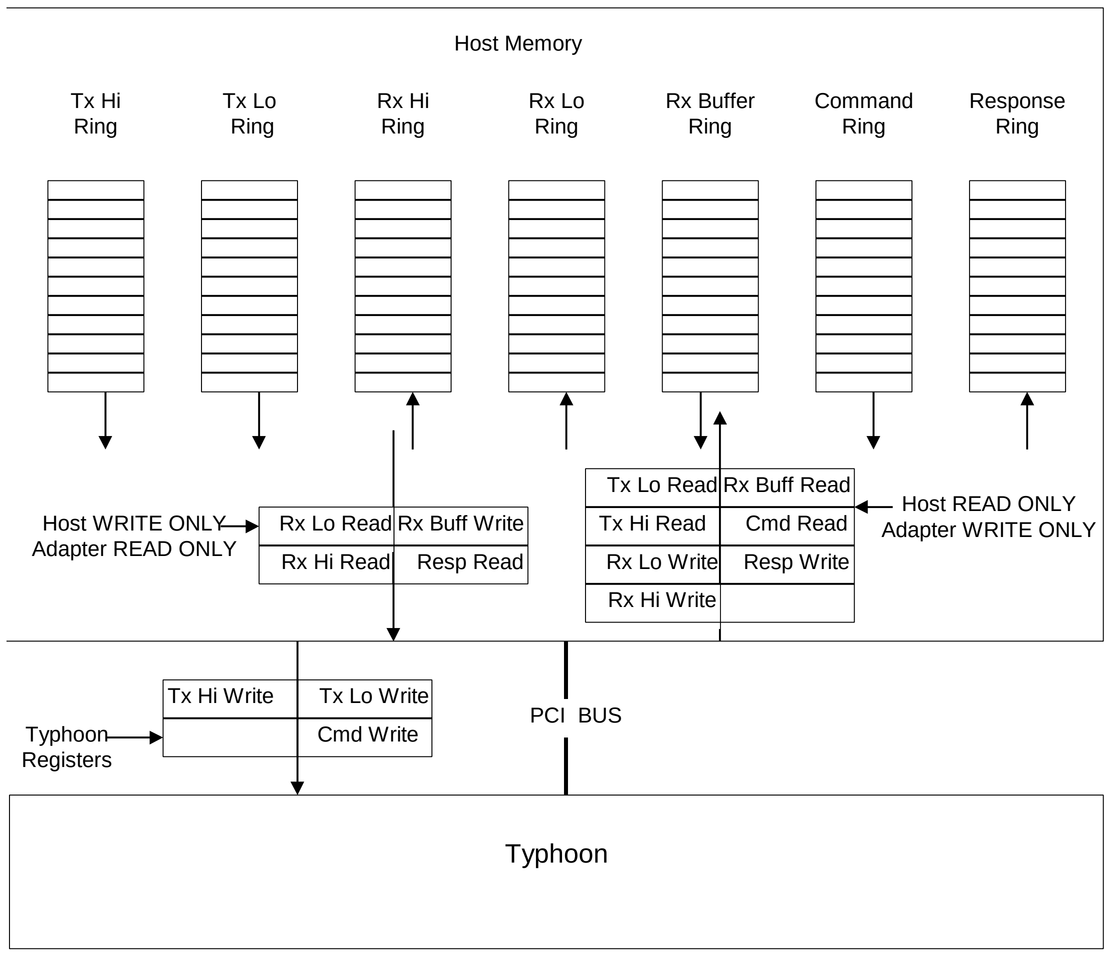
A number of ring buffers are set up in the host memory, which are dedicated for either host-to-adapter or adapter-to-host communication.

In the case of a ring buffer used for host-to-adapter data transfer, 2 index variables, a write and a read index are used to manage the ring. The host writes data into the ring and updates the write index to inform the adapter of the new data. The adapter reads data from the ring and updates the read index to inform the host that the space is available for re-use.

There are 6 ring buffers defined, 2 for transmit, 2 for receive and 2 for control, they are:

Tx Rings (Hi & Lo priority)	contain packet descriptors for packets to be transmitted or control commands to be executed in-band through the adapter.
Rx Rings (Hi & Lo priority)	contain rings of descriptors that the adapters writes to give the host access to received packets, to return response data or status for previously issued host command, or to issue a request to the host from the adapter.
Rx Buffer Ring	contains pointers to maximum size empty buffers that the adapter can use to store received packets.
Command Ring	contains command descriptors for host-to-adapter control information.
Response Ring	contains response descriptors for adapter-to-host control information.

All 7 rings and the index variables are shown in the following diagram. Note that only 3 of the indexes use registers, the other 11 are part of host resident data structures.



All rings have the following restrictions:

- Rings have base addresses of 64 bits, but no ring can traverse a 32 bit address boundary, i.e. the top 32 bits of the address of any element in a ring is constant.
- The rings must always have at least one empty location. This is due to the fact that when the read and write indexes are equal the ring is empty. The write index must never be incremented such that it equals the read index when the ring is full.

The index variables into the 7 rings are grouped into 2 data structures in host memory, one containing the data that is read-only to the adapter, the other that is write-only to the adapter. All but 3 of the indexes are stored in these structures. The other 3 indexes, *Tx Hi Write Index, Tx Lo Write Index* and *Command Write Index* are stored in a dedicated Host-to-Arm register.

The reason for 2 different forms of storing the indexes is to ensure immediate action on some and more delayed on others. Also host register writes are relatively efficient and can be used to interrupt the adapter, but Arm-to-Host register operations are inefficient and are avoided. The indexes in registers cause interrupts to the ARM when written by the host. The Host-to-Arm indexes stored in host memory are periodically polled (using DMA) by the adapter to determine if they have been updated by the host. The Arm-to-Host indexes are periodically updated by the adapter using DMA writes.

The following register assignments are used (note that since the indexes are 16 bits, each register contains 2 index values).

Host to Adapter Registers:

Register	**Runtime**
Register 0:*	unused
Register 1:*	*PCI Tx Hi Write Index,*
Register 2:*	*PCI Cmd Write Index*
Register 3:*	*PCI Tx Lo Write Index*
Register 4:	unused
Register 5:	unused
Register 6:	unused
Register 7:	unused

* These registers cause an ARM interrupt when written by the host.

Adapter to Host Registers

Register	**Runtime**
Register 0:	*PCI Status Register*
Register 1:	unused
Register 2:	unused
Register 3:	unused

The 2 data structures in host memory that contain the ring indexes that are not stored in memory are described below. The **HostToArm** structure is READ ONLY to the Typhoon, the **ArmToHost** structure is WRITE ONLY to the Typhoon.

ArmToHost data structure

32bit:	RxHiRead
32bit:	RxLoRead
32bit:	RxBuffWrite
32bit:	RespRead

HostToArm data structure

32bit:	TxLoRead
32bit:	TxHiRead
32bit:	RxLoWrite
32bit:	RxBuffRead
32bit:	CmdRead
32bit:	RespWrite
32bit:	RxHiWrite

3.3 Packet Flow Convention

Due to its limited storage space, all packet buffers reside in the host memory. The function and structure of those interface ring buffers reflect this asymmetry.

The passing of network packets generally does not expect a response or status return. The transmit buffer will be returned to the host after the adapter has transmitted the packet, to allow the host to free the buffer memory.

On the other hand, the passing of control packets usually follows the request/response pattern. The issuer of the request must properly prepare all the buffers to house both the input parameters and the response (output) data.

The passing of packet through the ring buffers can be characterized as follows:

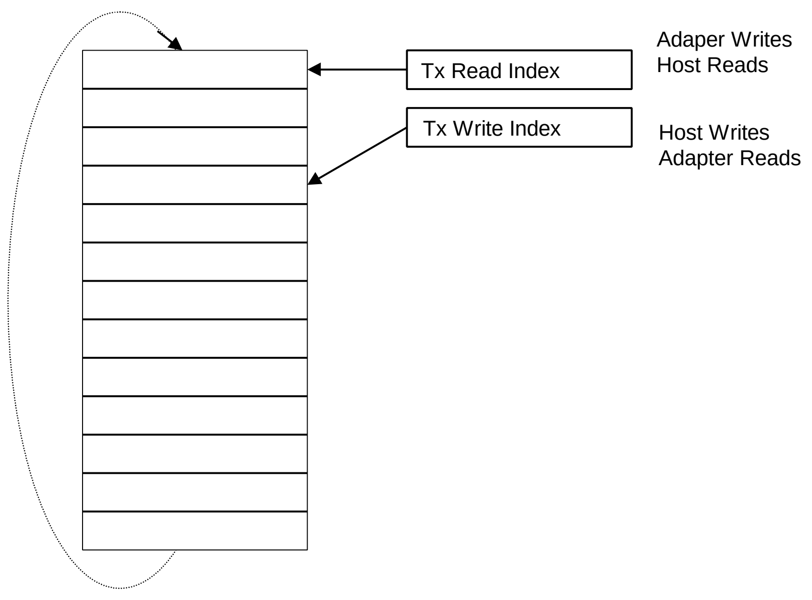
- Transmit network packets are passed from the host to the adapter through one of the *Tx Rings.*
- Host-to-adapter control packets can be passed to the adapter either in-band or out-of-band. The general case is to pass control information using the out-of-band data path, which uses the *Command Ring*. Control information that is synchronized to the data stream may be passed in-band through the *Tx Rings.* In either case the response is returned using the *Response Ring.*
- Received network packets are passed from the adapter to the host through the *Receive Ring*. The adapter must acquire a buffer from the host through the *Rx Buffer Ring* before it can send the packet to the host.
- Adapter-to-host control packets are relatively rare and typically do not expect response data to be returned. The adapter uses the *Response Ring* to pass the control packets to the host.
- The response data to an adapter-to-host control packet, if any, will be returned through the *Response Ring*.

3.3.1 Host-to-adapter Packet Flow

Packets are transmitted to the adapter by the host writing descriptors into shared ring buffers in host memory. Access to the ring buffer is controlled by two index variables, which are stored in host memory. The *Tx Write Indexes* and the *Tx Read Indexes* are used as follows.

The indexes are initialized to point to the start of an empty ring (initialized to zero). The *Tx Write Indexes* point to the next location in the ring into which the host will write data, they are never written by the adapter. The *Tx Read Indexes* point to the next descriptor to be read by the adapter, they are updated by the adapter after the descriptor has been copied to internal adapter memory and are never by written by the host except during initialization. The 2 indexes being equal indicate the ring is empty. The write index is never incremented to equal the read pointer for a full ring. This implies that the ring can not be completely filled and at least one entry must remain free to prevent the 2 indexes becoming equal.

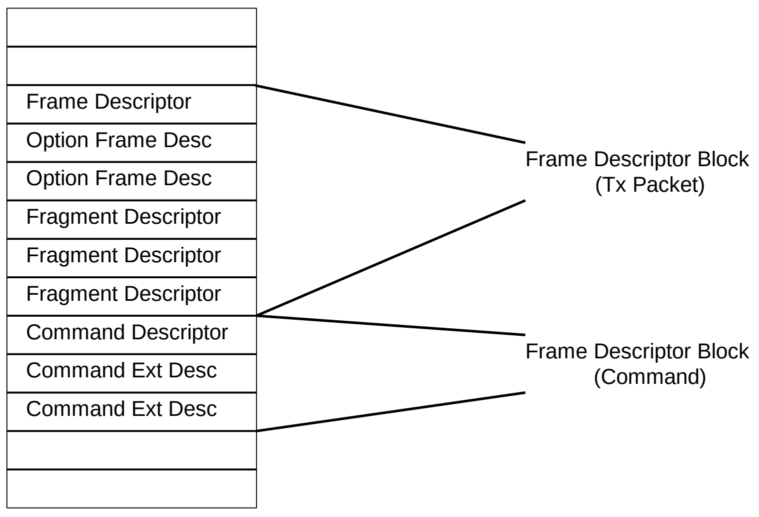
To transmit a frame the host writes a frame descriptor and multiple fragment descriptors into the ring. It then updates the write index and writes the value to the *PCI Tx Write Index* register. This interrupts the ARM which copies the complete *Packet Descriptor Block* to the adapter.



Tx Ring

Transmit frames are described by a sequence of descriptors formed into a *Frame Descriptor Block*. A *Frame Descriptor Block* describes a single frame and consists of one *Frame Descriptor* followed by one or more *Option Frame Descriptors* followed by one or more *Fragment Descriptors.*

A *Frame Descriptor Block* may wrap in the *Tx Ring*. The ring will be sized such that individual descriptors will never wrap, i.e. each data structure will always be in contiguous memory. This is achieved by defining all descriptors to be the same size (16 bytes) and ensuring that the ring is a multiple of the descriptor size.



Tx Descriptor Ring

Responses To Transmitted Packets

In the general case for transmitting packets, the following sequence is followed:

1. The host transmits a packet to the adapter.
2. The adapter transmits the packet over the network and informs the host that the buffer can be freed.
3. The host frees the buffer.

There is no positive or negative acknowledge of the result of a packet transmit. If the adapter fails to transmit a packet it may still return the buffer to be freed without informing the host that an error occurred.

In the case of large transmitted packets that require TCP segmenting by the adapter an acknowledgement for each large packet is required. The more common case of successful transmission requires a positive acknowledgement. This is done implicitly by the freeing of the transmit buffer, i.e. in the case where TCP segmentation occurs, the freeing of the buffer in the normal manner implies a successful transmission of the whole packet. If an error occurs then an error response is returned on the *Response Ring*.

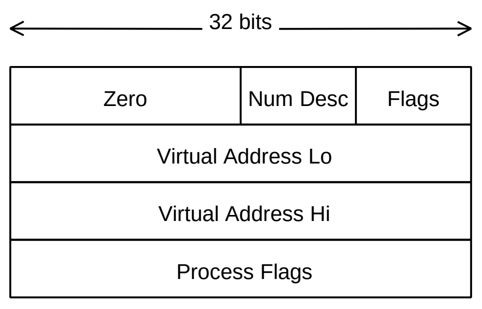
The same mechanism is used when transmitting encrypted packets. If the adapter successfully encrypts and transmits a packet it implicitly returns a positive acknowledge by freeing the buffer. If an error occurs it returns an error response on the *Response Ring*.

Packet Transmit Descriptors

The main purpose of the *Tx Ring* is to pass data packets to the adapter for transmission. The general case is transmitting non-encrypted, non-segmented packets and the format of the descriptors is tuned for this purpose. In this case a packet is described by a single *Data Frame Descriptor* followed by one or more *Fragment Descriptors*.

Tx Frame Descriptor

A *Tx Frame Descriptor* is the first descriptor in a *Frame Descriptor Block* and contains information pertinent to the complete frame.



Tx Frame Descriptor

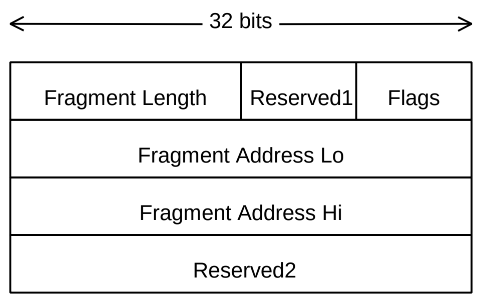
The descriptor has the following fields:

Flags [0:7]	[0:2] – *DescriptorType*. (Set to *001 – Tx Frame Descriptor)* 000 – Fragment Descriptor **001 – Tx Frame Descriptor** 010 – Command Frame Descriptor 011 – Option Descriptor 100 – Receive Descriptor 101 – Response Descriptor [3:7] – *Reserved*. Set to zero.
NumDesc [8:15]	The number of descriptors following this one in this *Frame Descriptor Block*. This includes *Option Descriptors* and *Fragment Descriptors*. It may be zero, one, or many.
Zero [16:32]	May be used for the Number of bytes in the complete packet. If zero then this field contains no valid data.
Virtual Address Low [0:31]	The least significant 32 bits of the virtual address in host memory of the packet. This value is passed back from the adapter to the host when the host can free the packet memory. The adapter does not use this field.
Virtual Address High [0:31]	The most significant 32 bits of the virtual address in host memory of the packet. This value is passed back from the adapter to the host when the host can free the packet memory. The adapter does not use this field.
Process Flags [0:31]	[0] – *Do Not Add CRC*. If 0 then the adapter adds a frame CRC else the CRC is already included in the packet. [1] – *IP Checksum*. If 1 then the adapter computes and inserts the IP checksum. [2] – *TCP Checksum*. If 1 then the adapter computes and inserts the TCP checksum. [3] – *TCP Segment*. If 1 then the adapter must perform TCP segmentation on the frame. [4] – *Insert VLAN*. If 1 then the adapter must insert the previously stored VLAN tag into the frame. [5] – *IPSec.*. If 1 then the adapter must perform some type of IPSec functionality on the frame. The encryption parameters are specified in *an Extension Frame Descriptor*. [6] – *Priority Valid*. Set if *Priority* field is valid [7] – *UDP Checksum*. If 1 then the adapter computes and inserts the UDP checksum. [8] – *Pad Frame*. If 1 then the pads the frame with zeros to the length specified in the *Pad Pkt* field. This option is only valid is the adapter is appending a MAC CRC. [9:11] – *Reserved*. Set to zero [12: 27] *VLAN and Priority Value*. Priority located at bits [20: 22].

[28:31] – *Reserved.* Used internally by Typhoon Software. Set to zero.

Fragment Descriptor

A number of *Fragment Descriptors* describe the multiple fragments that form the packet data. Each one points to a buffer in host memory and contains any other pertinent information on this single fragment.



Fragment Descriptor

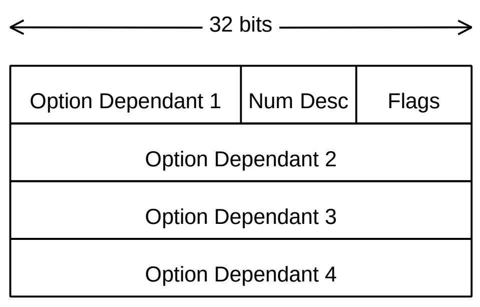
The descriptor has the following fields:

Flags [0:7]	[0:2] – *DescriptorType*. (Set to *000 – Fragment Descriptor)* **000 – Fragment Descriptor** 001 – Data Frame Descriptor 010 – Command Frame Descriptor 011 – Option Descriptor 100 – Receive Descriptor 101 – Response Descriptor [3:15] – *Reserved*. Set to zero.
Reserved1 [0:7]	Set to zero.
Fragment Length [0:15]	The number of bytes contained in the fragment.
Fragment Address Low [0:31]	The least significant 32 bits of the physical address in host memory of the fragment buffer.
Fragment Address High [0:31]	The most significant 32 bits of the physical address in host memory of the fragment buffer.
Reserved [0:31]	Set to zero.

Option Frame Descriptor

An *Option Frame Descriptor* contains information pertinent to a frame in addition to the information in the preceding Tx *Frame Descriptor*. It is used in special cases, such as when the transmitted frame requires IPSec processing or TCP segmentation.

One or more *Option Frame Descriptors* may be included in a *Frame Descriptor Block.* They always follow the Frame Descriptor and precede the *Fragment Descriptors*. It consists of the following fields:

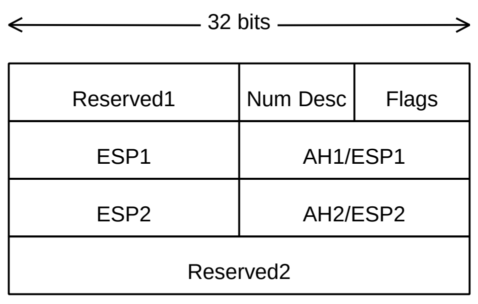


Option Descriptor

Flags [0:7]	[0:2] – *DescriptorType*. (Set to 011 – *Option Frame Descriptor)* 000 – Fragment Descriptor 001 – Data Frame Descriptor 010 – Command Frame Descriptor **011 – Option Descriptor** 100 – Receive Descriptor 101 – Response Descriptor [3] – *Unused*. Set to zero. [4:7] – *Type*. Defines the type of extension descriptor 0 – *IPSec* Type Option Descriptor 1 – *TCP Segment* Type Option Descriptor
NumDesc[0:7]	The number of descriptors describing this option (including this one. It cannot be zero. If multiple options are included in the same *Frame Descriptor Block* then this value only specifies the number for this particular option.
Option Dependant 1 [0:15]	
Option Dependant 2 [0:31]	
Option Dependant 3 [0:31]	
Option Dependant 4 [0:31]	

IPSec Option Frame Descriptor

The only defined type of Option Frame Descriptor at present is used when transmitting packets that are encrypted by the adapter. These are defined as follows:

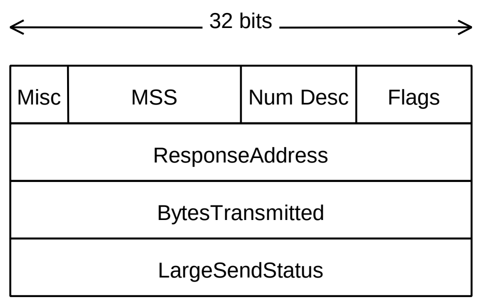


IPSec Option Descriptor

Flags [0:7]	[0:2] – *DescriptorType*. (Set to 011 – *Option Descriptor)* 000 – Fragment Descriptor 001 – Data Frame Descriptor 010 – Command Frame Descriptor **011 – Option Descriptor** 100 – Receive Descriptor 101 – Response Descriptor [3] – *Unused*. Set to zero. [4:7] – *OptionType*. Set to 0 (*IPSec Type Option Descriptor*). **0 – *IPSec* Type Option Descriptor** 1 – *TCP Segment* Type Option Descriptor
NumDesc[0:7]	The number of *Option Descriptors* (including this one) for this option. Equals 1 in this case.
IPsecFlags [0:15]	[0] – *UseIV*. 0 **-** The adapter will calculate a random IV and insert it into the IV field in the packet. 1 **- T**he DES algorithm will use the IV value contained in the packet. [1:15] – *Unused*. Set to zero.
AH1/ESP1 [0:15]	*Security Association Index for AH hdr 1 or ESP1 hdr (if no AH1).* If zero no hdr specified.
ESP1 [0:15]	*Security Association Index for ESP1 hdr.* If zero no hdr specified.
AH2/ESP2 [0:15]	*Security Association Index for AH hdr 2 or ESP2 hdr (if no AH2).* If zero no hdr specified.
ESP2 [0:15]	*Security Association Index for ESP2 hdr.* If zero no hdr specified.
Reserved2 [0:32]	*Reserved.* Set to zero.

TCP Segment Option Descriptor

This type of Extended Frame Descriptor is used when transmitting TCP packets that require segmenting by the adapter. These are defined as follows:



TCP Segmentation
Option Descriptor

Flags [0:7]	[0:1] – *DescriptorType.* (Set to 011 – *Option Frame Descriptor)* 000 – Fragment Descriptor 001 – Data Frame Descriptor 010 – Command Frame Descriptor **011 – Option Descriptor** 100 – Receive Descriptor 101 – Response Descriptor [3] – *Unused.* Set to zero. [4:7] – *OptionType.* Set to 1 (*Segment Type Option Descriptor*). 0 – IPSecType Option Descriptor **1 – *TCP Segment* Type Option Descriptor**
NumDesc[0:7]	The number of Descriptors (including this one) for this option. Equals 1 in this case.
MSS [0:11]	*MSS.* Maximum Segment Size.
Misc[0:3]	[0] – First Packet Descriptor [1] – Last Packet Descriptor See below for details. [2:3] – Unused… set to zero.
ResponseAddress[0:31]	*ResponseAddress.* Set to the PCI low physical address of the following two words.
BytesTransmitted [0:31]	***BytesTransmitted.* Set to Total Length of Large Send.
LargeSendStatus [0:31]	***LargeSendStatus.* Set to Send Complete without error.

<u>*Note:</u> The Driver should set <u>BytesTransmitted</u> and <u>LargeSendStatus</u> fields to the value of the total bytes of the original large send and passing status. The ResponseAddress field should be set to the PCI Lo Physical Address of the BytesTransmitted and LargeSendStatus fields. These fields are initialized so that the adapter is passed the total Length of the packet and the driver has preset a pass response so that Typhoon does not need to respond differently from a normal packet when no error is present. In the event of an error, using the ResponseAddress, Typhoon will modify these fields before the Ring Read Index has been advanced.

Misc:

For packets Smaller than 16 fragments: Set the First and Last bit to Zero. This indicates a complete Packets descriptor.

For packets larger than 16 fragments. Create the first packet descriptor with 16 fragments and set the First bit. Create a second packet descriptor adjacent in the ring to the first, set the Last Fragment bit.
A maximum of 32 fragments for TCP Segmentation is supported.

3.3.2 De-allocating Transmit Frames

The adapter processes transmit frames in either ring in order. When the adapter has copied the packet data into its internal memory it increments the Tx Read Index over the descriptors describing the packet. This informs the host that the adapter has finished with the descriptor space and the buffers associated with the packet can be freed.

Typical Transmit Data Flow

Transmit Packets passed down to the adapter are pre-formatted and contain place holders for all fields in the packet even if the host does not fill them in, with the following two exceptions:

1 The 4-byte VLAN tag if needed.
2 Large data frame for TCP Segmentation features: In this case, the adapter creates all the data segments, attached the template IP headers copied from the original large data frame, and allocate space for trailing bytes which include the IPSec explicit padding for DES and the hash data. The trailing bytes may contain layers of data if the packet data has nested IPSec encapsulation.

This section details the data flow for a transmit packet, it does not discuss more complex situations such as IP fragmentation and encryption.

1 The host writes descriptors to the *PCI Host Index* register which interrupts the adapter.

2 The adapter reads a block of the *Tx Ring* into internal memory.

3 The adapter parses a packet and sets up one or more DMAs to transfer the packet principal into internal memory.

4 The packet is classified based on the currently configured feature set. The types of information that can be determined are listed below. Note that only the minimum amount of processing for the current features is performed.
Types of results from packet classification:

Set MAC level flags to indicate unicast, multicast, broadcast, ethernet, 802.3, LLC, SNAP
Flags set to indicate IP, UDP,TCP
Index of IP header
Index of UDP/TCP header
Application Flags - DHCP

6 Application processing is performed on the packet such as broadcast limiting or DHCP inhibiting.

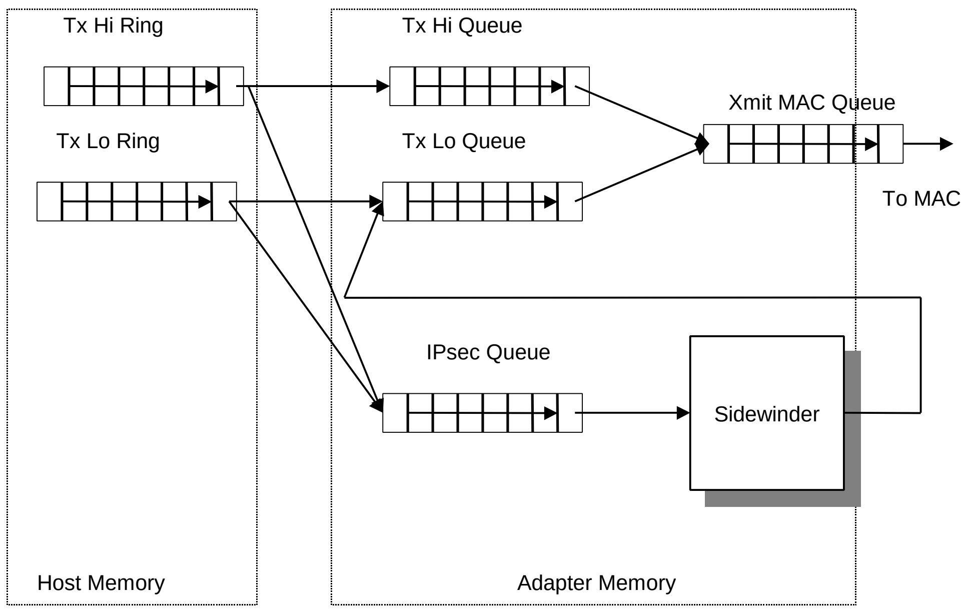
7 If space is available on the *Tx MAC Queue*, then the Xmit Dequeue Algorithm inspects the transmit priority queues and the remainder of the packet (if any) is copied from the host memory to adapter memory. A message is sent to the host to free the packet in host memory.

9 If the Packet requires encryption or hashing then the frame is first passed to the encryption engine.

8 If the Frame requires IP checksum calculation, then it is calculated at this time and the packet is updated.

7 The packet is added to the *Xmit MAC Queue.*

10 When the packet reaches the end of the queue, the MAC transmits the packet and updates its statistics counters. If any of the counters exceed programmed thresholds then an interrupt is generated to the host.



Broadcast Throttling

The host can set a broadcast threshold for transmit packets on the adapter. The broadcast threshold will be specified as a percentage of the ethernet line rate. The adapter will monitor the transmit packets and if the rate of broadcasts exceeds the threshold the packets will be discarded.

DHCP Inhibiting

The host can instruct the adapter to discard DHCP server packets. This feature prevents the host from acting as a DHCP server. The adapter will discard any received packets with a UDP port number set to DHCP server.

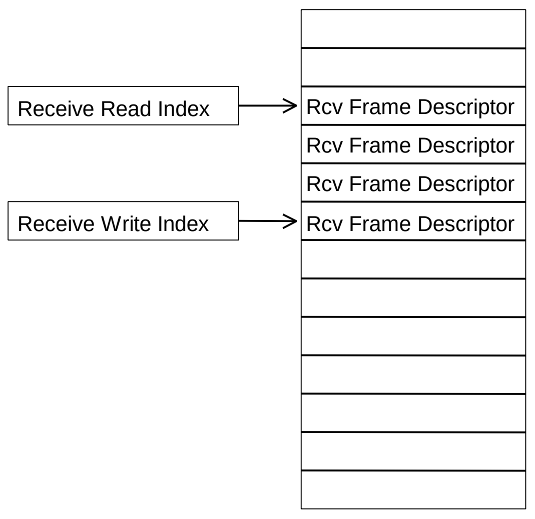
3.3.3 Adapter-to-Host (Rx) Interface

Receive packets are passed from the adapter to the host in full size packet buffers. Two shared rings in host memory are used to pass these buffers to and from the adapter. Each entry in the ring consists of a virtual and physical pointer to a packet buffer. The buffer may contain a maximum size ethernet packet as well as control information. Its size is discussed later in this section.. The virtual address is not used by the adapter, but is stored in the ring entry for use by the host.

The *Receive Ring* is used to pass full buffers from the adapter to the host. Two indexes stored in host memory control access to the buffer. The *Receive Write Index* and the *Receive Read Index* are used as follows.

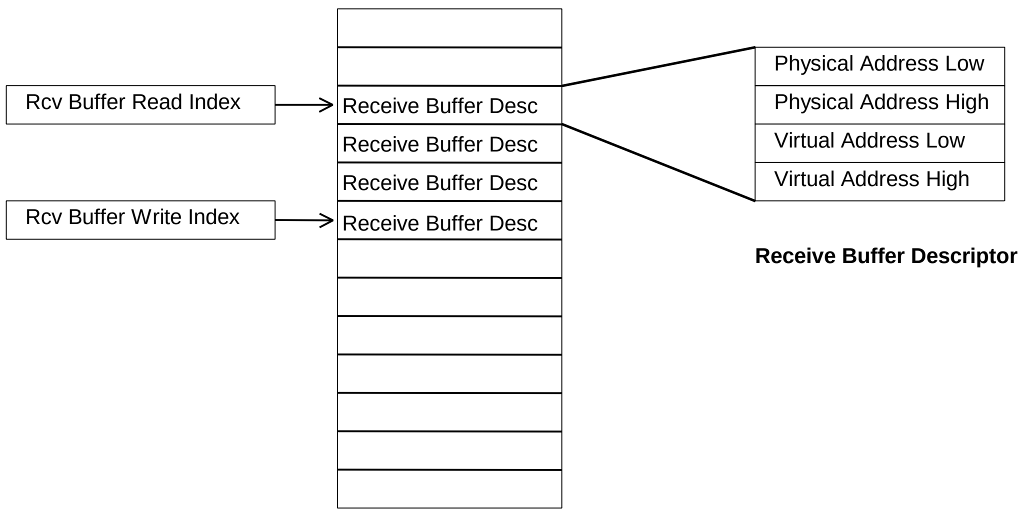
The 2 indexes are initialized to point to the start of an empty ring. The *Receive Write Index* points to the next location in the ring into which the adapter will write data, it is updated by the adapter. The *Receive Read Index* points to the next buffer address to be read by the host, it is updated by the host. The 2 indexes being equal indicate the ring is empty. The write index is never incremented to equal the read index for a full ring. This implies that the ring can not be completely filled and at least one entry must remain free to prevent the 2 indexes becoming equal.

In typical operation, the adapter will load entries into the ring as packets are received from the network and processed. The adapter will periodically generate a host interrupt to notify the host that packets are available in the ring. An intelligent algorithm will be used to minimize host interrupts, and it is expected that the adapter will not interrupt for each packet uploaded to the ring.



Receive Ring

The *Receive Buffer Ring* is used to pass free buffers from the host to the adapter. Its operation is identical to the *Receive Ring*, except that the host writes entries onto the ring as they become available, and the adapter reads them as necessary. The adapter will read and keep several free buffers locally, and will replenish them from this ring as needed.

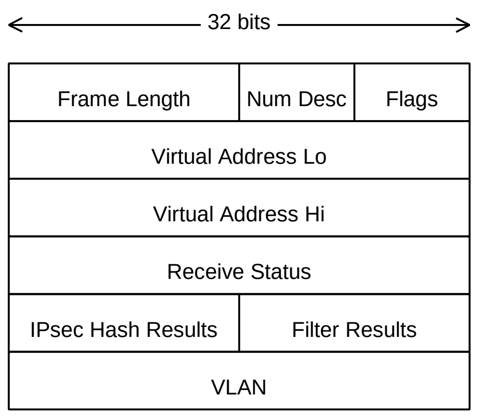


Receive Buffer Ring

The adapter keeps a local pool of empty *Receive Buffers* and will poll the host for new buffers when its local pool falls below a threshold. It is the host's responsibility to monitor the *Receive Buffer Ring* and ensure that there is sufficient buffers for the adapters use..

Receive Descriptor

A *Receive Frame Descriptor* contains information pertinent to the received frame. It consists of the following fields:



Receive Descriptor

Flags [0:7]

[0:2] – *DescriptorType*. (Set to 100 – *Receive Descriptor)*
- 000 – Fragment Descriptor
- 001 – Data Frame Descriptor
- 010 – Command Frame Descriptor
- 011 – Option Descriptor
- **100 – Receive Descriptor**
- 101 – Command Descriptor
- 110 – Response Descriptor

[3:4] – *Rcv Type*. Type of Receive Descriptor (set to 0).
- **0 – Receive Packet**
- 1 – Command Response

[5] – *Unused*. Set to zero.

[6] – *Error*. Set to 1 if the receive frame has any type of error.

[7] – *Unused*. Set to zero.

NumDesc [0:7]

Always set to zero, there are never any option descriptors in this ring. Any out-of-band information is passed in *the Receive Packet Structure* internal to the *Receive Packet Buffer*.

Frame Len [0:15]

Contains the length of the received frame in bytes.

Virtual Address Lo [0:31]

The least significant 32 bits of the virtual address in host memory of the *Receive Packet Buffer*.

Virtual Address Hi [0:31]

The most significant 32 bits of the virtual address in host memory of the *Receive Packet Buffer*.

Receive Status [0:31]

Error Code. If the *Error* flag is set in *Flags* then this field indicates the type of receive error.
- 0 – Adapter Internal Error
- 1 – FIFO underrun
- 2 – Bad SSD
- 3 – Runt
- 4 – CRC
- 5 – Oversize
- 6 – Alignment
- 7 – Dribble bit

Receive Status. If the *Error* flag is not set in the *Flags* then this field contains receive status information for this packet.

[0:1] – *Protocol*.
- 0 – unknown
- 1 – IP
- 2 – IPX
- 3 – reserved

[2] – *VLAN*. Set if VLAN tag removed (VLAN field in header is valid).

[3] – *IP fragment*, IPSec processing not performed

[4] – *IPsec*, set if received packet was decoded as an IPsec packet.

[5] – *IP Checksum Failed*

[6] – *TCP Checksum Failed*

	[7] – *UDP Checksum Failed* [8] – *IP Checksum Succeeded* [9] – *TCP Checksum Succeeded* [10] – *UDP Checksum Succeeded* [11:31] – *Unused*. Set to zero.
Filter Results [0:15]	Results of filtering on the received packet [0:14] – *Filter Mask*. One bit for each of the filters (bit 0 for filter 1, bit 14 for filter 15). Set when a received packet matches a filter. [15] – *Filtered*. Set if the packet has been filtered
IPsec Hash Results [0:15]	Indicates which hash values were compared successfully and unsuccessfully on the received IPsec packet [0] – *AH1 hash OK*, set if AH1 hash checked successfully. [1] – *ESP1 hash OK*, set if ESP1 hash checked successfully. [2] – *AH2 hash OK*, set if AH2 hash checked successfully. [3] – *ESP2 hash OK*, set if ESP2 hash checked successfully. [4] – *AH1 hash Failed*, set if AH1 hash checked unsuccessfully. [5] – *ESP1 hash Failed*, set if ESP1 hash checked unsuccessfully. [6] – *AH2 hash Failed*, set if AH2 hash checked unsuccessfully. [7] – *ESP2 hash Failed*, set if ESP2 hash checked unsuccessfully. [8] – *Unknown SA*, no SA matched on received packet. [9] – *ESP Format Error*, ESP length not a multiple of 8 bytes. [15:10] – *Unused*. Set to zero.
VLAN [0:31]	VLAN field removed from the received packet. Only valid if the *VLAN* bit in the *Receive Status* field is set.

3.3.4 Receive Priority Processing

There are 2 Rx Rings that the Typhoon can use to send received packets to the host, a high and a low priority ring. In normal operation all packets are sent to the host on the low priority ring.
If VLAN receive offloading is enabled and a packet is received with a VLAN tag, then the priority field in the VLAN tag is used to determine which ring is used. This is the only case at present where the high priority Rx Ring is used. The priority level that determines which ring the packet will use will be set to a default value by the Typhoon firmware and a host command will be implemented to allow the host to modify the value.

3.3.5 Receive Data Flow

1. A packet arrives on the wire and the MAC loads it into its local FIFO. The MAC DMA engine loads the packet into a ring buffer in adapter memory. The receive status and checksum are stored in the Receive Packet Ring at the head of the packet.
2. The principal of the packet is moved from the ring buffer into local ARM93C memory on the adapter.
3. The packet is classified based on the currently configured feature set. The types of information that can be determined are listed below. Note that only the minimum amount of processing for the current features is performed.

   Types of results from packet classification:

   Set MAC level flags to indicate unicast, multicast, broadcast, ethernet, 802.3, LLC, SNAP
   Flags set to indicate IP, UDP, TCP
   Index of IP header
   Index of UDP/TCP header
   Index of start of application datagram
   Application Flags - dRMON capture filter
4. If the frame requires decryption, the frame is passed to the external decryption engine.
5. For TCP/IP packets, the ARM93C processes the TCP and IP checksums, discarding the packet if the checksums are bad.
6. The ARM93C sets up a PCI DMA descriptor to move the packet from the internal receive ring to host memory. The host buffer address was previously downloaded from the *Receive Buffer Ring* in host memory.
7. When the DMA is complete, the ARM93C writes the buffer address entry to the *Receive Ring* in host memory. Note that this step may be deferred if it is necessary to process packets out of order. That is, the adapter can store packets in host memory, but the host is only aware of them in the order the adapter writes the buffer addresses into the *Receive Ring*.
8. Periodically, the adapter will interrupt the host if packets have been uploaded, which will cause the host to examine the *Receive Ring* and process the packets on it.
9. When packet buffers are passed back to the host driver by the stack, the host will put a pointer entry to each buffer on the *Receive Buffer Ring*, making them available to the adapter again.

3.4 Host-to-adapter Command Descriptions

The host can send commands to the adapter either in-band or out-of-band. The general case is for the host to send all commands out-of-band. Only those commands that are synchronized to the data stream and must be executed relative to the data are sent in-band.

In-band commands are sent as part of the data stream in the *Tx Ring*. They are executed by the adapter in the order they are received relative to the data packets. If a command is situated in the ring before a transmit packet then it is guaranteed to be executed before that packet is processed by the adapter.

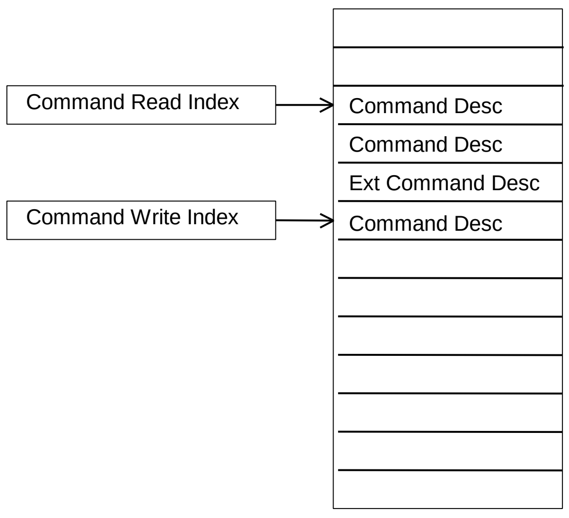
Out-of-band commands are sent to the adapter through the *Command Ring*. Their execution is asynchronous to the transmit and receive data paths. Responses to all commands are placed in the *Response Ring*.

Two rings are defined, the *Command Ring* and the *Response Ring*, which are used to send commands to the adapter and responses to the host.

All communication through the Command and Response Rings use register mode; polled mode is not supported.

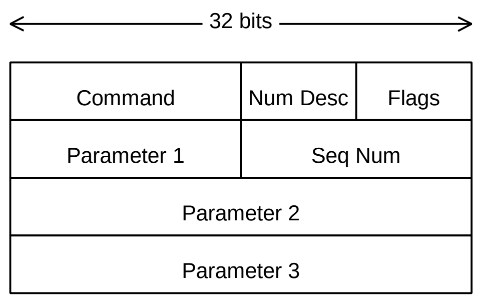
3.4.1 Command Ring

The *Command Ring* contains *Command Descriptors,* which contains command information from the host to the adapter. The majority of the commands are contained in a single *Command Descriptor*, which is described below. Commands that contain more information than will fit into *a Command Descriptor* are described by *one Command Descriptor* and a number of *following Extension Command Descriptors.* There is no set format for *Extension Command Descriptors*, each command that requires them defines the internal data structures.



Command Ring

The *Command Descriptor* has the following fields:



Command Descriptor

Flags[0:7]	[0:2] – *DescriptorType*. (Set to *101 – Command Descriptor)* 000 – Fragment Descriptor 001 – Data Frame Descriptor 010 – Command Frame Descriptor 011 – Option Descriptor 100 – Receive Descriptor **101 – Command Descriptor** 110 – Response Descriptor [3:5] – *Reserved*. Set to zero. [6] – *Respond*. If set, the adapter must respond to the command with a positive acknowledge packet. If clear, the adapter only responds if the command requires a response or an error occurs parsing the command. [7] – *Reserved.* Set to zero.
NumDesc [0:7]	The number of Extension Command Descriptors following this descriptor. The value may be 0.
Command [0:15]	Command ID
Seq Num [0:15]	Sequence number of this command.
Parameter1 [0:15]	Command's first parameter (16 bits)
Parameter2 [0:31]	Command's second parameter (32 bits)
Parameter3 [0:31]	Command's third parameter (32 bits)

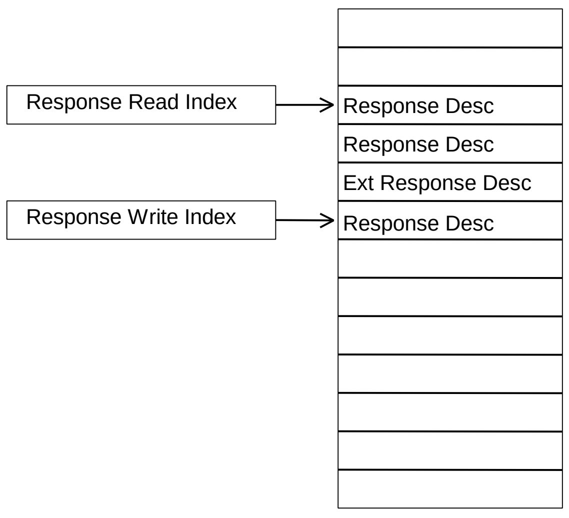
Note that the *Flags* field is the same as for the other types of transmit descriptors. This enables the commands to be easily inserted in to the data stream in the *Tx Descriptor Ring*.

Extension Command Descriptor

An *Extension Command Descriptor* contains information specific to the command specified in the *Command Descriptor*. It is a fixed length (16 bytes) the same size as a *Command Descriptor*. A variable number of *Extension Command Descriptors* (including 0) can follow a *Command Descriptor* with the last one always padded to its full length by zeros.

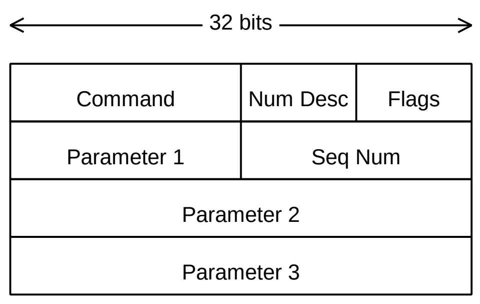
3.4.2 Response Ring

The *Response Ring* is used to pass command responses from the adapter to the host. It is also used to pass any other control information from the adapter to the host. Most commands do not require a response, only those that do use this ring. Both in-band and out-of-band commands return their responses on this ring.



Response Ring

Response Descriptor



Response Descriptor

The Response Descriptor has the following fields:

Flags[0:7]	[0:2] – *DescriptorType.* (Set to *110* – Response *Descriptor)* 000 – Fragment Descriptor 001 – Data Frame Descriptor 010 – Command Frame Descriptor 011 – Option Descriptor 100 – Receive Descriptor 101 – Command Descriptor **110 – Response Descriptor** [3:5] – *Reserved*, set to zero. [6] – *Error*. Set to 1 if the response is reporting any type of error. [7] – *Reserved*, set to zero.
NumDesc [0:7]	The number of Extension Command Descriptors following this descriptor. The value may be 0.
Command [0:15]	The type of command to which we are responding.
Seq Num [0:15]	Sequence number of the command that generated this response.
Parameter1 [0:15]	Response's first parameter (16 bits)
Parameter2 [0:31]	Response's second parameter (32 bits)
Parameter3 [0:31]	Response's third parameter (32 bits)

Note that the *Flags* field is the same as for the *Receive Descriptor*. This enables the response to be easily inserted in to the packet data stream in the *Receive Ring*. Although the design of the descriptors allows this, no responses are passed in the *Receive Ring* at present.

Extension Response Descriptor

An *Extension Response Descriptor* contains information specific to the command specified in the *Response Descriptor*. It is a fixed length (16 bytes) the same size as a *Response Descriptor*. A variable number of *Extension Response Descriptors* (including 0) can follow a *Response Descriptor* with the last one always padded to its full length by zeros.

3.4.3 Arm to Host Interrupts

The following section is taken from the Typhoon ASIC Specification and describes the ARM to Host interrupt register.

HostIntStatus Register

Synopsis	Provides specific interrupt information to the host.
Size	32 bits
Offset	0xE8

Side Effects **None.**

The *HostIntStatus* register holds the individual interrupt bits from their various sources, so the host can determine the cause of an interrupt. These bits are set by hardware events, are readable by the host, and are cleared by writing a “1” to the bits.

Reserved	[31:10]		Reserved.
HostSelfInt	[9]	R/W1C	Host self-interrupt, set as a side-effect of a host write to the *HostSelfInterrupt* register. This allows the driver on the host to generate an adapter interrupt to itself.
PciDmaDone[3]	[8]	R/W1C	Indicates that PCI DMA channel 3 completed a DMA transfer in which the HST_INT descriptor flag was set.
PciDmaDone[2]	[7]	R/W1C	Indicates that PCI DMA channel 2 completed a DMA transfer in which the HST_INT descriptor flag was set.
PciDmaDone[1]	[6]	R/W1C	Indicates that PCI DMA channel 1 completed a DMA transfer in which the HST_INT descriptor flag was set.
PciDmaDone[0]	[5]	R/W1C	Indicates that PCI DMA channel 0 completed a DMA transfer in which the HST_INT descriptor flag was set.
ArmComm[3:0]	[4:1]	R/W1C	General purpose 93C-to-Host communication interrupts. Set as a side-effect of writes to the *Arm2HostComm* registers.
HostInt	[0]	R	Set if any interrupt bit is set. This is the logical OR of the interrupt-causing bits after their masks. This bit isn’t explicitly cleared by the host; if all active interrupt sources are cleared, this bit will deassert.

In normal course of operation the adapter does not interrupt the host. Interrupts only occur if the adapter determines that is needs to immediately inform the host of a status change. For example, when receiving packets, the adapter copies the received data to free receive buffers and updates the *Rx Ring.* It assumes that the host will notice that it has receive data and empty the *Rx Ring.* If the adapter determines that the host is not emptying the ring it will interrupt the host.

The adapter issues a host interrupt by writing Arm-to-Host Register 3 (*ArmComm[3]* in *HostIntStatus* register).

During the boot process the adapter frequently writes its status to Arm-to-Host Register 0. This will cause host interrupts if *ArmComm[3]* is enabled in the *HostIntStatus* register.

3.5 Adapter Boot

The following sections contain information on the operation of the Typhoon embedded code boot sequence following the removal of RESET.

3.5.1 Overview

Upon release of RESET, the processor on the Typhoon adapter will perform the following boot sequence:

ROM Boot - Execute code from ROM and load an image from the Non-Volatile Device(NVDevice) into instruction memory (IRAM).

NVDevice Boot - Execute the code in IRAM that was loaded from NVDevice. Perform self test, PCI Configuration and copy the Run-Time Image (RTI) from NVDevice to IRAM/SRAM. The functionality of the NVDevice Boot code will be limited by NVDevice size. The detailed explanation of NVDevice configuration is in section 8.1.

Runtime Boot - Execute code loaded from Run-Time Image section of NVDevice. RTI will do normal communication process with the additional capabilities for upload of NetBoot code from NVDevice to host, NVDevice programming, download of new RTI to adapter memory and off-line diagnostic capability. At this point we are running our operational code.

Soft Reset – Allows Host or Run-Time code to soft-reset some or all of the adapter.

The adapter updates its PCI Status Register at significant points in the boot sequence. This allows the host to monitor the progress of the boot and to determine if any failures occur and at what point. At later stages of the boot process the host uses the status as part of a handshake procedure to determine when the adapter needs attention from the host.

3.5.2 ROM Boot Loader

The ROM on the Typhoon chip is programmed during chip tape out. To reduce the likelihood of a software bug forcing a new tape out, care will be taken to reduce the amount of code and complexity of code in the ROM that cannot be patched or fixed externally using the NVDevice. Therefore, the primary function of the ROM Boot will be to copy a block of instructions from the NVDevice into Instruction Memory and execute it.

The ROM Boot Loader is initiated by a reset and completes when execution is passed to the memory resident code from the NVDevice.

After release from reset, the processor will execute the instruction at address zero, which is mapped into the on-chip ROM. The ROM will contain code necessary to read instructions from NVDevice into instruction memory and begin executing.

It is assumed that the PCI does not need software intervention at this point in time and that the NVDevice contains a valid image. If the NVDevice is blank or contains an error, it must be programmed. See the Manufacturing Support- NVDevice Configuration section for details on layout of the NVDevice.

By default, the PciConfigDone register is set to disable PCI access preventing access by the host. During the ROM Boot process, the current status will be written to the PCI Arm2HostComm0 register. During a successful boot process, this status will be unavailable to the host from reading. In the event of a Boot Failure, the PciConfigDone bit will be cleared and the error status displayed in this register will be available to the host. For debug purposes, the boot status will be written to a shadow location **Attic_ShadowAddress**, which on the Typhoon was in external memory and could be viewed via a logic analyzer. On T2, most of the values are written to the Arm2HostComm1 register.

```
// Debug
#define Attic_ShadowAddress            0x40200000

// Macros
#define setShadowStatus(x)   *(volatile U32*)(Attic_ShadowAddress) = x;

#define setMyStatus(x)       *(volatile U32*)(ARM_Arm2HostComm1)   = x;\
                             *(volatile U32*)(Attic_ShadowAddress) = x;

#define setErrorStatus(x)    *(volatile U32*)(ARM_Arm2HostComm0) = x;
```

The ROM Boot performs the following actions:

Validate the NV Device headers.
Read the image from the NV Device and load it into memory.
Validate the CRC of the image as it is read from the NV Device.
Validate the CRC of the image that was written into memory.
Execute the image.

If failure occurs at any time, enable PCI, wait for commands from host.

The following is a list of ROM Boot Status codes, from **romboot\trboot.h**

```
#define DEFAULT_STATUS                  0
#define HELLO_WORLD                     1
#define TRBOOT_SECTION_HDR_INVALID      2
#define TRBOOT_TRBOOT_HDR_INVALID       3
#define TRBOOT_LENGTH_INVALID           4
#define TRBOOT_IMAGE_INVALID            5
#define TRBOOT_AHHHHHHH                 6
#define TRBOOT_DONE                     7
#define TRBOOT_MAGIC_ERROR              8
#define TRBOOT_MEMORY_INVALID           9
#define TRBOOT_BOOT_COMMAND_ERROR       10
```

3.5.3 Boot Command Language for Boot Failure

In the event of Boot Failure, the NV Device has not been programmed or has an invalid image checksum, the Typhoon Boot Loader will implement a Boot Command Language for memory access.

The Boot Command Language consist of 3 PCI registers from the Host used for Command, Address, Data and 2 PCI registers from Typhoon for Command Response, Data Response.

The sequence for commands to Typhoon should be as follows:

a) Write the Address
b) Write the Data
c) Write a Command

The order of Address and Data may be reverse but the must proceed the command. Writing the Command Register will cause an interrupt to the Typhoon 93C processor. This will initiate execution of the command. Therefore, the address and data must be valid before the command is written. For commands that do not need data or address, the register need not be initialize to a known value.

Following execution of the command, the 93C will clear the host interrupt and write a response data word, if needed, proceeded by an increment or change in the value of the Command Response register to indicate the response.

After receiving the Command Response, Typhoon will be ready for the next command.

Certain commands take a longer period of time then others, such as DataFlash buffer transfers. The command requested will not complete until the operation is done, including waiting for the DataFlash to be ready.

3.5.3.1 Host Access though registers in PCI space:

Register	Description
Host2ArmComm3	**Command** from Host (interrupts 93C for execution of command)
Host2ArmComm4	**Address** from Host
Host2ArmComm5	**Data** from Host
Arm2HostComm0	**Status** of Typhoon
Arm2HostComm1	**Sub Status** of finer details of status
Arm2HostComm2	**Data** Response from 93C
Arm2HostComm3	**Command** Response from 93C (increments for each command)

Host2ArmComm3 - This register is used to send Commands to the 93C. Writing this register will generate an interrupt to the 93C. This register should not be written again until the adapter has completed the command and writes the response to the Host.

Host2ArmComm4 - This register is used to send the Address used by some Commands to the 93C. This register does not generate an interrupt and should be valid before the Command register is written.

Host2ArmComm5 - This register is used to send the Data used by most Commands to the 93C. This register does not generate an interrupt and should be valid before the Command register is written.

Arm2HostComm0 – This is the status register for the adapter during the boot process. This register cannot be read until PciConfigDone has been set.

Arm2HostComm1 – This is the sub status register for the adapter during the boot process. This register cannot be read until PciConfigDone has been set. The sub status is a finer detailed description of the boot process.

Arm2HostComm2 - This register is used to return Data or Status words or bytes back to the Host from the completion of a Command from the Host. This value will be valid before the Command Completion interrupt and register to the Host is set. This register is only set as needed by the Command requested.

Arm2HostComm3 - This register is used to send the Command Completion of Response indication to the Host. The value of this register will be an incrementing count of the number of commands executed by the 93C, i.e. the value increments once for each command. The host may poll for a change in value or enable interrupts on this register.

3.5.3.2 Commands

WriteMemoryWord

Syntax: Address → Data → Command *Response:* Done
Semantic: Write the Data Word (32 Bit) to the memory location specified by Address. The address may be any memory or register address accessible by the 93C.

WriteDataFlashByte

Syntax: Address → Data → Command *Response:* Done
Semantic: Write the Data Byte (8 Bit) to the DataFlash buffer memory location specified by Address. The address should be in the format of a DataFlash memory address:
32 bits - XXXXXXXXRRRRRPPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)
B == Buffer address (from 0 to 264 bytes)

WriteDataFlashWord

Syntax: Address → Data → Command *Response:* Done
Semantic: Write the Data Word (32 Bit) to the DataFlash buffer memory location specified by Address. The address should be in the format of a DataFlash memory address:
32 bits - XXXXXXXXRRRRRPPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)
B == Buffer address (from 0 to 264 bytes)

StoreDataFlashPage

Syntax: Address → Command *Response:* Done
Semantic: Transfer the contents of the DataFlash buffer into the DataFlash page specified by Address. The address should be in the format of a DataFlash memory address:
32 bits - XXXXXXXXRRRRRPPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)

B == Buffer address (from 0 to 264 bytes)

LoadDataFlashPage

Syntax: Address → Command *Response:* Done
Semantic: Transfer the contents of the DataFlash page specified by Address into the DataFlash buffer. The address should be in the format of a DataFlash memory address:
32 bits - XXXXXXXXRRRRRPPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)
B == Buffer address (from 0 to 264 bytes)

Execute

Syntax: Address → Command *Response:* Done
Semantic: Begin executing code at the location specified by Address. The address may be any memory or register address accessible by the 93C. If you execute to an address that does not contain executable code, you get what you deserve.

ReadMemoryWord

Syntax: Address → Command *Response:* Data → Done
Semantic: Read a Word (32 Bit) from the memory location specified by Address and return the Data word with the response. The address may be any memory or register address accessible by the 93C.

ReadDataFlashByte

Syntax: Address → Command *Response:* Done
Semantic: Read a Byte (8 Bit) from the DataFlash buffer at the location specified by Address and return the Data byte with the response. The address should be in the format of a DataFlash memory address:
32 bits - XXXXXXXXRRRRRPPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)
B == Buffer address (from 0 to 264 bytes)

ReadDataFlashStatus

Syntax: Command *Response:* Data → Done
Semantic: Read the Status of the DataFlash and return the Status byte with the response.

Return status format: least significant 8 bits – RCDDDXXX

R == Ready / ~ Busy status	Ready == 1, Busy == 0
C == Compare status	Differ == 1, Match == 0
D == Device Size	010 == 2.1 Mbytes, 001 == 1 Mbytes
X == Undetermined	

DataFlashCompBuffer

Syntax: Address → Command *Response:* Data → Done
Semantic: Compare the contents of the DataFlash page specified by Address with the contents of the DataFlash buffer and return the Status byte with the response. The address should be in the format of a DataFlash memory address:

32 bits - XXXXXXXXRRRRRPPPPPPPPPPBBBBBBBBB
X == Don't care.
R == Reserved must be zero.
P == Page address (from 0 to 512 pages)
B == Buffer address (from 0 to 264 bytes)

3.5.3.3 Command Table

Command	**Num. (Dec)**	**Description**
WriteMemoryWord	1	Write Data Word at Memory Address
WriteDataFlashByte	2	Write Byte to DataFlash Buffer
WriteDataFlashWord	3	Write Word to DataFlash Buffer
StoreDataFlashPage	4	Store DataFlash Buffer to DataFlash Page
LoadDataFlashPage	5	Load DataFlash Page to DataFlash Buffer
Execute	6	Execute instructions at address
ReadMemoryWord	7	Read Data Word at Memory Address
ReadDataFlashByte	8	Read Data Byte from DataFlash Buffer
ReadDataFlashStatus	9	Read DataFlash Status
DataFlashCompBuffer	10	Compare DataFlash Page to DataFlash Buffer

3.5.3.4 Examples for Boot Command Language:

Write a word of data 0x12345678 into memory address 0x40200004:

Write Host2ArmComm4 (Address) = 0x40200004
Write Host2ArmComm5 (Data) = 0x12345678
Write Host2ArmComm3 (Command) = 0x00000001

Wait for Arm2HostComm3 interrupt or poll for change.

Read a word of data from memory address 0x40200010:

Write Host2ArmComm4 (Address) = 0x40200010

```
        Write Host2ArmComm3 (Command) = 0x00000007

        Wait for Arm2HostComm3 interrupt or poll for change.
        Read Arm2HostComm2 = data from address 0x40200010
```

Write a word of data 0x9ABCDEF0 into DataFlash page 0x1 buffer location 0x20

```
{       Write Host2ArmComm4 (Address) = 0x00000020
        Write Host2ArmComm5 (Data) = 0x9ABCDEF0
        Write Host2ArmComm3 (Command) = 0x00000003

        Wait for Arm2HostComm3 interrupt or poll for change.

}       Repeat steps above until all the desired data for this page has been written.

        Store the page:
        Write Host2ArmComm4 (Address) = 0x00000200
        Write Host2ArmComm3 (Command) = 0x00000004

        Wait for Arm2HostComm3 interrupt or poll for change.
```

Read a byte of data from DataFlash page 0x5 buffer location 0x7

```
        Store the page:
        Write Host2ArmComm4 (Address) = 0x00000A00
        Write Host2ArmComm3 (Command) = 0x00000005

        Wait for Arm2HostComm3 interrupt or poll for change.

{       Write Host2ArmComm4 (Address) = 0x00000007
        Write Host2ArmComm3 (Command) = 0x00000008

        Wait for Arm2HostComm3 interrupt or poll for change.
        Read Arm2HostComm2 = data byte from address 0x00000007

}       Repeat steps above until all the desired data for this page has been read.
```

3.5.4 NVDevice Boot

The NVDevice Boot is from the start of code loaded from the NVDevice through the execution of Sleep Image from the Sleep Image section of NVDevice.

The Boot Image (NVDevice Boot code) will run Power-On Self Test (POST) and when POST completes, it will indicate a done status in a general-purpose register. The POST results will be saved to a reserved location in SRAM and also in a general -purpose I/O register. The PCI configuration will be transferred to the PCI registers from the VAR Section of NVDevice, if the VAR checksum verified. If the VAR checksum cannot be verified the PCI registers will have default values.

If the POST succeeds, then the Sleep Image will be copied from the NVDevice to IRAM and SRAM as specified by the load address in section header located at the beginning of the sleep section. Once the Sleep Image is loaded, the execution is transferred to beginning of this code.

If the POST fails, PCIConfigDone will be set; and the adapter will allow NVDevice programming.

NVDevice Boot performs the following actions:

```
PCI Status Register  = STATUS_NVDEVICE_POST;

Perform Power On Self Tests;     /* See  Boot Level Self Test for details */
if ( tests fail )
{
        PCI Status Register  = STATUS_POST_FAIL;
        PCI Status Register 2 = details of failure;
        PCIConfigDone  = 1;

        Wait for Host;     /* wait for host to take necessary action
                              or program NVDevice and reset adapter */
}
else  /* all tests pass */
{
        PCI Status Register  = STATUS_NVDEVICE_HW_CONFIG;
        Configure the hardware (PCI, MAC, DMAs etc);
        PCI Status Register  = STATUS_NVDEVICE_LOADING_SLEEP_IMAGE;
        Copy Sleep Image from NVDevice to IRAM/SRAM;
        PCI Status Register  = STATUS_LOADING_SLEEP_IMAGE_COMPLETE;
        Jump to Sleep Image;
}
```

3.5.4.1 Boot Level Self Test

The Boot Image will indicate HELLO_WORLD status (0x1) to the Arm2HostComm1 register and then call POST functions. If a POST failure occurs then the Arm2HostComm0 register will indicate STATUS_POST_FAIL. Testing will not continue once a failure has been identified. Depending on the test failure, the other Arm2HostComm registers may be used to indicate further test failure information. If a test has failed, the PCIConfigDone register will be set and the code will enter the Boot Monitor. The Boot Command Language for Boot Failure is then supported.

POST will test internal register bits which have read/write capability and no adverse side-effects. Tables are setup to indicate which bits in which registers are testable. Incremental values are written and then verified. Locations are cleared after an access test has completed. If a register access fails the Arm2HostComm1 register will indicate POST_FAIL_REG_ACCESS. If the register is a crypto specific register then the Arm2HostComm1 register will indicate POST_FAIL_SW_REG_ACCESS. The Arm2HostComm2 register will indicate the address which was tested when the failure occurred.

The following registers and bits are tested:

Arm2HostComm0 [31:0]
Arm2HostComm1 [31:0]
Arm2HostComm2 [31:0]
Arm2HostComm3 [31:0]

StationAddress0[15:0]
StationAddress1[15:0]
StationAddress2[15:0]
StationMask0[15:0]
StationMask1[15:0]

StationMask2[15:0]
VlanMask[15:0]
VlanType[15:0]
DmaConfigReg1[1:0]
MacControl[9:6, 4:0]

SWInDMADescriptorRingBase[16:0]
SWInDMADescriptorRingEnd[16:0]
SWOutDMADescriptorRingBase[16:0]
SWOutDMADescriptorRingBase[16:0]
SWInterruptMask[7:0]
SWContextReadPtr[4:0]
SWContextWritePtr[4:0]
SWRandomizerResult[31:0]
SWExpoMultK[31:0]
SWNumExpoBits[9:0]
SWMultiplyPointer[23:0]

Internal instruction RAM is tested from the beginning of IRAM to the starting location of the Boot Image in the IRAM. Data RAM is tested from the beginning of DARAM to the starting location of the Boot Image stack. The external Attic RAM is then tested. Incremental values are written to the blocks of memory starting with 0xAAAAAAA. The block is verified and then written with incremental values starting with 0x55555555 and verified. The tested block is then initialized to zero. If the IRAM test fails, the Arm2HostComm1 register will indicate POST_FAIL_IRAM. If the DARAM test fails the Arm2HostComm1 register will indicate POST_FAIL_DARAM. If the Attic RAM test fails the Arm2HostComm1 register will indicate POST_FAIL_ATTIC. The Arm2HostComm2 register will indicate the specific address where the access test failed.

POST will then verify the checksum of the static section in the FLASH. If the static section does not verify then the Arm2HostComm1 register will indicate POST_FAIL_FLASH and the Arm2HostComm2 register will indicate STATIC_SECTION_ID. If the static section checksum is verified, then the PCI Device ID stored in this section is transferred to the PCI Device ID register. The variable section flash checksum is then verified. If the variable section does not verify then the Arm2HostComm1 register will indicate POST_FAIL_FLASH and the Arm2HostComm2 register will indicate VAR_SECTION_ID. If the variable section checksum is verified, then the OverrideNodeAddress stored there will be transferred to the station address registers in the MAC.

The external RAM which is located in the crypto chip is then verified. The RAM test failure is not reported if the PCI Device ID in the FLASH indicates that no crypto is installed. If the failure occurs and the PCI Device ID indicates that a crypto should have been populated, then the Arm2HostComm1 register is set to POST_FAIL_SW_MULT_RAM or POST_FAIL_SW_EXPO_RAM, depending upon the location of the RAM failure. The Arm2HostComm2 register will indicate the address where the failure occurred.

A functional test is then run on the crypto to verify device operation and to verify that the correct crypto type has been installed for the PCI Device ID programmed in the static section of the FLASH. A triple DES functional test is run first. If the test fails and the PCI Device ID (0x9903) indicates that a crypto with 3DES support should be populated, then the Arm2HostComm1 register will indicate POST_FAIL_SW_FUNC. If the 3DES test passes and the PCI Device ID does not indicate that a 3DES capable crypto should be installed, then the Arm2HostComm1 register will indicate POST_FAIL_DEVICE_CONFIG.

If POST succeeds the PCIConfigDone register is set if the RomInfo parameter in the verified variable section indicates that RomBoot is disabled. If RomBoot is enabled, then the PCIConfigDone is set by the Sleep Image which has the ROM boot support.

The following is the enumerated value used to indicate a POST failure in Arm2HostComm0:

STATUS_POST_FAIL	0x800E

The following enumerated values are used to indicate POST failures in Arm2HostComm1:

POST_SUCCESS	0x1000
POST_FAIL_FLASH	0x1001
POST_FAIL_ATTIC	0x1002
POST_FAIL_IRAM	0x1003
POST_FAIL_DARAM	0x1004
POST_FAIL_REG_ACCESS	0x1005
POST_FAIL_SW_REG_ACCESS	0x1006
POST_FAIL_SW_EXPO_RAM	0x1007
POST_FAIL_SW_MULT_RAM	0x1008
Reserved1	0x1009
Reserved2	0x100A
POST_FAIL_SW_FUNC	0x100B
POST_FAIL_DEVICE_CFG	0x100C

The Sleep Image is then transferred from the flash to the specified RAM locations. As the image is transferred a checksum is generated and verified. If the checksum is not verified the Arm2HostComm0 register will indicated STATUS_EEBOOT_FAIL (0x8005) and the Arm2HostComm1 register will indicate the source of the failure. The following are the enumerated values for the possible failures:

Reserved	0-1
EEBOOT_HDR_INVALID	2
EEBOOT_SLEEP_HDR_INVALID	3
EEBOOT_LENGTH_ZERO	4
EEBOOT_IMAGE_INVALID	5
Reserved	6-8
EEBOOT_MEMORY_INVALID	9
Reserved	10
EEBOOT_NO_SLEEP_IMAGE	11

3.5.4.2 Sleep Image Boot

The Sleep Image executes at a slower clock (5Mhz), and is responsible for setting up the hardware and software environments and PCIConfigDone register and then wait for one of the following requests from the host.

- Waiting for the *Boot Record* from the host. The *Boot Record* contains pointers to all shared memory data structures and needs to be read before any packet transfers can take place.
- Host accesses the Expansion ROM address. Sleep Image will turn the clock up to full speed, and upload the NetBoot code from the NVDevice to host memory.
- Host writes to Power Management Control Register to state D3, in which case the Sleep Image will begin sleep events. The only sleep event currently supported is Magic Packet.
- Download of Runtime Image using Host to Adapter command Interface.

Register usage in both parts of the sleep code is as follows:

Host to ARM Registers:

Register	**Sleep time Init**	**Sleep**
Register 0:	*PCI Host Command Register*	unused
Register 1:	*PCI Host Address Register Lo*	*PCI Tx Write Index*
Register 2:	*PCI Host Address Register Hi*	*PCI Cmd Write Index*
Register 3:	unused	unused
Register 4:	unused	unused
Register 5:	unused	unused
Register 6:	unused	unused
Register 7:	unused	unused

PCI Tx Write Index	- write index into the *Tx Descriptor Ring*
PCI Cmd Write Index	- write index into the *Command Descriptor Ring*

Arm to Host Registers

Register	**Sleep time Init**	**Sleep**
Register 0:	*PCI Status Register*	*PCI Status Register*
Register 1:	*PCI Status Register 2*	unused
Register 2:	unused	unused
Register 3:	unused	unused

Once the Boot Record is successfully downloaded, the sleep image will post the status as STATUS_RUNNING to the Host.

The Sleep time boot sequence is as follows:

PCI Status Register = STATUS_INIT;
Initialize h/w and s/w data structures;

PCI Status Register = STATUS_WAITING_FOR_HOST_REQUEST;

Sleep-time Init:

```
while ( Host Command Register !=
      (CMD_BOOT_RECORD |
        Host initiated memory reads from Expansion ROM address);
       /* wait for host to load the request */

If (Host Initiates to read Expansion Rom Address)
        {
        Turn on the clock to full speed;
        PCI Status Register = STATUS_UPLOAD_NETBOOT_CODE;
        Upload of NetBoot code from NVDevice;
        Goto waiting for Host to load the request;
        }

If (Host Command Register == CMD_BOOT_RECORD)
        {
        PCI Status Register = STATUS_READING_BOOT_RECORD;
        Initiate DMA to load Boot Record into internal memory (via snoop buffer);
```

```
    while ( DMA not complete )
            ;   /* wait for Boot Record load */
    Configure internal variables;
    PCI Status Register  = STATUS_RUNNING;
    }
```

Normal Sleep-time operation:

Adapter starts normal operation for Transmit and Receive;

Adapter will allow NVDevice programming;

Adapter will perform sleep events.

Adapter will download of Runtime Image;

```
If (Host Command Register == CMD_NVDEVICE_PROGRAM)
    {
    PCI Status Register  = STATUS_PROGRAMMING_NVDEVICE;
    Copy Data from Host to NVDevice using HostToAdapter Commands;
    Issue Reset for loading new data from NVDevice;
    }
```

3.5.4.3 Runtime Boot

The Runtime image is downloaded from the host using the Host to Adapter command interface. Once the RTI is successfully downloaded, the execution control is transferred to this code.

Note that the Diag image is a more or less a superset of the Runtime image, and is loaded the same way.

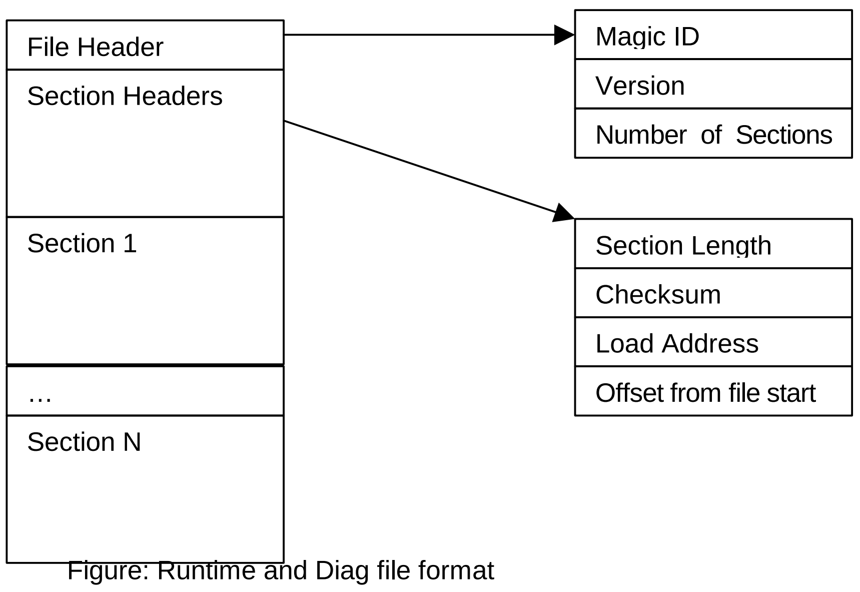


Figure: Runtime and Diag file format

The download file for the runtime image will consist of a file header followed by a variable number of section headers, followed by the sections themselves. Each section header contains a checksum for that section, the address in Typhoon where the section should be loaded, and the offset from the head of the file where the binary element of the section begins.

Zero-length sections are permitted. The load address of the first section in the file is the start address of the runtime image, so section order must be preserved when reading in the runtime file and passing it to Typhoon.

File Header

```
typedef struct {
        U8  magicID[8];                 /*  'TYHPOON\0'   */
        U32 version;                    /* TBD */
        U32 numSections;                /* number of section headers following this. */
} RUNTIME_FILE_HDR_S;
```

Section Header

```
typedef struct {
        U32 numBytes;                   /* number of bytes in this section */
        U16 checksum;                   /* IP checksum of data in section*/
        U16 unused;                     /* for alignment */
        U32 armStartAddr;               /* addr in arm memory to load this section */
        U32 fileOffset;                 /* offset from start of file to 1st byte of section*/
} RUNTIME_FILE_SECTION_S;
```

To pass the runtime image to Typhoon the steps are as follows: the driver must first ensure that the message WAITING_FOR_HOST_REQ is present in the Arm2Host0 register.

1) Monitor Arm2Host0 for status WAITING_FOR_HOST_REQUEST
2) Write HOST_CMD_RUNTIME_IMAGE to Host2Arm0
3) Wait for the interrupt from Arm2Host0 indicating a WAITING_FOR_SEGMENT status
4) Fill in the Host2Arm registers with the necessary information for this segment
   - Host2Arm1 = length, in bytes
   - Host2Arm2 = checksum of segment
   - Host2Arm3 = address in Internal Memory to write this data
   - Host2Arm4 = upper 32 bits of PCI address of buffer containing segment
   - Host2Arm5 = lower 32 bits of PCI address of buffer containing segment
5) Write HOST_CMD_SEGMENT_AVAILABLE to Host2Arm0
6) Repeat steps 3 through 5 until the entire image is downloaded – STATUS_SEGMENT_FAIL indicates that the download process must be aborted and the adapter reset.
7) Write HOST_CMD_DOWNLOAD_COMPLETE into Host2
8) Start ARM running at New address.
9) Monitor Arm2Host0 for status WAITING_FOR_BOOT
10) Proceed with normal runtime boot by downloading a boot record.

The values above are defined as:

STATUS_WAITING_FOR_BOOT	0x0007
STATUS_WAITING_FOR_HOST_REQUEST	0x000D
STATUS_WAITING_FOR_SEGMENT	0x0010
STATUS_SEGMENT_FAIL	0x8010
HOST_CMD_RUNTIME_IMAGE	0x00FD
HOST_CMD_SEGMENT_AVAILABLE	0x00FC
HOST_CMD_DOWNLOAD_COMPLETE	0x00FB

3.5.4.4 Boot Record

The *Boot Record* data structure is copied from the host to the adapter prior to any packets being transmitted or received. It contains pointer to all of the static data structures in host memory. The download of the *Boot Record* implies that the rings are empty and the read and write pointers for each ring are assumed to be pointing to the start of the rings.

Note that all pointers are 64 bits long. In 32 bit PCI address mode the high order 32 bits are always zero. It is also assumed that the high order 32 bits for the start and end pointers of each ring are the same (i.e. each ring does not cross a 32 bit addressing boundary).

***Boot Record* structure**

32bit value:	HostRingPtr;	/* points to *Host Ring Pointer* data structure */
32bit value:	VarsHi;	
32bit value:	TxLoPriStart;	/* Tx low priority descriptor ring */
32bit value:	TxLoPriStartHi;	
32bit value:	TxLoPriSize;	
32bit value:	TxHiPriStart;	/* Tx high priority descriptor ring */
32bit value:	TxHiPriStartHi;	
32bit value:	TxHiPriSize;	
32bit value:	RxLoPriStart;	/* Rx low priority ring */
32bit value:	RxLoPriStartHi;	
32bit value:	RxLoPriSize;	
32bit value:	RxBufferStart;	/* Rx buffer ring */
32bit value:	RxBufferStartHi;	
32bit value:	RxBufferSize;	
32bit value:	CtrlStart;	/* Command ring */
32bit value:	CtrlStartHi;	
32bit value:	CtrlSize;	
32bit value:	RespStart;	/* Command Response ring */
32bit value:	RespStartHi;	
32bit value:	RespSize;	
32bit value:	ZeroWordUL;	/* Word at this location is zero, for dma Burp */
32bit value:	ZeroWordHiUL;	
32bit value:	RxHiPriStart;	/* Rx high priority ring */
32bit value:	RxHiPriStartHi;	
32bit value:	RxHiPriSize;	

3.5.4.5 Host Ring Pointer Structure

The *Host Ring Pointer* data structure is resident in host memory and is shared between the host and the adapter. It contains the read and write pointers into the 7 shared rings. The location of this data structure is defined by the host during initialization and a pointer to it is passed to the adapter in the *Boot Record*. The *Host Ring Pointer* data structure is shown below. Note that the *TxDescWriteIndex* and the *CmdWriteIndex* are passed from the host to the adapter in registers and are therefore not stored in the *Host Ring Pointer* structure.

Host Ring Pointer **structure**

32bit value:	RxHiReadIndxUL;	/* Host -> Typhoon */
32bit value:	RxLoReadIndxUL;	/* Host -> Typhoon */
32bit value:	RxBuffWriteIndxUL;	/* Host -> Typhoon */
32bit value:	RespReadIndxUL;	/* Host -> Typhoon */
32bit value:	TxLoDescReadIndxUL;	/* Typhoon -> Host */
32bit value:	TxHiDescReadIndxUL;	/* Typhoon -> Host */
32bit value:	RxLoWriteIndxUL;	/* Typhoon -> Host */
32bit value:	RxBuffReadIndxUL;	/* Typhoon -> Host */
32bit value:	CmndReadIndxUL;	/* Typhoon -> Host */
32bit value:	RespWriteIndxUL;	/* Typhoon -> Host */
32bit value:	RxHiWriteIndxUL;	/* Typhoon -> Host */

3.5.4.6 Boot with BIOS Extension ROM (NetBoot)

Once the ARM initializes the PCI the host will either boot locally and execute the adapter's driver, or in the case of a network boot, it will read the extended BIOS (NetBoot code) from the adapters NVDevice. If the ARM determines that the host is reading the Extended BIOS, it must read the NetBoot code from NVDevice and return it to the host. Once the host has copied the NetBoot to its local memory, it executes it. The ARM runtime image will be running at the time of NetBoot Code execution from the Host.

The Host will request a NetBoot if it has a valid NetBoot Image in NVDevice and is configured for network boot. Reset proceeds as normal through the ROM boot stage and starts executing the NVDevice boot image. This code checks for the network boot option set and that there is a valid image in NVDevice. If this is so, it sets a bit in the PCI configuration register to advertise the feature to the host and sets the configuration registers to allow the host to map the adapter's ROM into its address space.

The NetBoot code must contains basic network transmit and receive functions. These are needed to allow the network boot procedure to load operating system and driver from the network.

As part of the system boot the host checks for a network boot ROM presence on the adapters and if so attempts to read it into its own memory. The Typhoon advertises a virtual ROM address space to the host. When the host reads these locations the Typhoon hardware interrupts the ARM and presents the address of the host read to the ARM in a register. The ARM then reads the 32 bit value from NetBoot section of NVDevice and writes it into a data register which is returned to the host as the result of its read. In this manner the adapter fakes a ROM. The host should see no difference in functionality except for a severe decrease in performance.

The reading of the ROM should take no longer than 2 seconds.

The space allocated for the network boot code should be at least 64k bytes and preferably 128k bytes.

3.6 Adapter Sleep

3.6.1 Putting Typhoon to sleep

“Putting the adapter to sleep” means executing the transition between state D0 (fully-on) and any other state except D3-Disabled. To put the adapter into sleep-mode, meaning executing sleep/wake-up events instead of the normal flow of data, the following steps must be followed.

1. The driver issues TxDisable and RxDisable commands to halt the data flow through the adapter.
2. The driver issues a Halt command and waits for typhoon to post a STATUS_HALTED. See Adapter Reset for details.
3. The driver issues GlobalReset to cause the sleep image to be reloaded from the NV device. This has the side effect of releasing all buffers held by Typhoon.
4. The driver polls the card looking for the STATUS_WAITING_FOR_HOST_REQUEST command.
5. The driver passes the boot record to the Typhoon. (see section 5.5.4.1 for details)
6. Once the typhoon has posted STATUS_RUNNING, the driver may issue any other commands desired:
    - StationAddressWrite if a MAC address other than the factory default is required.
    - RxFilterWrite to configure the receive hash filter if necessary.
    - AddWakeupFrame commands if desired, one per wake-up frame to be added.
    - Add SleepFrame commands if desired, one per sleep frame.
    - EnableSleepEvents with accompanying parameters for each event enabled.
    - EnableWakeupEvents with the desired events enabled.
7. The host writes the PCI PowerMgmtCtrl register to place the adapter in a lower power state. If the driver executes this action it must be done through the WritePCIConfigReg command. Alternatively, the driver may issue the GoToSleep command, telling Typhoon to begin sleep and wake events without waiting for the PCI power state to change.

There are two ways to wake the adapter. If it was put into a low power state via the PowerMgmtCtrl register, it may be woken the same way. If it was put to sleep manually, via the GoToSleep command, it must be woken by writing HOST_CMD_WAKEUP into Host2Arm register 0. Regardless of which wakeup method used, the driver must reload the runtime image into the Typhoon when changing from sleep mode to runtime mode, as the runtime image is not stored in the NV device.

3.6.2 Wakeup Events

3.6.2.1 Network Link State

The following Link States will be supported:

- Any change to the current network link state.
- The transition from link down to link up.
- The transition from link up to link down.

When any of these states are detected, the Typhoon can generate a wakeup signal to the host system.

3.6.2.2 Network Wakeup Frame

Wakeup frames for Typhoon are loosely modeled on the wakeup frame format used in Windows NT5.0 to minimize the amount of processing required in the driver. The driver will pass to the firmware a mask of the bytes in the incoming frame to be examined and list of the values to compare those bytes to. The mask may be no greater than 190 bytes; the buffer may be no greater than 1518 bytes.

Example:

> Mask: 00 00 00 C0 00 00 00…
> Buffer: 08 06
>
> This buffer/mask combination would match on any ARP packet in standard Ethernet format – that is, any ethernet packet with an 0x806 in the type field.

No Sidewinder decryption will be performed while in sleep mode. If an incoming wakeup packet will be encrypted, it is the driver's responsibility to pass down a wakeup frame that matches on the encrypted packet, not the unencrypted packet.

3.6.2.3 AMD Magic Packet™

A Magic Packet is defined as a packet that contains the data pattern six 0xFF followed by sixteen repetitions of our MAC address. This pattern may appear anywhere within a valid packet.

For example, if our MAC address is 00A086010203 then a Magic Packet would contain the data:

FF FF FF FF FF FF 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03 00 A0 86 01 02 03

3.6.3 Sleep Events

3.6.3.1 MAC keepalives

MAC keepalives require the IP address associated with the Typhoon. Typhoon will broadcast a MAC Address Refresh Frame periodically to keep the local hub cache current. This frame is based on the Ethernet Configuration Testing Protocol described in Chapter 8 of "The Ethernet Version 2.0" November 1982.

The frame is a reply frame with the source and destination addresses set to the host's address.

FIELD	SIZE	VALUE
Destination Address	6 bytes	Host's address
Source Address	6 bytes	Host's address
Type Field	2 bytes	0x9000
Skip Count	2 bytes	0
Function	2 bytes	1 - Reply
Receipt Number	2 bytes	Any 16 bit value
Data	40 bytes	Loopback data (anything)

This makes a minimum size packet.

3.6.3.2 ARP replies

ARP replies also require the IP address associated with the Typhoon. Typhoon will check incoming packets for an ARP request matching our IP address. When it finds one, it will form and transmit an ARP reply. See RFC 826, Ethernet Address Resolution Protocol for details on detecting and responding to ARP requests.

3.6.3.3 Ping replies

Ping replies require the IP address associated with the Typhoon. Typhoon will filter incoming packets for an ICMP Echo Request matching this IP address. When it finds one, it will form and transmit an ICMP Echo Reply packet. See RFC 792, Internet Control Message Protocol, for details on detecting and responding to ICMP Echo Requests.

3.6.3.4 DHCP Lease Renewal

DHCP lease renewal requires the IP address associated with the Typhoon and the Client Identifier that the DHCP client on the host used to secure the original lease. Typhoon will maintain the IP address lease by sending DHCPREQUEST messages to the local server. See Sleep Events Implementation Appendix for details.

3.6.3.5 Master Browser Host Announcements

Host Announcements require the Windows Host Name, Workgroup/Domain Name and IP address associated with the Typhoon. The Typhoon will, at 12 minute intervals, broadcast a Host Announcement packet to the Master Browser. See Sleep Events Implementation Appendix for details on forming a host announcement

3.6.3.6 Sleep Timer (alarm clock)

The Sleep Timer requires the exact time from now, in milliseconds that the driver wishes the system to be awakened.Typhoon will store this information and wake the system at the requested time. Note that the 32 bit number used to pass the number of milliseconds imposes a 49 day limit on the timer.

3.6.3.7 Novell NCP Watchdogs

Novell Watchdogs do not require any information from the driver. The Typhoon will filter for a watchdog packet directed to its MAC address. When it finds one it will formulate a response based on the incoming packet. See Sleep Events Implementation Appendix for details on detecting and responding to a watchdog packet.

3.6.3.8 Sleep Frames

Typhoon provides the ability for the operating system to send packets out even when asleep. These packets must be loaded into the Typhoon via the AddSleepFrame command when the system is being put to sleep. The packets must be physically contiguous, fully formed, valid Ethernet packets. Typhoon will not do TCP segmentation, IP checksum, IPSEC or any other processing on these Sleep Frames.

3.7 Adapter Reset

When resetting Typhoon using the PCI SoftReset register, you must use the following procedure to prevent problems with the DMA causing failure to reboot.

If you have not downloaded a boot record, simply write the PCI SoftReset register to reset the adapter. If you have downloaded a boot record, you must issue a CMD_halt command through the command ring and wait for typhoon to post a status of STATUS_HALTED. This halt command will cause the typhoon software to stop all activity and wait for the DMA's to become idle. Once the DMA's are idle it will post a STATUS_HALTED and it is safe to issue a PCI SoftReset. If the DMA's fail to become idle, Typhoon will remain posting a STATUS_HALTING and fail to post STATUS_HALTED, the driver may timeout and issue a PCI SoftReset.

Scope:

This document addresses the Anti-Starvation algorithm and De-Queue method contained within txmac.c for Typhoon. The queues referred to are the three queues (High, Low and IpSec) that head into the single Mac DMA ring.

Original De-Queue Algorithm:

The original Typhoon 1.0 De-Queue algorithm for the Txmac priority queues does not implement any type of anti-starvation algorithm. The implementation works as follows:

`TxmacQueueStatusUL` is a variable that contains a bit for each of the priority queues. This variable is updated by the macros TXMAC_QueueEntry(x) and TXMAC_QueueEmpty(x).

When an entry is added to the priority queue a bit for that queue is set. When the queue goes empty the bit is cleared.

For the De-Queue process, if there is space on the Mac DMA ring (less than 16 entries available) a priority is selected for the next packet to be De-Queued. This priority is selected by a simple table lookup as follows:

```
priorityUL = TxmacDeQueueTable[TxmacQueueStatusUL];
```

The table contains 8 elements favoring High, IpSec and then Low priority:

```
/********************** High Priority ... H I L ****************************/
  TxmacDeQueueTable[TXMAC_EMPTY_QUEUES          ] = TXMAC_HIGH_PRI;
  TxmacDeQueueTable[TXMAC_IPSEC_QUEUEING        ] = TXMAC_IPSEC_PRI;
  TxmacDeQueueTable[TXMAC_HIGH_QUEUEING         ] = TXMAC_HIGH_PRI;
  TxmacDeQueueTable[TXMAC_IPSEC_HIGH_QUEUEING   ] = TXMAC_HIGH_PRI;
  TxmacDeQueueTable[TXMAC_LOW_QUEUEING          ] = TXMAC_LOW_PRI;
  TxmacDeQueueTable[TXMAC_IPSEC_LOW_QUEUEING    ] = TXMAC_IPSEC_PRI;
  TxmacDeQueueTable[TXMAC_HIGH_LOW_QUEUEING     ] = TXMAC_HIGH_PRI;
  TxmacDeQueueTable[TXMAC_IPSEC_HIGH_LOW_QUEUEING] = TXMAC_HIGH_PRI;
```

The limitation of this method is that when the higher priority queues are being filled at a rate that is higher than the de-queue operation, the lower priority queues will be starved indefinitely.

3.8 Anti-Starvation De-Queue Algorithm:

The new Anti-Starvation De-Queue Algorithm implemented for Typhoon 1.1 works around the same priority table lookup used in Typhoon 1.0. This table has been increased to 8 de-queue options. The list below shows a line for each table with the order of queue favoritism:

```
  TxmacDeQueueTable[Table 0][…] = Low, High, IpSec;
  TxmacDeQueueTable[Table 1][…] = High, IpSec, Low;
  TxmacDeQueueTable[Table 2][…] = High, IpSec, Low;
  TxmacDeQueueTable[Table 3][…] = IpSec, High, Low;
  TxmacDeQueueTable[Table 4][…] = High, IpSec, Low;
  TxmacDeQueueTable[Table 5][…] = High, IpSec, Low;
  TxmacDeQueueTable[Table 6][…] = IpSec, High, Low;
  TxmacDeQueueTable[Table 7][…] = High, IpSec, Low;
```

A new variable has been added, TxmacAntiStarvationIndexUL, to index into this table array. For each De-Queue operation, the table index is incremented to favor the next chosen priority scheme in a round robin fashion. The De-Queue algorithm now look like:

```
     /* select a priority to dequeue */
     priorityUL =
TxmacDeQueueTable[TxmacAntiStarvationIndexUL++][TxmacQueueStatusUL];
     /* limit test the table index */
     if (TxmacAntiStarvationIndexUL < TXMAC_ANTI_STARVATION_SIZE)
       {
       TxmacAntiStarvationIndexUL = 0;
       }
```

The new anti-starvation algorithm provides the following approximate ratios assuming all priorities are full:

High Priority to IpSec Priority = 2:1

IpSec must still be favored closely to High priority since the IpSec queue could contain High priority IpSec packets, yet since this ring could also contain Lo priority packets a 1:1 ratio is not desired.

High Priority to Low Priority = 7:1

Comments by Anand Rajagopalan:

This is a simple algorithm that provides very little overhead for the queue decision process. Only an index wrap condition was added. Each table contain 8 bytes of data.

If desired, more tables could be added to increase the resolution of priority selections for fairness. Or a selective starvation index based on packets or bytes sent.

I think it is a neat little algorithm and will work fine. I found a couple of things that needed clarification: the Hi Priority to Ipsec Priority ratio - I think it is 3:1 as described by the tables. I'm including the case for Table 0, with no Low priority frames, and only High and Ipsec frames. So, if you actually want it to be 2:1 you will need to put Ipsec ahead of High in one of the tables - maybe table 0.

Its easy to change the ratios of any of combinations by changing the table entries. As long as we accept the algorithm I think we should have Chris merge the code and we can play with the relative priorities as we wish.

About the final comment in Chris' paper - He mentions a selective starvation index based on packets/bytes sent. I believe the index in it's present form is based on packets. i.e., the ratios indicate the number of High packets to Low packets. Or am I mistaken?

You are correct, the table index is incremented based on packets being processed.

Also, to make it byte based, we'll need to keep track of bytes sent out on each stream and do some comparisons to set the value of TxmacAntiStarvationIndeUL - right ?

Yes, to make it byte based requires a lot more overhead. We would need to keep track of the bytes transmitted on each queue and then switch the table index based on the relative byte transmitted over a set time period. If we decide we need this much accuracy then I think we should look at other algorithms that fit better to the task. This is getting more into queuing fairness algorithms than anti-starvation. Chris's algorithm solves the simple anti-starvation problem with very low overhead.

Mick

3.9 Encryption Enable Logic

For legal reasons, it was determined that shipping the boards as unencrypted NICs, then allowing the customer to enable them at there site was the best solution. The users would enter any required name/location/business information that was legal required into a WEB based interface, and then after validating that customer the Enable keys would be downloaded to his machine. Note that this also presents us with the opportunity to charge the customer for this feature.

The exact anti-logic, and procedures required to implement what is in the above paragraph is beyond this document and is described elsewhere. The only thing that we are concerned with here is the actual enable Key, and how it is used.

The original proposal for the Enablement feature had unique magic values held within ASIC registers that had to be matched by the keys in order for the particular ASIC to be enabled. As of June 2000, with the reduction of restrictions by the US Government on exporting encrypted products, this “unique” feature has been relaxed. The same key can be used for all cards, but the user still must be verified in order to receive it.

The actual mechanism is based upon the fact that the T2 chips support hardware to enable or disable the IPSec hardware offload features of the Mongoose built into there ASICs.

The Enable Key returned via the WEB process is simply an encoded file that, when downloaded to the firmware, provides information on how to write to the hardware mentioned above. Within the hardware is a register that the firmware can read to determine what IPSec features have been enabled, and this can be returned on up to the OS Device drivers when requested.

As this Key is simply a file, it can be, if desired, installed by the OEM on hard drive of the machine as it is shipped from their manufacturing site. This has no effect on how it is actually used.

From the customer’s point of view, the entire process looks like this:

1. Customer Logs into Website and provides User/Location information as is required by law.
2. Servers decide if customer can receive Enable Key.
3. Key is passed down to customer hard disk within an install application.
4. User runs the installer, which places enable key file in correct location
5. Device driver is restarted.
6. When the driver initializes, it discovers the Enable Key file on the hard disk, and downloads it to the Firmware.
7. The Firmware verifies the Key for version/size/checksum, and patches portions of it’s software with the data contained within key.
8. The firmware then reinitializes the part of itself that handles the hardware IPSec enable features, thus enabling the features of the ASIC.
9. When the firmware returns, the OS Device Driver can inquire the capabilities of the hardware, at which time it will see enabled IPSec features.
10. The OS Driver can then report these features to the OS as needed.

Note that once the enable key file resides on the machine’s hard disk, that the sequence starts at step #6 above. As long as the file remains on the machine, the IPSec features of the T2 card will be avaliable.

See the function **GetIPSecEnable** and **SnipitDownload** for more details. You can also refer to the section on **Encryption Enabling Hardware** in the T2 ASIC specification. Additional documentation on the Web Enablement and Installation process is also available.

4. Command Descriptions

4.1 Host-to-adapter Command Descriptions

This section lists the commands that the host sends to the adapter and the response (if any) generated by the adapter. All the parameters to the commands are described and whether the commands can be issued in-band, out-of-band, or both.

Commands typically perform read or write operations on the adapter. In general read operations result in the adapter generating a response. Write operations, if successful, do not generate a response. Any command that fails generates an error response. The format of this is shown below.

AcknowledgeResponse (out-of-band and in-band)

This response is returned if the command requests a positive acknowledge. It is not returned if the command is defined to reply with a response. If any errors occur in parsing the command then the Error Response is returned instead of this one.

Response:

Flags	U8	*0x86* (Response Descriptor | No Error | Valid)
NumDesc	U8	0 – no following Extension Descriptors
Command	U16	*Command*
Seq Num	U16	Command's sequence number.
Parameter1:	U16	*Error Code* 0 – no error

Error Response (out-of-band only and in-band)

This response is returned if any errors are detected in processing a command.

Response:

Flags	U8	*0xC6* (Response Descriptor | Error | Valid)
NumDesc	U8	0 – no following Extension Eescriptors
Command	U16	*Command*
Seq Num	U16	Command's sequence number.
Parameter1:	U16	*Error Code* 0 – no error 1 – unknown command 2 – command cannot be executed at this time 3 – command format invalid 4 – internal error processing command
Parameter2:	U32	Command specific error information.
Parameter3:	U32	Command specific error information.

GlobalReset (out-of-band only)

This command is not currently supported. See SoftwareReset command.

Command:	U16	**GlobalReset**
Parameters:	none	
Response:	none	

TxEnable (out-of-band only)

Enable the MAC transmitter. The default state is **disabled**.

Command:	U16	**TxEnable**
Parameters:	none	
Response:	none	

TxDisable (out-of-band only)

Disable the MAC transmitter. All frames on the MAC transmit queue will be transmitted before the transmitter is disabled. The default state is **disabled**.

Command:	U16	**TxDisable**
Parameters:	none	
Response:	none	

RxEnable (out-of-band only)

Enable the MAC receiver. The default state is **disabled**.

Command:	U16	**RxEnable**
Parameters:	none	
Response:	none	

RxDisable (out-of-band only)

Disable the MAC receiver after the current frame is received. There may be many frames buffered in the adapter, these are discarded and no further receive frames will be copied to the host. The default state is **disabled**.

Command:	U16	**RxDisable**
Parameters:	none	
Response:	none	

RxFilterWrite **(out-of-band only)**

Defines the value of the receive filter. The 5 active bits in this command may be used in any combination and are shown below in the first parameter of the command. The default value is binary 00111.

Command:	U16	**RxFilterWrite**
Parameter1:	U16	*RxFilter* xxxx1 - individual (must match station address). xxx1x - all multicast (including broadcast). xx1xx – broadcast. x1xxx - all (promiscuous). 1xxxx - multicast hash filter.
Response:	none	

The effect of each bit is additive. That is a 0x00011 pattern would enable individually addressed packets that match the adapter's *station address* as well as all multicast packets. Setting bit 3 (promiscuous) would override bits 2 through 0.

RxFilterRead (out-of-band only)

Read the state of the receive filter. The receive filter is described in the *RxFilterWrite* command.

Command:	U16	**RxFilterRead**
Parameters:	none	

Response:		
Parameter1:	U16	*RxFilter*

ReadStatistics (out-of-band only)

Read the adapters statistics into the response ring.

Command:	U16	**ReadStatistics**
Parameters:	U16	**StatsType**

0 – MAC statistics
1 – Ofload statistics
2 – Error statistics
3 – Internal statistics
4 – Transmit statistics*
5 – MAC Transmit statistics*
6 – Tx Alloc statistics*
7 – Alloc statistics*
8 – Receive statistics*
9 – Command statistics*
10 – IPsec statistics*
11 – Checksum statistics*
12 – TCP Segmentation statistics*

The StatsType parameter selects which statistics are returned. The only statistics externally visible are the MAC statistics, the others are kept as part of the internal monitoring of the system. The selections marked with an asterix are only enabled in debug versions of the microcode.

The response contains the statistics structure, which is defined as shown below.

Response:		
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*TxPackets*
Parameter3:	U32	*TxBytesLo*
Ext Cmd Desc 1:	U32	*TxBytesHi*
	U32	*TxDeferred*
	U32	*TxLateCollisions*
	U32	*TxCollisions*
Ext Cmd Desc 2:	U32	*TxCarrierLost*
	U32	*TxMultipleCollisions*
	U32	*TxExcessiveCollisions*
	U32	*TxFifoUnderruns*
Ext Cmd Desc 3:	U32	T*xMulticastTxOverflows*
	U32	*TxFiltered*
	U32	*RxPacketsGood*
	U32	*RxBytesGoodLo*

Ext Cmd Desc 4:	U32	*RxBytesGoodHi*
	U32	*RxFifoOverruns*
	U32	*BadSSD*
	U32	*RxCrcErrors*
Ext Cmd Desc 5:	U32	*RxOversize*
	U32	*RxBroadcast*
	U32	*RxMulticast*
	U32	*RxOverflow*
Ext Cmd Desc 6:	U32	*RxFiltered*
	U32	*LinkStatus* [0] – link (0 = no link, 1 = link) [1] – speed (0 = 10Mbps, 1 = 100 Mbps) [2] – duplex mode (0 = half, 1 = full) [3:31] – reserved, set to zero
	U32	*unused*, set to zero
	U32	*unused*, set to zero

ClearStatistics (out-of-band only)

Clear the adapters statistics counters.

Command:	U16	**ClearStatistics**
Parameters:	U16	**StatsType** 0 – MAC statistics 1 – Ofload statistics 2 – Error statistics 3 – Internal statistics 4 – Transmit statistics* 5 – MAC Transmit statistics* 6 – Tx Alloc statistics* 7 – Alloc statistics* 8 – Receive statistics* 9 – Command statistics* 10 – IPsec statistics* 11 – Checksum statistics* 12 – TCP Segmentation statistics*

The StatsType parameter selects which statistics counters are to be cleared. The only statistics externally visible are the MAC statistics, the others are kept as part of the internal monitoring of the system. The selections marked with an asterix are only enabled in debug versions of the microcode.

Response: none

CycleStatistics (out-of-band only)

This command initiates a periodic transfer of statistics from the adapter to the host. The rate of update (in microseconds) is supplied by the parameter. The statistics data structure is identical to the one returned by the *ReadStatistics* command.

Command:	U16	**CycleStatistics**
Parameter1:	U16	*unused*, set to zero.

Parameter2:	U32	*CyclePeriod*, number of milliseconds between statistics updates. Zero disables cycle.
Response:	none immediate	

The adapter will periodically transmit a *CycleStatistics* to the host. It is the responsibility of the host to ensure that the ring is purged frequently. The *CycleStatiststics* is described in the **Unsolicited Responses** section.

The following actions stop the statistics update: hardware reset, software reset, issuing the *CycleStatistics* command with the *CyclePeriod* set to zero.

ReadErrors (out-of-band only)

This command is not currently supported. This is a debug command to return a copy of the internal data structure that contains the error counts. The contents of the error structure will be defined later.

Command:	U16	**ReadErrors**
Parameters:	none	
Response:		
	ErrorStruct	

ReadMemory (out-of-band only)

This command is not currently supported. Read data from the adapter to the host. The host specifies an adapter address (bottom 2 bits must be zero) as the source of the read and the number of *words* to transfer. Any number of words can be read from the external memory. Data can also be read from the ARM's internal instruction memory, data memory or from memory mapped registers. In these cases the ARM copies the data to a buffer in external before transferring it to the host. This buffer is 32 bytes long, so reads from these areas can not exceed 128 bytes.

Command:	U16	**ReadMemory**
Parameter1:	U16	*NumWords*
Parameter2:	U32	*AdapterAddr*
Response:		
Parameter1:	U16	*NumWords*
Parameter2:	U32	*AdapterAddr*
Parameter3:	U32	*Value*

WriteMemory (out-of-band only)

This command is not currently supported. Write an 8 bit, 16 bit or 32 bit value to the adapter's memory space. A single read is performed and the host must ensure correct alignment for 16bit and 32 bit values. Only 32 bit writes are permitted to instruction memory, data memory and registers.

Command:	U16	**WriteMemory**
Parameter1:	U16	*WordSize* 0 - 8 bits 1 - 16 bits 2 - 32 bits
Parameter2:	U32	*AdpaterAddr*
Parameter3:	U32	*Value*
Response:	none	

VarSectionRead (out-of-band only)

Read the VAR section in from non-volatile memory. The checksum algorithm used is a simple XOR of the bytes inverting the sum when complete. The ErrorResponse will be returned if the section is not valid in the NVDevice.

Command:	U16	**VarSectionRead**
Parameters:	none	
Response:		
Parameter1:	U16	*checksum*
Parameter2:	U32	*VarLen, number of bytes (each parameter is 2 bytes)*
Parameter3:	U32	*Parameters1,0* Bits[7:0] – LSB parameter0 Bits[15:8] – MSB parameter0 Bits[23:16] – LSB parameter1 Bits[31:24] – MSB parameter1
Ext Resp Desc1:	U32	*Parameters3,2*
	U32	*Parmeters5,4*
	U32	*Parameters7,6*
	U32	*Parameters9,8*
Etc.	…..	……………………

VarSectionWrite (out-of-band only)

Write the VAR section in non-volatile memory. The specified bytes are written into the reserved VarSection page in non-volatile memory. A checksum parameter is included to ensure data integrity. The checksum algorithm used is a simple XOR of the bytes inverting the sum when complete. AcknowledgeResponse will be sent to the host after programming has completed or the ErrorResponse will be sent if the checksum is not valid.

Command:	U16	**VarSectionWrite**
Parameter1:	U16	*checksum*
Parameter2:	U32	*VarLen, number of bytes (each parameter is 2 bytes)*
Parameter3:	U32	*Parameters1,0* Bits[7:0] – LSB parameter0 Bits[15:8] – MSB parameter0 Bits[23:16] – LSB parameter1 Bits[31:24] – MSB parameter1
Ext Resp Desc1:	U32	*Parameters3,2*
	U32	*Parmeters5,4*
	U32	*Parameters7,6*
	U32	*Parameters9,8*
Etc.	…..	……………………
Response:	AcknowledgeResponse	

StaticSectionRead (out-of-band only)

Read the STATIC section in from non-volatile memory. The checksum algorithm used is a simple XOR of the bytes inverting the sum when complete. The ErrorResponse will be returned if the section is not valid in the NVDevice.

Command:	U16	**StaticSectionRead**
Parameters:	none	
Response:		
Parameter1:	U16	*checksum*
Parameter2:	U32	*VarLen, number of bytes (each parameter is 2 bytes)*
Parameter3:	U32	*Parameters1,0* Bits[7:0] – LSB parameter0 Bits[15:8] – MSB parameter0 Bits[23:16] – LSB parameter1 Bits[31:24] – MSB parameter1
Ext Resp Desc1:	U32	*Parameters3,2*
	U32	*Parmeters5,4*
	U32	*Parameters7,6*
	U32	*Parameters9,8*
Etc.	…..	……………………

StaticSectionWrite (out-of-band only)

Write the STATIC section in non-volatile memory. The specified bytes are written into the reserved StaticSection page in non-volatile memory. A checksum parameter is included to ensure data integrity. The checksum algorithm used is a simple XOR of the bytes inverting the sum when complete. AcknowledgeResponse will be sent to the host after programming has completed or the ErrorResponse will be sent if the checksum is not valid.

Command:	U16	**StaticSectionWrite**
Parameter1:	U16	*checksum*
Parameter2:	U32	*VarLen, number of bytes (each parameter is 2 bytes)*
Parameter3:	U32	*Parameters1,0* Bits[7:0] – LSB parameter0 Bits[15:8] – MSB parameter0 Bits[23:16] – LSB parameter1 Bits[31:24] – MSB parameter1
Ext Resp Desc1:	U32	*Parameters3,2*
	U32	*Parmeters5,4*
	U32	*Parameters7,6*
	U32	*Parameters9,8*
Etc.	…..	……………………
Response:	AcknowledgeResponse	

ImageSectionProgram (out-of-band only)

This command is not currently supported.

Command:	U16	**ImageSectionProgram**
Parameter1:	U16	*SectionID*
Parameter2:	U32	*NumBytes*
Parameter3:	U32	*PageOffset*
Ext Resp Desc1:	U32	*checksum*
	U32	*PCI Addr Hi* – most significant 32 bits of PCI address where the page data begins
	U32	*PCI Addr Lo* – most significant 32 bits of PCI address where the page data begins
	U32	*unused, set to*
	U32	*unused, set to zero*
Response:	AcknowledgeResponse	

ReadNVRamPage (out-of-band only)

Read a page of data from the adapter's serial NVRam. The NVRam is divided into 512 pages of 264 bytes (66 word). This command reads one of these pages from the NVRam device.

Command:	U16	**ReadNVRamPage**
Parameter1:	U16	*PageNum (0-511)*
Parameter2:	U32	*unused, set to zero*
Parameter3:	U32	*unused, set to zero*
Response:		
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*Word 0 – Bytes03-00*
Parameter3:	U32	*Word 1 – Bytes07-04*
Ext Resp Desc1:	U32	*Word 2*
	U32	*Word 3*
	U32	*Word 4*
	U32	*Word 5*
Ext Resp Desc2:	U32	*Word 6*
	U32	*Word 7*
	U32	*Word 8*
	U32	*Word 9*
. . . .	. . .	. . . .
Ext Resp Desc16:	U32	*Word 62*
	U32	*Word 63*
	U32	*Word 64*
	U32	*Word 65*

WriteNVRamPage (out-of-band only)

Write a page of data to the adapter's serial NVRam. The NVRam is divided into 512 pages of 264 bytes (66 word). This command writes one of these pages to the NVRam device.

Command:	U16	**WriteNVRamPage**
Parameter1:	U16	*PageNum (0-511)*
Parameter2:	U32	*Word 0 – Bytes03-00*
Parameter3:	U32	*Word 1 – Bytes07-04*
Ext Resp Desc1:	U32	*Word 2*
	U32	*Word 3*
	U32	*Word 4*
	U32	*Word 5*
Ext Resp Desc2:	U32	*Word 6*
	U32	*Word 7*
	U32	*Word 8*
	U32	*Word 9*
. . . .	. . .	. . . .
Ext Resp Desc16:	U32	*Word 62*
	U32	*Word 63*
	U32	*Word 64*
	U32	*Word 65*

Response: AcknowledgeResponse

XcvrSel (out-of-band only)

This command allows control of the tranceiver type.

Command:	U16	**XcvrSel**
Parameter1:	U16	*Xcvrtype*

0 – 10Mbps, half duplex
1 – 10 Mbps, full duplex
2 – 100 Mbps, half duplex
3 – 100 Mbps, full duplex
4 – auto-negotiation

Response: none

TestMux (out-of-band only)

This command allows control of the TestMux register in the MAC. Setting the TXEN bit will

Command:	U16	**TestMux**
Parameter1:	U16	*TestMuxVal*

[0:5] SIGSELECT (Signal Select, see table below)
[6] TXEN (MII TxEn pin)
[7] TRIEN (Tristate enable)
[8:15] TXDATA (MII TxData)

Following table taken for ASIC spec –

SIGSELECT	SIGNAL
0	25 Mhz Mac Clock
1	MII Rx Dv pin
2	MII Rx Er pin
3	MII Col pin
4	MII Crs pin
5	MII 10/100 pin
6	MII Tx Clk pin
7	MII Rx Clk pin
8	MII Link Detect pin
9	PLL clock out
10	Power On Reset
Other	0

Response: none

PhyLoopbackEnable (out-of-band only)

This command enables loopback in the PHY.

Command:	U16	**PhyLoopbackEnable**
Response:	none	

PhyLoopbackDisable (out-of-band only)

This command disables loopback in the PHY.

Command:	U16	**PhyLoopbackDisable**
Response:	none	

MacControlRead (out-of-band only)

Read the MAC Control Register.

Command:	U16	**MacControlRead**
Parameters:	none	
Response:		
Parameter1:	U16	*MACConfig* [0] deferExtendedEnable [4:1] deferTimerSelect [5] enable full duplex [6] allow large packets [7] extend after collision [8] flow control enabled [9] VLT enable

MacControlWrite (out-of-band only)

Write the MAC Control Register. The register bit definition is defined in the description of command *MacControlRead.*

Command:	U16	**MacControlWrite**
Parameter1:	U16	*MACConfig*
Response:	none	

MaxPktSizeRead (out-of-band only)

Read the maximum packet size.

Command:	U16	**MaxPktSizeRead**
Parameters:	none	

Response:		
Parameter1:	U16	*MaxPktSize*

MaxPktSizeWrite (out-of-band only)

Write the maximum packet size

Command:	U16	**MaxPktSizeWrite**
Parameter1:	U16	*MaxPktSize*

Response: none

MediaStatusRead (out-of-band only)

Read the Media Status Register.

Command:	U16	**MediaStatusRead**
Parameters:	none	

Response:
Parameter1: U16 *MediaStatus*

[1:0] reserved
[2] CRC strip disable
[3] reserved
[4] collision detection
[5] carrier sense
[9:6] reserved
[10] polarity reversed
[11] no link detect (0 = link, 1 = no link)
[12:15] reserved

MediaStatusWrite (out-of-band only)

Write the Media Status Register.

Command:	U16	**MediaStatusWrite**
Parameter1:	U16	*MediaStatus*

Response: none

NetworkDiagsRead (out-of-band only)

Read the Network Diagnostics Register.

Command:	U16	**NetworkDiagsRead**
Parameters:	none	

Response:		
Parameter1:	U16	*NetworkDiags* [6:0] reserved [7] statistics enabled [9:8] reserved [10] Rx enabled [11] Tx enabled [12] mac loopback [13:15] reserved

NetworkDiagsWrite (out-of-band only)

Write the Network Diagnostics Register.

Command:	U16	**NetworkDiagsWrite**
Parameter1:	U16	*NetworkDiags*
Response:	none	

PhysicalMgmtRead (out-of-band only)

Command for reading the value in a PHY register.

Command:	U16	**PhysicalMgmtRead**
Parameter1:	U16	*unused, set to zero*
Parameter2:	U32	*PHY register*

Response:		
Parameter1:	U16	PHY register data

PhysicalMgmtWrite (out-of-band only)

Command for reading and writing to the PHY.

Command:	U16	**PhysicalMgmtWrite**
Parameter1:	U16	*PHY register data*
Parameter2:	U32	*PHY register*
Response:	none	

VarParamRead (out-of-band only)

Command for reading a parameter from the NVDevice VarSection.

Command:	U16	**VarParamRead**
Parameter1:	U16	*offset in bytes – each parameter is 2 bytes*

Response:		
Parameter1:	U16	*data*

VarParamWrite (out-of-band only)

Command for writing a parameter to the NVDevice VarSection.

Command:	U16	**VarParamWrite**
Parameter1:	U16	*offset in bytes – each parameter is 2 bytes*
Parameter2:	U32	*data*

Response: none

FireWallControl (out-of-band only)

This command is no longer supported. DHCP prevention is now controlled by the Offload command.

*Enables or disables (default) the personal firewall features. The default for this feature is **disabled**.*

***Command**:*	*U16*	***FireWallControl***
Parameter1:	*U16*	*control*
		[0] – DHCP control
		0 – disable receive IP option descriptor
		1 – enable receive IP option descriptor

***Response**:* *none*

MulticastHashMaskWrite (out-of-band only)

Write the 64 bit multicast hash mask.

Command:	U16	**MulticastHashMaskWrite**
Parameter1:	U16	*MulticastEnable*
		0 – disable hash on mac
		1 – enable hash on mac
		2 – set multicast hash
Parameter2:	U32	*Mask0*, (lsb)
Parameter3:	U32	*Mask1,* (msb)

Response: none

StationAddressWrite (out-of-band only)

Write the adapter's MAC address. If parameters 1 and 2 are zero, the working MAC address will be set back to the factory address. If not, the address contained in parameters 1 and 2 will be checked for validity then stored in the NV device as the new working MAC address. This address is preserved across a reboot or power loss until explicitly written back to the factory address.

Command:	U16	**StationAddressWrite**
Parameter1:	U16	*MACAddrHi,* high 16 bits of address
Parameter2:	U32	*MACAddrLo,* low 32 bits of address
Response:	none	

StationAddressRead (out-of-band only)

Read the adapter's MAC address.

Command:	U16	**StationAddressRead**
Parameters:	none	
Response:		
Parameter1:	U16	*MACAddrHi,* high 16 bits of address
Parameter2:	U32	*MACAddrLo,* low 32 bits of address

StationMaskWrite (out-of-band only)

Write the adapter's Station Mask value.

Command:	U16	**StationMaskWrite**
Parameter1:	U16	*maskHi,* high 16 bits of mask
Parameter2:	U32	*maskLo,* low 32 bits of mask
Response:	none	

StationMaskRead (out-of-band only)

Read the adapter's Station Mask value.

Command:	U16	**StationMaskRead**
Parameters:	none	
Response:		
Parameter1:	U16	*maskHi,* high 16 bits of mask
Parameter2:	U32	*maskLo,* low 32 bits of mask

VlanEtherTypeRead (out-of-band only)

Read the adapter's 802.1q VLAN EtherType.

Command:	U16	**VlanEtherTypeRead**
Parameters:	none	
Response		
Parameter1:	U16	*EtherType*, 2 byte VLAN tag

VlanEtherTypeWrite (out-of-band only)

Write the adapter's 802.1q VLAN EtherType.

Command:	U16	**VlanEtherTypeWrite**
Parameter1:	U16	*EtherType*, 2 byte VLAN tag
Response:	none	

VlanMaskRead (out-of-band only)

Read the VLAN Mask.

Command:	U16	**VlanMaskRead**
Parameters:	none	
Response:		
Parameter1:	U16	*vlanMask*

VlanMaskWrite (out-of-band only)

Write the VLAN Mask.

Command:	U16	**VlanMaskWrite**
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*Mask0*, (lsb)
Parameter3:	U32	*Mask1*
Ext Resp Desc1:	U32	*Mask2*
	U32	*Mask3*
	U32	*Mask4*
	U32	*Mask5*
Ext Resp Desc2:	U32	*Mask6*
	U32	*Mask7*, (msb)
	U32	*unused*, set to zero
	U32	*unused*, set to zero
Response:	none	

BroadcastThrottleWrite (out-of-band only)

Write the broadcast throttle limit. The *BroadcastLimit* parameter specified the amount of broadcasts / multicasts allowed as percentage of bandwidth. Any broadcast / multicast packets exceeding this limit will be discarded. The default value is 100 (no throttling).

Command:	U16	**BroadcastThrottleWrite**
Parameter1:	U16	*BroadcastLimit*, percentage
Response:	none	

BroadcastThrottleRead (out-of-band only)

Read the current *BroadcastLimit.*

Command:	U16	**BroadcastThrottleRead**
Response:		
Parameter1:	U16	*BroadcastLimit*, percentage

DHCPPreventWrite (out-of-band only)

This command is not supported. See command FireWallControl.

Command:	U16	**DHCPreventRead**
Response:	none	

DHCPPreventRead (out-of-band only)

This command is not supported. See command FireWallControl.

Command:	U16	**DHCPreventRead**
Response:	nonte	

ReceiveBufferControl (out-of-band only)

Informs the adapter of the format of the receive buffer. The host can specify any padding to be used when DMAing received packets to host .This feature should correspond to the cache line size in host memory to allow for more efficient DMA transfers over the PCI bus.

Command:	U16	**ReceiveBufferControl**
Parameter1:	U16	*Unused*
Parameter2:	U16	*Padding*, The adapter will always pad the received packet to this number of bytes (default 0 – no padding)

Response: none

SoftwareReset (out-of-band only)

Issue a software reset to the adapter. The ARM jumps to its reset vector and all context and previously downloaded code is lost. The ARM will disable the transmitter and receiver and reset all its ring indexes to zero.

Command:	U16	**SoftwareReset**
Parameters:	none	

Response: none

CreateSA (**in-band or out-of-band**)

Issues a SA (Security Association) creation request. Note this is an entry for Security Association, part of the SAD (SA Database), not part of the SPD (Security Policy Database). It is tied to either a single AH or a single ESP.

SATYPE_IPSEC_AH and SATYPE_IPSEC_ESP define the SA for Ipsec. For other SA types, such as those needed for local file encryption or signature that uses the adapter as a BITW coprocessor, the SAType is SATYPE_LOCAL.

An SA is identified by the triplet <SAType, SPI, DestAddress>, that is used to search for the matching SA on Rx Network Packet processing.

The adapter does not maintain the life time of a SA. The host must issue a DeleteSA command when the lifetime duration of a SA expires.

Note that there is no Selector specification in the SA structure. The adapter does not provide Selector matching and filtering functions.

(NOTE: The ability to specify an index is a Typhoon1.1 feature. In Typhoon 1.0 code, the adapter always specifies the index).

The supported algorithms for Ipsec are:

DES in CBC mode with explicit IV
Triple-DES in CBC mode with explicit IV
Triple DES in CBC mode
HMAC with MD5
HMAC with SHA-1

The supported algorithms for BITW are:

DES in CBC mode with explicit IV
Triple-DES in CBC mode with explicit IV
MD5
SHA-1
HMAC with MD5
HMAC with SHA-1

Command:	U16	**CreateSA**
Parameter1:	U16	*Ipsec Proto Type* 0 – NULL - no hash or encryption 1 – AH 2 – ESP
Parameter2:	U8	*HashFlags* [0:8] – Hash Flags [0] –hash enable [1] – SHA-1 [2] – MD5 [3:7] – *unused*, set to zero
	U8	*Direction* [0] – Direction, 0 – Receive, 1 – Transmit [1:7] – *unused*, set to zero
	U8	*EncryptionFlags* [0] – encryption enable [1] – *singleDES*, 0-triple DES, 1-single DES

		[2] – *3Key*, 0-3DES, 2 key, 1-3DES, 3 key
		[3] – *CBC*, 0-ECB, 1-CBC
		[4:7] – *unused*, set to zero
	U8	*SpecifyIndex*, 0 – Typhoon returns index, 1 – force to *Index* field *(NOTE: This is a Typhoon1.1 feature, the value is always 0 in Typhoon 1.0 code)*
Parameter3:	U32	*SPI*, Security Parameters Index
Ext Cmd Desc1:	U32	*DestSpec,* Destination IP Address
	U32	*DestMask*, Dest IP Address subnet mask
	U32	*HASH Key 0*
	U32	*HASH Key 1*
Ext Cmd Desc2:	U32	*HASH Key 2*
	U32	*HASH Key 3*
	U32	*HASH Key 4*
	U32	*Encrypt Key 0 Lo*
Ext Cmd Desc3:	U32	*Encrypt Key 0 Hi*
	U32	*Encrypt Key 1 Lo*
	U32	*Encrypt Key 1 Hi*
	U32	*Encrypt Key 2 Lo*
Ext Cmd Desc4:	U32	*Encrypt Key 2 Hi*
	U32	*Index*, if *SpecifyIndex* = 1 then this entry is added to the index specified here. *(NOTE: This is a Typhoon1.1 feature, the value is always 0 in Typhoon 1.0 code)*
	U32	*Unused,* set to zero
	U32	*Unused,* set to zero

Not all of the key parameters are valid in all cases. The following table indicates which options use the different key combinations. Unused keys must be set to zero.

SHA-1	uses	HASH keys 0-4
MD5	uses	HASH keys 0-3
DES	uses	Encrypt key 0
3DES, 2 key	uses	Encrypt key 0-1, duplicate key 0 to key 2
3DES, 3 key	uses	Encrypt key 0-2

Response:

Parameter1:	U16	*SAId*, SA Identifier (0 – error defining SA)

DeleteSA (out-of-band only)

Issues a SA (Security Association) deletion request. Note this is an entry for Security Association, part of the SAD (SA Database), not part of the SPD (Security Policy Database). It is tied to either an AH or an ESP.

Command:	U16	**DeleteSA**
Parameter1:	U16	*SAId*, SA Identifier

Response: none

ReadIpsecInfo *(NOTE: This is a Typhoon1.1 command)*

Read the sizes of the Transmit and Receive SA tables from the adapter.

Command: U16 **ReadIpsecInfo**

Response:

Parameter1:	U16	*IpsecEnables*, 0 – not enabled, 1 – enabled
Parameter2:	U16	TxSaTableSize, max number of entries in Tx SA table
Parameter3:	U16	RxSaTableSize, max number of entries in Rx SA table
Parameter4:	U16	TxSaEntries, number of valid entries in Tx SA table
Parameter5:	U16	RxSaEntries, number of valid entries in Rx SA table

T2: Note that the IpsecEnables field shown above in Parameter1 is there for backward compatibility, and may not provide the necessary information. Effectively, this bit will only indicate if the 56 bit IPSec features are enabled, and does not provide any on the 168 bit section. See the **GetIPSecEnable** command for information on how to check the enable status of the T2 chip.

TestComplete (out-of-band-only)

Response: U16 **TestComplete**

This is a diagnostic test response – see the Touchdown specification

DiagArm2HostComm (out-of-band-only)

Response: U16 **DiagArm2HostComm**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagPciDmaDoneInt (out-of-band-only)

Response: U16 **DiagPciDmaDoneInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagPciDmaPingPong (out-of-band-only)

Response: U16 **DiagPciDmaPingPong**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagHost2ArmComm (out-of-band-only)

Response: U16 **DiagArm2HostComm**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagPciDmaInt (out-of-band-only)

Response: U16 **DiagPciDmaInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagRxDmaDoneInt (out-of-band-only)

Response: U16 **DiagRxDmaDoneInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagTxDmaDoneInt (out-of-band-only)

Response: U16 **DiagTxDmaDoneInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagSwDmaDoneInt (out-of-band-only)

Response: U16 **DiagSwDmaDoneInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagSwInt

Response: U16 **DiagSwInt**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagMacTxComplete

Response: U16 **DiagMacTxComplete**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification.

DiagOneShot

Response: U16 **DiagOneShot**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification for detail.

DiagFreeTimer

Response: U16 **DiagFreeTimer**

This is a diagnostic test supported by diagnostic image only– see the Touchdown specification for detail.

VersionsRead (out-of-band only)

Read the version of the image currently running.

Typhoon 1.0 implements the following **VersionsRead** command. This command is still valid for images other than the runtime image.

```
Command:        U16    VersionsRead
Parameters:     none

Response:
Parameter1:     U16    TyphoonASICVersion,SidewinderVersion
                          [15:8 ]   Typhoon ASIC Revision ID
                          [7:0]     Sidewinder chip revision
Parameter2:     U32    SWVersion – current image running
                          [31:16] –  1 (boot image – reserved)
                                     2 – sleep image
                                     3 – runtime image
                                     4 – diagnostic image
                          [15:0] -  month/day
```

Typhoon 1.1 implements the following command:

Command:	U16	**VersionsRead**
Parameters:	none	

Response:

Parameter1:	U16	*RuntimeType* - 0 – Typhoon 1.1, 1 – ECB aware
Parameter2:	U32	*RuntimeImageVersion* – current image running in format a.b.c.d [31:24] – a [23:16] – b [15:8] – c [7:0] – d
Parameter3:	U32	MemorySize – the size of Attic and Barn memory in 128k byte increments. [31:16] – Barn memory size 0 – no Barn memory 1 – 128k 2 – 256k 4 – 512k 8 – 1024k [15:0] – Attic memory size 1 – 128k 2 – 256k 4 – 512k
Ext Cmd Desc1:	version string (see below)	
Ext Cmd Desc2:	version string (see below)	

The version string contains an ASCII representation of the image build time, date and the unique build number, it is a NULL terminated string of 25 characters, it has the form:

"hh:mm:ss mm/dd/yy nnnnnnnn"

where *nnnnnnnn* is a unique 8 digit hexadecimal build number.

TransmitWaveform (out-of-band only)

Support for manufacturing,

Command:	U16	**TransmitWaveform**
Parameter1:	U16	*Waveform* 0 – disable waveform 1 – 100Mbps MLT3 long puls 2 – 100 Mbps unscrambled idle 3 – 10 Mbps pattern generation 4 – DC level out

Response: none

DefineRmonFilter (out-of-band only)

Define an RMON filter. The Typhoon can store up to 15 filters, numbered 0 – 14. Each filter requires 3 fields each up to 256 byte long (*Data*, *DataMask* and *DataNotMask*) and a *Filter Offset* which specifies the offset (in number of byte from the start of the packet) to where the filtering should start. The *Filter Offset* must be word (32 bit) aligned from the start of the packet.

This command specifies the physical address and the size of a *Filter Data Block*, which contains the description of 1 to 15 filters. The internal format of the *Filter Data Block* is described below:

Command:	U16	**DefineRmonFilter**
Parameter1:	U16	*Filter Block Size*, # bytes in the filter block.
Parameter2:	U32	*Filter Block Addr Low* – least significant 32 bits of PCI address of filter block.
Parameter1:	U32	*Filter Block Addr Hi* – most significant 32 bits of PCI address of filter block.

Response:		
Parameter1:	U16	*ReturnCode* 0 – add failed 1 – frame successfully added

Filter Data Block:

32bit value:	*NumID*;
16bit value:	*DataLen*;
16bit value:	*DataMaskLen*;
16bit value:	*DataNotMaskLen*;
16bit value:	*Offset*;
16bit value:	Data*ProtocolID*;
16bit value:	*ProtocolID*;
8 bit values:	*Data[DataLen]*
8 bit values:	*DataMask[256]*
8 bit values:	*DataNotMask[256]*

The response indicates that the adapter has transferred all data in the buffer and it can now be freed.

RmonFilterCapabilities (out-of-band only)

Command to determine how many receive filters are supported by the adapter and which have been enabled.

Command:	U16	**RmonFilterCapabilities**

Response:		
Parameter1:	U16	*NumFilters* – number of full size filters supported by Typhoon
Parameter2:	U16	*EnableMask* – bit mask, one bit for each filter indicating if that filter is enabled or disabled.

RmonFilterEnable (out-of-band only)

Command enables of disables one or more of the previously defined receive filters. Each filter is represented by a single bit, filter 1 – bit 0, filter 15, bit 14.

Command:	U16	**RmonFilterEnable**
Parameter1:	U16	*EnableMask* – bit mask, one bit for each filter, 1 enables a filter, 0 disables it

MacBackoffWrite (out-of-band only)

The host can control the MAC collision backoff time by writing a value to this register.

Command:	U16	**MacBackoffWrite**
Parameter1:	U16	*BackoffValue* – value to write to MAC backoff alogrithm

AddWakeupFrame (out-of-band only)

Adds a wakeup frame to the adapter's list. This frame is defined in the Adapter Sleep section. Both he mask and the pattern buffer portions of the frame must be in a physically contiguous data buffer for the adapter to download, though they need to be contiguous to each other.

This command is only supported in the Sleep Image.

Command:	U16	**AddWakeupFrame**
Parameter1:	U16	*Mask len* - length of buffer containing mask to be downloaded.
Parameter2:	U32	*PCI Addr Hi* – most significant 32 bits of PCI address of buffer containing mask.
Parameter3:	U32	*PCI Addr Low* – least significant 32 bits of PCI address of buffer containing mask.

.

Ext Cmd Desc1:	U32	*Pattern len* – length of buffer containing pattern to be matched
	U32	*PCI Addr Hi* – most significant 32 bits of PCI address of buffer containing mask.
	U32	*PCI Addr Low* – least significant 32 bits of PCI address of buffer containing mask
	U32	*unused*, set to zero

Response:		
Parameter1:	U16	*ReturnCode* 0 – add failed 1 – frame successfully added

AddSleepFrame (out-of-band only)

Adds a sleep frame to the adapter's list of packets to be sent periodically. This frame is defined in the Adapter Sleep section. The frame must be in a physically contiguous data buffer for the adapter to download.

This command is only supported in the Sleep Image.

Command:	U16	**AddSleepFrame**
Parameter1:	U16	*Data len* - length of buffer to be downloaded.
Parameter2:	U32	*PCI Addr Hi* – most significant 32 bits of PCI address of buffer containing frame.
Parameter3:	U32	*PCI Addr Low* – least significant 32 bits of PCI address of buffer containing frame.
.		
Ext Cmd Desc1:	U32	*Xmit interval* – number of milliseconds between transmitting instances of this frame.
	U32	*unused*, set to zero
	U32	*unused*, set to zero
	U32	*unused*, set to zero

Response:		
Parameter1:	U16	*ReturnCode* 0 – add failed 1 – frame successfully added – There is only a limited number of slots available. Requests that exceed this hardcoded limit are not added

T2: This command may be removed or limited to free up memory constraints. For each packet, it takes (1514 + 1514/8 + a few) bytes of attic memory to preserve the frame.

EnableWakeupEvents (out-of-band only)

Command for selecting which events should cause a wakeup. This command will override any values stored in the EEPROM for magic packet or link event enables.

Command:	U16	**EnableWakeupEvents**
Parameters:	U16	*Events* [0] magic packet enable [1] link event enable [2] ICMP echo request enable [3] ARP request enable
Response: none		

EnableSleepEvents (out-of-band only)

Tells the adapter what sleep events to support. Contains the parameters necessary to do so.

This command is only supported in the Sleep Image.

Command:	U16	**EnableSleepEvents**
Parameter1:	U16	*SleepEventMask* [0] - MAC keep-alives [1] – ARP replies [2] – Master Browser Host Announcements [3] – Wake-on-timer (DHCP alarm clock) [4] – DHCP lease renew [5] – Respond-to-Ping [6] – Novell Watchdog replies
Parameter2:	U32	*unused*, set to zero
Parameter3:	U32	*unused*, set to zero
Ext Cmd Desc1:	U32	*IP address*
	U32	Subnet mask
	U32	Default gateway
	U32	*Wake-on-error.* If non-zero, Typhoon will wake system upon an error
Ext Cmd Desc2:	U23	IPX address
	U32	*MB Data len.* Length of Master Browser info data field
	U32	*MB PCI Addr Hi* – most significant 32 bits of PCI address of buffer containing Master Browser info data field
	U32	*MB PCI Addr Low* – most significant 32 bits of PCI address of buffer containing Master Browser info data field. Format of data field: Bytes 0-23: computer name (padded with 0) Bytes 24-47: workgroup name (padded with 0) Bytes 48+: computer description
Ext Cmd Desc3:	U32	*Wakeup time* - # of milliseconds from now to wakeup.
	U32	*DHCP Data len.* Length of DHCP client indentifier
	U32	*DCHP PCI Addr Hi* – most significant 32 bits of PCI address of buffer containing DHCP client indentifier
	U32	*DHCP PCI Addr Low* – most significant 32 bits of PCI address of buffer containing DHCP client indentifier
Response:	none.	

GoToSleep (out-of-band only)

Tells the adapter to begin sleep functions. Upon receipt of this command, the adapter will write "STATUS_SLEEPING" to Arm2Host register 0. The adapter must be manually woken by issuing a HOST_CMD_WAKEUP through Host2Arm register 0.

This command is only supported in the Sleep Image.

Command:	U16	**GoToSleep**
Response:	none	

GetIpAddress (out-of-band only)

Read the currently stored IP address in the adapter. Note that this is merely the last IP address used as a source on outgoing packets. If using IPSec with tunneling, or multiple IP stacks on the same adapter, this address is likely to be useless.

Command:	U16	**GetIpAddress**
Parameters:	none	
Response:		
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*Address*

TcpSegmentResponse

Unsolited response, see unsolicited response section for detail.

Response:	U16	**TcpSegmentResponse**

ReadPCIConfigReg (out-of-band only)

Reads the value of the PCI configuration register at the given offset and returns that value to the host.

Command:	U16	**ReadPCIConfigReg**
Parameter1:	U16	*register offset* – offset of register to be read (see PCI spec.)
Response:		
Parameter1:	U32	*offset* – the register read
Parameter2:	U32	*value* – the value read from the above register.

WritePCIConfigReg (out-of-band only)

Writes the value given into the PCI config register at the given offset

Command:	U16	**WritePCIConfigReg**
Parameter1:	U16	*register offset* – offset of register to be written (see PCI spec.)
Parameter2:	U32	*value* – value to write into register
Response:	none	

OffloadWrite (out-of-band only)

Enable or disable features in the adapter firmware.

Command:	U16	**OffloadWrite**
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*Offload Tx Flags.* Set to a 1 to enable the features: [0] – Reserved set to zero, Classify Bit [1] – TCP Checksum enable. [2] – UDP Checksum enable. [3] – IP Checksum enable. [4] – IPSEC enable. [5] – Broadcast Throttling enable. [6] – DHCP Prevention enable. [7] – VLAN enable. [8] – Filtering enable. [9] – TCP Segmentation enable. [10-31] – Unused
Parameter2:	U32	*Offload Rx Flags.* Set to a 1 to enable the features: Same as *Offload Tx Flags*

OffloadRead (out-of-band only)

Read the capabilities of the adapter and which of these features are enabled.

Command:	U16	**OffloadRead**
Parameters:	none	
Response:		
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	*Tx Capability Flags*. If set to a 1 then the adapter is capable of the specified features: [0] – Reserved set to zero, Classify Bit [1] – TCP Checksum enable. [2] – UDP Checksum enable. [3] – IP Checksum enable. [4] – IPSEC enable. [5] – Broadcast Throttling enable. [6] – DHCP Prevention enable. [7] – VLAN enable. [8] – Filtering enable. [9] – TCP Segmentation enable. [10-31] – Unused
Parameter3:	U32	Rx *Capability Flags*. If set to a 1 then the adapter is capable of the specified features: Same bit definitions as *Tx Capability Flags*
Ext Cmd Desc1:		
	U32	*Tx Enabled Offload Flags*. If set to a 1 then the feature is enabled in the adapter Same bit definitions as *Tx Capability Flags*
	U32	*Rx Enabled Offload Flags*. If set to a 1 then the feature is enabled in the adapter Same bit definitions as *Tx Capability Flags*
	U32	*unused*
	U32	*unused*

SOSRead (out-of-band only)

Read the state of the SOS pins. This command is implemented by the manufacturing diagnostic image.

Command:	U16	**SOSRead**
Parameters:	none	
Response:		
Parameter1:	U16	State of the SOS pins. [0] – SOS0 [1] – SOS1 [2] – SOS2 [15:3] – *unused*, set to zero

SetPMEStatus (out-of-band only)

Command the adapter to set the state of the pmeStatus bit in the PowerMgmtCtrl register. This command is implemented by the manufacturing diagnostic image.

Command:	U16	**SetPMEStatus**
Parameter1:		*pmestate – 0 or 1*
Response:	none	

MfgWOL **(out-of-band only)**

Command the adapter to set the pmeStatus bit when a change in link state occurs. This command is implemented by the manufacturing diagnostic image.

Command:	U16	**MfgWOL**
Parameters:	none	
Response:	none	

GetIPSecEnable (out-of-band only)

Returns the current IPSec Enable State of the card, the Serial number of the ASIC, and if this is a T1 vs. T2 ASIC.

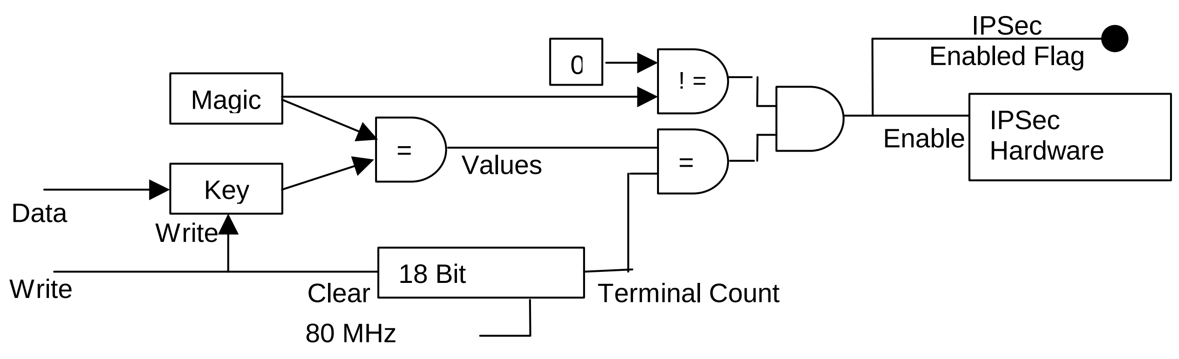
Command: U16 **GetIPSecEnable**

Response:		
Parameter1:	U16	*ReturnCode* [0] 56 Bit IPSec Enabled [1] 168 Bit IPSec Enabled
Parameter2:	U32	Unique Serial Number (not used)
Parameter1:	U32	0) This is a T1 card, with fixed IPsec features 1) This is a T2 card with variable IPSec features

Hardware features interfacing to:

The T2 ASIC may contain a **unique serial number** that is used in an external database lookup to provide two unique magic keys. It also contains two sets of Magic/Key values, one to Enable the 56 bit IPSec features, and another the 168 bit features. The Magic number is programmed at ASIC manufacture time, and cannot then be either read or written to. When the Key value is then written so that it matches the Magic number, the IPSec logic feature will then be enabled.

The IPSec features in the T2 Hardware will become available approximately 3.28 mSec after the correct key has been written. The Hardware places an 18 bit counter at TBus speeds (80 MHz in normal operating mode) between output of the compare logic and the IPSec enable signal. The counter is zeroed when the Enable register is written causing a delay before we know if the key that was supplied is correct or not. This works out to 114 years try every possible combination.



The TBus speed is NOT reduced to 10 MHz in sleep mode. In other words, the delay remains constant regardless of the mode the process is in, unless it's in D3Disable that is, when all of the clocks are stopped.

There is no method available to determine the state of the 18 Bit Delay counter. In other words, the **IPSec Enable Flag** shown above will be zero if read before the timer expires, even if a proper key was written. This is the flag whose current value is returned in the **GetIPSecEnable** function. Thus it is up the calling software to manage the delay if needed.

The Enable register is cleared whenever the TBus receives a Soft Reset. The firmware is responsible for rewriting the key registers if this occurs.

Because of the way that the Hardware is designed, the 56Bit IPSec features must be correctly enabled in order for the 168 Bit portion to work.

Be aware that the ASIC Hardware itself has a special meaning a Magic Number of 0. Even if the Keys are written with the same 0 value, there is extra logic to keep the IPSec features from being enabled.

```
if (magic != 0)
       enable = (magic ==  key);
else
       enable = FALSE;
```

Actual software interface to the hardware

NOTE: As of June 2000, it was decided that due to the reduction of the restrictions in the US laws for shipping encrypted products, that the unique serial and magic number enable features of the T2 ASIC was overkill. However it was too late in the process to remove the logic from the ASICS that control this. Thus it was decided that all ACICS be programmed with the default values as shown below:

Serial Number	0xFFFF-FFFF	
56Bit Enable Key	0xFF-FFFF-FFFF	== default value of 56Bit magic key
168Bit Enable Key	0xFF-FFFF-FFFF	== default value of 168bit Magic key

Because of this simplification, the method that was designed to allow the OS device drivers to write to the Keys was deemed unneeded and removed from the Typhoon command set. Instead, a more general method based on **Snipit**'s was devised.

A **Snipit** is a code fragment that can either overlay existing code within the firmware, or hook itself into the firmware's data control and interrupt path much like a Device Driver. They are intended to allow OEM and product specific features to be added to the base firmware, without having to relink the firmware each time. See the T2 Snipit Architecture document for more details.

There are 3 flavors of the **T2 IPSec Enable Snipit,** one to disable IPSec, one to Enable 56 bit, and one to enable both 56 and 168 bit IPSec. Regardless of the capabilities that **T2 IPSec Enable Snipit** enables, its name will always be **el99cryp.snp**. The description of how the **Snipit's** are created and end up on the User's machine is in the section on **T2 Encryption Enable** and other White papers.

The **T2 IPSec Enable Snipit** file may or may not exist on the hard drive, and it's actually capabilities are transparent to anyone but the Firmware. Thus it is very important that the OS device driver inquire the capabilities of the IPSec features after the download has occurred.

Sequence of steps required to Enable IPSec Offload features on all Typhoon cards.

- The T2 ASIC, as delivered from the ASIC manufacture, will be an enabled device.
- The EEBoot code will use this feature to run POST internal Loopback tests on the IPsec hardware, and exit POST with an error code if the Loopback fails. This is the same behavior as T1 cards with sidewinders.
- On T2 devices, the EEBOOT code will disable the IPSec features of the ASIC by writing something other than 0xFF-FFFF-FFFF into both the 56 bit and 168 bit enable keys.
- When the OS device driver initializes, it is responsible for using the **GetIPSecEnabl**e command to see if the card can be enabled, (I.E it's a T2 card), and then downloading a Snipit file by the name of **el99cryp.snp** -- if it exists --, from the User's hard disk via the **SnipitDownload** command.
- Once the download is complete, and the 3.28ms delay period has occurred, the driver can then inquire either the **GetIPSecEnable** command, or the **OffloadRead** command to determine the actual IPSec features supported by the card, and configure itself accordingly

T2 vs T1 implementation differences:

In terms of the device driver, the biggest thing that it needs to recognize is if the firmware has this **GetIPSecEnable** command or not. T1 cards, with T1 firmware (pre July 2000), do not support this command. T2 cards will be the first to ship (Aug 2000) with it.

However, since the Firmware is designed to be backward compatible, Device drivers may discover T2 firmware on T1 cards, and will need to know how to handle it. T1 cards are "fixed enabled", while the OS Device driver must enable the T2 versions.

The OS Device driver can recognize T2 vs T1 cards via either the PCI id of the card, or by looking a the value returned in Parameter3 of the response to this command.

The **T2 IPSec Enable Snipit** is then downloaded to the T2 card. As it turns out, this Snipit can also be downloaded to the T1 cards without any negative side effects. It just won't effect the actual enablement of the T1 card in any manor what so ever.

In all cases, Parameter1 of the response to this **GetIPSecEnable** command will contain the actual currently useable IPSec features of the card.

SnipitDownload (out-of-band only)

A **Snipit** is a code fragment that can either overlay existing code within the firmware, or hook itself into the firmware's data control and interrupt path much like a Device Driver. They are intended to allow OEM and product specific features to be added to the base firmware, without having to relink the firmware each time. See the T2 Snipit Architecture document for more details.

The actual Snipit files are heavily version, check-summed and typed by the firmware. There is no need for the OS device driver to do any verification on the file. Its only requirement is to take the correct action based upon the multitude of return codes that the firmware can return.

Command:	U16	**SnipitDownload**
Parameter1:	U16	*unused*, set to zero
Parameter2:	U32	Host address of buffer containing entire Snipit.
Parameter2:	U32	Length in bytes of the Snipit.

Response:

Parameter1:	U16	SNIP_ERR *ReturnCode* Will be one of the following values. See snipit.h.

```
SNIP_ERR_OK = 0,                // no error
SNIP_ERR_NOT_IMPLEMENTED,       // feature will be implemented later
SNIP_ERR_MAGIC,                 // bad magic number in header
SNIP_ERR_ALIGNMENT,             // compiler alignment errors
SNIP_ERR_LENGTH,                // len of snippet does not match input
SNIP_ERR_VERSION,               // snippet version mismatch
SNIP_ERR_FIRMWARE_VERSION,      // snipit != firmware version.
SNIP_ERR_INVALID_ID,            // unknown snippet id
SNIP_ERR_HEADER_CHECKSUM,       // checksum invalid
SNIP_ERR_IMAGE_CHECKSUM,        // data checksum invalid
SNIP_ERR_BAD_DYN_MEM_PTR,       // unexpected release location
SNIP_ERR_NO_PCBS_AVAILABLE,     // out of PCBs to store snippet
SNIP_ERR_NO_DYN_MEM,            // out of dynamic memory
```

Parameter2:	U32	Handle to Snipit PCB if relocateable Snipit.

The Snipit PCB is a internal firmware data structure that indicates how the resources for this Snipit have been allocated. It is used by other functions to inquire/modify/remove the Snipit. This value will be NULL if a “static” non-relocateable Snipit was downloaded.

The following code downloads a Snipit in DOS. Device drivers need to accomplish the same set of steps.

```
/****************************************************************************/
/* int snippet_download_command
**
** Description:     Working portion of snippet download. Takes buffer
**                  and it's length and DMA's it down to the typhoon
*/
SNIP_ERR
snippet_download_command(U32 fileBuf, U32 fileLen)
{
    HOST_CMD_DESC *respPS, cmdS = { 0 };
    U32 readUL;
    SNIP_ERR nRet = 0;
    cmdS.cmdU.cmdFlagsS.frNumDescUC = 0;     /* no ext desc */
    cmdS.cmdU.cmdFlagsS.frFlagsUC   = FRAME_TYPE_CMD_HDR | HOST_DESC_VALID;
    cmdS.cmdU.cmdFlagsS.frOptionUW  = CMD_snipit_download;
    cmdS.cmdParam1UW                = 0x0000;
    cmdS.cmdParam2UL                = (U32)fileBuf;
    cmdS.cmdParam3UL                = fileLen;

    readUL = HostVarsPS->hvReadS.regRespReadUL;
    if ( send_cmd_outOfBand_b(&cmdS, (U32*)0) ) {

                                    // wait for response

        readUL = HostVarsPS->hvReadS.regRespReadUL;
        while (readUL == HostVarsPS->hvWriteS.mr.regRespWriteUL) {
            if ( kbhit( ) ) {
                printf( "aborted - download failed to return\n" );
                nRet = -1;
                break;
            }
           readUL = HostVarsPS->hvReadS.regRespReadUL;
        }
        if (!nRet && readUL != HostVarsPS->hvWriteS.mr.regRespWriteUL) {
            respPS = (HOST_CMD_DESC*)((U8*)RespRingPS + readUL);
            nRet = respPS->cmdParam1UW;     // firmware error response here
        }
    }
    return( nRet );
}
/*****************************************************************************/
/* int download_snippet_file
**
** Description:     Asks for the name of a file, reads it into a buffer
**                  and then sends that buffer to the typhoon via
**                  a CMD
*/
int
download_snipit_image(char *szFName)
{
    U8 *fileBuf;
    SNIP_ERR nRet;
    U32 phy_fileBuf;
    U32 fileLen;
    U8 *pBuf;
    FILE *fp;

    fp = fopen( szFName, "rb" );
    if ( fp == NULL ) {
        printf( "ERROR: cannot open %s: %s\n", szFName, sys_errlist[errno] );
        return( 0 );
    }
    fseek( fp, 0L, SEEK_END );
    fileLen = ftell( fp );                  // length of the file
    fseek( fp, 0, SEEK_SET );

    fileBuf = malloc( (U8)(fileLen + 4));
    pBuf = (U8 *)((((U32)fileBuf / 4) + 1) * 4);  // align at 4 byte boundary
    fread( pBuf, 1, (int )fileLen, fp );
    fclose( fp );

    phy_fileBuf = pptr( pBuf );            // Convert the DOS addr to Physical

    nRet = snippet_download_command( phy_fileBuf, fileLen );
    if (nRet < SNIP_ERR_LAST)
        printf("%s\n", szSnipErr[nRet]);
    else
        printf("Undefined SNIP_ERR %d\n", nRet);
    free( fileBuf );
    return( 1 );
}
```

TyphoonCmdLevel (out-of-band only)

Command:	U16	TyphoonCmdLevel (0x69)
Parameters:	none	

Response:

Parameter1:	U16	[0:7] Minor Level [8:15] Major Level

We should insure as much as possible that old revisions of the driver and firmware can still operate with Typhoon 2 drivers and firmware. This will help to avoid problems both within 3Com as well as with the user base.

This is especially true as we extend existing commands. For example, I'm extending the XcvrSel command and will use Parameter2[0] for Flow Control (1=enable, 0=disable). If the Typhoon 2 firmware gets this command from a 1.1 driver, then it will disable Flow Control by mistake since the NDIS driver doesn't use Parameter2.

The major/minor level values would reflect the product release levels (e.g., 2.0 for Typhoon 2). These values are not intended to show the build or revision number of the host driver.

The semantics of this command are the following:

- The host driver will prepare a TyphoonCmdLevel during initialization. It will set Parameter1 to its major and minor product levels.
- It will issue the command to the runtime image. The firmware will set the response Parameter1 value to its major and minor product levels.

4.2 Unsolicited Responses

HelloResponse

The adapter will send a *HelloResponse* to the host if it has not received any communication from the host for a set period of time. The host must issue a command (any command) or other transaction in response to this to indicate it is still alive. If the adapter get no response to a number of these, then it will assume the host has died and take appropriate action.

Response:	U16	**HelloResponse**
Parameters:	none	

TcpSegmentResponse

Response:	U16	**TcpSegmentResponse**
Parameter1:	U16	*Error Code* 0 – no error 1 – invalid header 2 – insufficient buffer space 3 – adapter internal error 4 – network error 5 – URG bit set in TCP header 6 – SYN bit set in TCP header 7 – RST bit set in TCP header
Parameter2:	U32	*Virtual Address Low*, address of data buffer
Parameter3:	U32	*Virtual Address High*
Ext Resp Desc1:	U32	*NumBytes*, number of bytes successfully transmitted.
	U32	*unused, set to zero*
	U32	*unused, set to zero*
	U32	*unused, set to zero*

MediaStatusRead (out-of-band only)

Response: U16 **MediaStatusRead**
The remainder of the response is identical to the *MediaStatusRead* response.

CycleStatistics

Response: U16 **CycleStatistics**

The remainder of the response is identical to the *ReadStatistics* response.

TestComplete

Response: U16 **TestComplete**

This is a diagnostic test response – see the Touchdown specification

5. Manufacturing Support

5.1 NVDevice configuration

The NV Device on the Typhoon adapter uses a serial SPI interface to connect to Typhoon. The NV Device is a 128K byte Atmel DataFlash and is read/write-able from the Typhoon Processor, PCI interface and Typhoon Command Interface. See Atmel AT45DB021 Data Sheet for details beyond the scope of this document.

T2 Server cards can be stuffed with up to 1Meg of flash if required. Note that the physical dimension of the flash is different when it's greater than 128k Bytes. The T2 Server board (June 2000) was laid out to support both the 128k Byte chip and the larger 256k, 512k and 1Meg Byte chips. There may be software that needs to be modified in order to support this however.

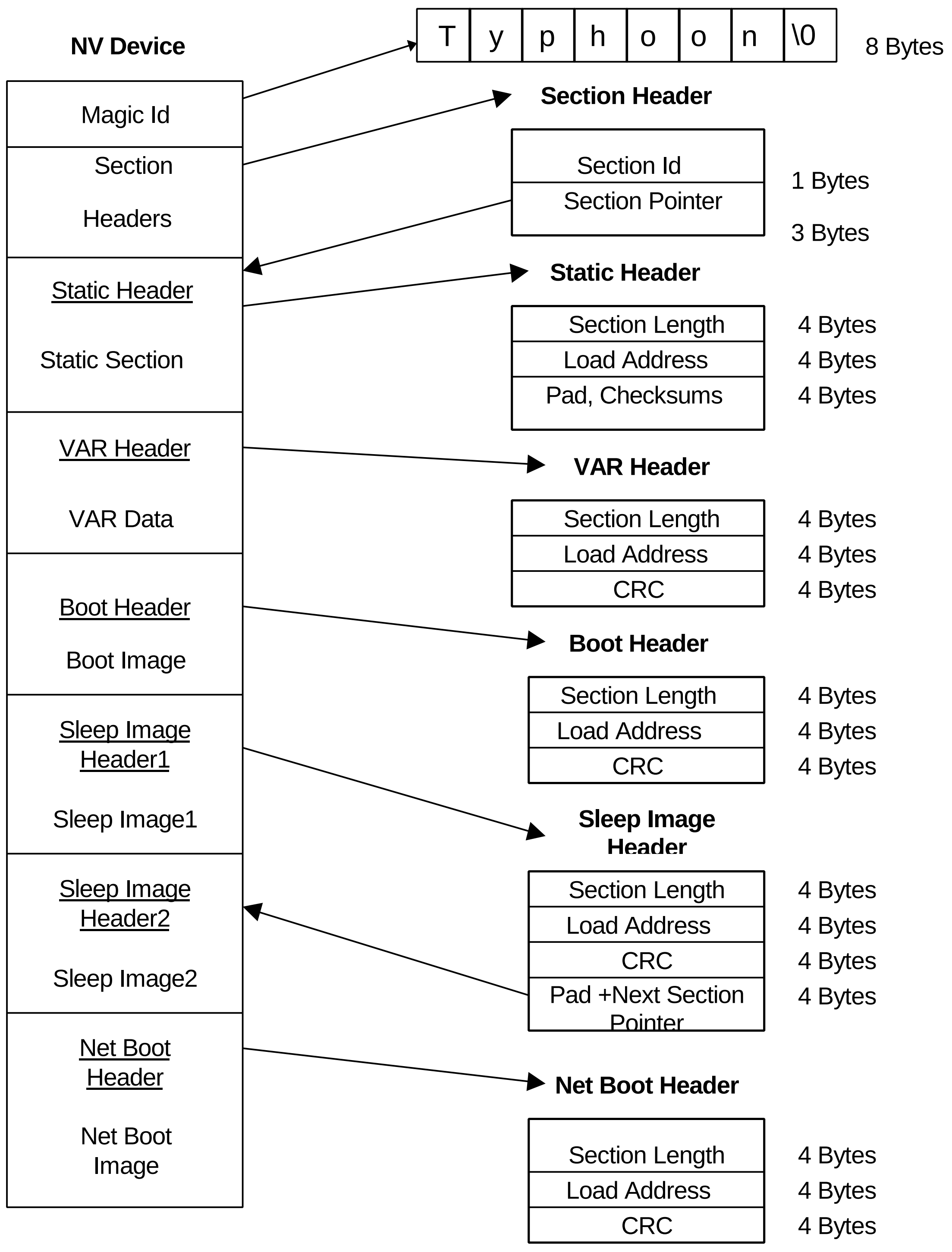
The NV Device contains Typhoon Boot code, Sleep Image, Net Boot code, and Typhoon non-volatile parameters for configuration and static parameters such as MAC Address and serial numbers.

The checksum algorithm used for the images in the NV Device is a simple XOR of the bytes inverting the sum when complete. Each images and header sections contain a separate checksum.

The complete NV Device Flash is divided into different sections and each section is fixed the start of one of the 264 long byte pages.

The document **Typhoon Firmware Deliverables.doc** contains a very detailed description of the Flash image and how it is created.

5.1.1 Memory Layout Example



5.1.2 Magic Id

The first entry in the NV Device will be a validation Id to allow us to easily tell that the NV Device has been programmed. This does not guarantee that the images are valid. The Magic Id will be 8 bytes and contain the word **"Typhoon".**

5.1.3 Section Headers

This section following the Magic ID will contain a list of sections within the NV Device. Each section will contain:

1. Section ID, identifying the section (1 byte).
2. Section Pointer, to allow flexibility on where images are located (3 bytes).

If there is additional Header information that is required, it will be stored at the beginning of that section. *The Section Header section will not be changeable since it is read by routines in ROM.*

The first header in this section must ALWAYS be the Boot Image Header. This is a requirement of how the RomBoot loader code built into the Typhoon ASIC was designed.

The number of Sections is currently limited to 16, but can be increased what fits into one Flash page if needed.

Section ID's: - defined in **serial.h**

00	NULL_SECTION_ID	**Null section**. May be used as a terminator
01	EEBOOT_SECTION_ID	**Typhoon Boot Image Section**. MUST BE 1st HEADER
02	STATIC_SECTION_ID	**Static Data Section**. MAC Address, serial numbers etc
03	VAR_SECTION_ID	**Variable Data Section** Used for Non-volatile config params
04	SLEEP_SECTION_ID	**Sleep Image** for card bring up. Repeated for scattered load
05	NETBOOT_SECTION_ID	**NetBoot**. BIOS Extension ROM space for LanWorks
06	VAR_DEFAULT_SECTION_ID	**Variable Data Section** Used for Non-volatile configuration
07	VERSION_SECTION_ID	**Version** Contains ASCII date/version information
08	ASFINFO_SECTION_ID	**ASF/GPIO Tables**
09	DEBUG_SECTION_ID	**Debug Section.** Data kept across reboots for debugging
0a	LAST_SECTION_ID	**MUST BE LAST ID**

There is much more information on how the flash image is created and laid out in the document **Typhoon firmware deliverables.doc**

5.1.4 Static Section

This section will contain information regarding static data that can never be changed.
1. **Section Length,** the total length in bytes of the section (4 Bytes).
2. **Load Address**, not used.
3. **Section Checksum,** a simple checksum to validate the section (? Bytes).
4. **Section Body**, the contents of the section (? Bytes).

The following table summarizes the contents of static data.

Byte Offset (hex)	**Value**	**Field Description**
00	0000	3Com Node Address (word 0)
02	0000	3Com Node Address (word 1)
04	0000	3Com Node Address (word 2)
06	9903	DeviceID
08	0000	Manufacturing Data – Date (word 0)
0A	0000	Manufacturing Data – Date (word 1)
0C	0000	Manufacturing Data – Division
0E	0000	Manufacturing Data – Product Code
10	10B7	SubsystemVendorID
12	9903	SubsystemID
14	0000	PciParm0
16	0000	PciParm1
18	87F8	PciParm2
1A	DFFE	Capabilities Word
1C	0003	Media Options
1E	0101	Hardware Revision

3Com Node Address:
This is the 3Com node address for the adapter. This is not the address to be programmed as Station Address into the MAC. Refer OEM address below.

DeviceID:
This is the 2 byte product identifier assigned within 3Com, which gets loaded into the ASIC and made available in the DeviceID register of the PCI configuration space.

Manufacturing Data – Date

This is the date of manufacture, encoded according to the following,

Date (word 0)

year	[15:0]	

Date (word 1)

day	[4:0]	The day (1 through 31)
month	[8:5]	The number of the month (1 through 12)
	[15:9]	Reserved

Manufacturing Data - Division
This is the manufacturing division code, as shown on the product bar code label.

Manufacturing Data - Product Code
This is the manufacturing product code which is two alphanumeric ASCII characters from the bar code label.

Subsystem VendorID
This is the 2 byte subsystem VendorID. Since in this case the subsystem is 3Com adapter, we use 3Com's PCI VendorID, 10b7h.

SubsystemID
This is the 2 byte subsystem ID. For 3Com adapters, we use the same code as DeviceID.

PCIParm0
This is loaded into the ASIC and controls various hardware functions related to PCI bus operation.

MinGnt	[7:0]: Determines the value returned in bits [4:1] of the MinGnt register.
maxGnt	[15:8]: Determines the value returned in bits [5:0] of the MaxLat register.

PCIParm1

d3Hot	[0]: This bit determines the value for the bit 14 in PowerMgmtCap. This bit indicates the adapter's ability to generate power management events from the d3Hot state.
d3ColdPme	[1]:Provides the value returned in bit 15 of the PowerMgmtCap register. This bit, when set, indicates that the adapter is capable of signaling wakeup from the d3Cold state.
d1Support	[2]:Provides the value returned in d1Support and also bit 12 in the PowerMgmtCap register.
d2Support	[3]:Provides the value returned in d1Support and also bit 13 in the PowerMgmtCap register.
Aux_current	[8:6] Reports Typhoon's 3.3Vaux current requirements.

PCIParm2
These 2 bytes contains data defining the PCI capabilities of the NIC. The following table summarizes the PCI capabilities of Typhoon Adapter.

Bit	PciParm2 Bit	Value
0	*SupportsBustMaster*	1
1	*supports64bitPci*	0
2	*supports64bitMasterAddressing*	0
3	*supports64bitTargetAddressing*	0
4	*supports66MhzPci*	0
5	*SupportsMRM*	1
6	*SupportsMRL*	1
7	*SupportsMWI*	1
8	*SupportsPowerMgmt*	1
9	*SupportsWakeupPkts*	1
10	*SupportsMagicPkts*	1
11	*SupportsLinkEvents*	1
12	*SupportsHelloPkt*	1
13	Unused	0
14	Unused	0
15	Unused	0

supportsBusMaster	[0]: Indicates the adapter supports bus master data transfers.
supports64bitPci	[1]: Indicates that the NIC supports 64 bit PCI transactions.
supports64bitMasterAddressing	[2]: Indicates that the NIC supports 64 bit addressing and DAC as a Master.
supports64bitTargetAddressing	[3]: Indicates that the NIC supports 64 bit addressing and DAC as a Target.

supports66MhzPci	[4]: Indicates that the NIC supports 66Mhz PCI Interface.
supportsMRM	[5]: Indicates that the PCI interface supports the Memory Read Multiple PCI command.
supportsMRL	[6]: Indicates that the PCI interface supports the Memory Read Line PCI command.
supportsMWI	[7]: Indicates that the PCI interface supports the Memory Write and Invalidate PCI command.
supportsPowerMgmt	[8]: Indicates that the adapter supports the OnNove/ACPI power management scheme.
supportsWakeUpPkts	[9]: Indicates that the NIC supports WakeUp packets as part of Wake On LAN functionality.
supportsMagicPkts	[10]: Indicates that the NIC supports the Power Management WakeUp Events based on reception of Magic Packets.
supportsLinkEvent	[11]: Indicates that the NIC supports the Power Management WakeUp Events based on changes in the link state.
supportsHelloPkt	[12]: Indicates that the NIC supports transmission of Hello Packets in standby/sleep mode.
Unused	[13-15]: Reserved for future use.

Capabilities Word
This field contains data defining the basic capabilities of the adapters.

Bit	Capabilities Bit	Value
0	*supportsPlugNPlay*	1
1	*supportsJumboFrames*	1
2	*supportsTxFlowControl*	0
3	*supportsRxFlowControl*	1
4	*supportsVLanTagging*	1
5	*SupportsIpChecksumInsertion*	1
6	*SupportsTcpChecksu Insertion*	1
7	*SupportsUdpChecksumInsertion*	0
8	*SupportsIpChecksumVerification*	1
9	*SupportsTcpChecksumVerfication*	1
10	*SupportsUdpChecksumVerfication*	0
11	*supportsTxPriority*	1
12	*supportsRxPriority*	1
13	*SupportsHalfDuplex*	1
14	*supportsFull Duplex*	1
15	*supportsNetBoot*	0

SupportsPlugNPlay	[0]: Indicates that Microsoft Oses recognize our adapter and will automatically load the appropriate driver.
SupportsJumboFrames	[1]: Indicates that the NIC supports packet size over 1514 bytes.

SupportsTxFlowControl	[2]: Indicates that the NIC supports transmitting Flow Control packets as per 802.3x.
SupportsRxFlowControl	[3]: Indicates that the NIC supports reception of Flow Control packets as per 802.3x.
SupportsVLanTagging	[4]: Indicates that the NIC supports VLAN tagging as per 802.3ac and 802.1q.
SupportsIpChecksumInsertion	[5]: Indicates that the NIC supports IP Checksum insertion.
SupportsTcpChecksumInsertion	[6]: Indicates that the NIC supports TCP Checksum insertion.
SupportsUdpChecksumInsertion	[7]: Indicates that the NIC supports UDP Checksum insertion.
SupportsIpChecksumVerification	[8]: Indicates that the NIC supports IP Checksum verification.
SupportsIpChecksumVerification	[9]: Indicates that the NIC supports TCP Checksum verification.
SupportsIpChecksumVerification	[10]: Indicates that the NIC supports UDP Checksum verification.
SupportsTxPriority	[11]: Indicates that the NIC supports Tx priority queues.
SupportsRxPriority	[12]: Indicates that the NIC supports Rx priority queues.
supportsHalfDuplex	[13]: Indicates the NIC supports half-duplex media operation.
supportsFullDuplex	[14]: Indicates the NIC supports full-duplex media operation.
SupportsNetBoot	[15]: Indicates the NIC supports NetBoot operation.

MediaOptions
Indicates physical media connections available in the adapter. A value of zero indicates that connection is not available and a value of one indicates that the connection is available.

XcvrAvailable	[0]:	100Base-Tx
	[1]:	10 Base-T
	[15:2]:	Reserved

HardwareRevision
Indicates the ASIC and board level revisions.

AsicLevel	[7:0]	Indicates the ASIC revision level	
		[3:0]	Major ASIC revision
		[7:4]	Minor ASIC revision
BoardLevel	[15:8]	Indicates board revision level	
		[15:12]	Major board revision
		[11:8]	Minor board revision

5.1.5 VAR Section

This section will contain non-volatile variables and user set-able information that must be saved across reset.

1. **Section Length,** the total length in bytes of the section (4 Bytes).

2. **Load Address**, not used.
3. **Section Checksum,** a simple checksum to validate the section (? Bytes).
4. **Section Body**, the contents of the section (? Bytes).

The following table summarizes the contents of the VAR data

Byte Offset (hex)	**Value**	**Field Description**
00	0000	RomInfo
02	0000	OEM Node Address (word 0)
04	0000	OEM Node Address (word 1)
06	0000	OEM Node Address (word 2)
08	0100	XcvrSelect
0A	0001	Software Information1
0C	0000	Software Information2
0E	0000	Software Information3
10	0000	Lanworks Data0
12	0000	Lanworks Data1
14	0000	Override Node Address (word 0)
16	0000	Override Node Address (word 1)
18	0000	Override Node Address (word 2)
1A	0000	XcvrOption
1C	0005	SleepEvents
1E	0000	SleepTimerInterval0
20	0000	SleepTimerInterval1
22	0000	SOSOption
24	0000	SOSIPAddress0 (word 0)
26	0000	SOSIPAddress1 (word 1)
28	0000	SOSSubnetMask0 (word 0)
2A	0000	SOSSubnetMask1 (word 1)
2C	0000	SOSGateway0 (word 0)
2E	0000	SOSGateway1 (word 1)
30	C600	SMBus address

Data Field Details

ROMInfo
This conveys to a driver or configuration program whether a NetBoot Code is included in the NVDevice and whether the system must be loaded remotely from a Server.

NetBootEnable [0]: Enables the Expansion ROM for NetBoot.
0 – enabled
1 – disabled

OEM Node Address
This is the node address to be programmed into the MAC address register. For 3Com adapters, this field contains the same value as in 3COM Node Address. OEM customers may choose to program this field with a different value.

XcvrSelect

xcvrSelect	[15:0]:This read/write field indicates the selected transceiver type:		
	[7:0]	*XcrvType*	
		00000000	– 100Base-TX
		00000001	– 10Base-T
		00000010 – 11111111	– Reserved
	[8]	*AutoSelect*	
		0 – select transceiver by *XcvrType*	
		1 – auto-negotiation	
	[15:9]	Reserved	

Software Information1
This field contains environmental information for use by the driver.

OptimizeFor [1:0]:Specifies the environment for which to optimize.
00: Unused
01: Normal
02: Maximum network Performance
03: Minimum CPU Utilization

Software Information 2
Additional information for drivers for future use.

Software Information 3
Additional information for drivers for future use.

Lanworks Data0, Lanworks Data1
These words are reserved for use by Lanworks, to hold data related to the Expansion ROM access.

Override Node Address
This is the node address given by the user using a configuration file like ‘net.cfg or installation utility to be programmed into the MAC address register. This value needs to be stored between loading of sleep and runtime images.

XcvrOption
Transceiver options available.

DuplexMode [0]: Selects duplex mode
0 – half duplex
1 – full duplex

SleepEvents
Enable/disable sleep events. Events set to ‘0’ are disabled, events set to ‘1’ are enabled.
SleepEvents [0]: MagicPacket
[1]: LinkEvent
[2]: Wake-On-Error
[3:15]: Reserved

SleepTimerInterval
Interval for Wake-on-Timer (DHCP alarm clock)

SOSOption
Enable/disable SOS traps. ‘0’ is disabled, ‘1’ is enabled.

	[0]: Sos0
	[1]: Sos1
	[2]: Sos2
	[3:15]: Reserved

SOSIPAddress
Destination IP Address for outgoing SOS traps.

SOSSubnetMask
SubnetMask for outgoing SOS traps.

SOSGatewayAddress
Gateway Address for outgoing SOS traps.

SMBusAddress
Address used for SMBus transfers.

SMBusAddress	[0:7]: Reserved
	[15:8]: SMBus Address

5.1.6 Boot Image Section

This section will contain a binary Typhoon boot image. The entry point for execution is always at the start of the section.

1. Section Length - the total length in bytes of the section (4 Bytes).
2. Section Load Address - the location to load this section into memory (4 Bytes).
3. Section Checksum - a simple checksum to validate the image (1 Byte).
4. Header Checksum - a simple checksum to validate the headers (1 Byte).
4. Word alignment pad, a simple checksum to validate the image (2 Bytes).
5. Section Body - the contents of the section (Targeted to be less than 8KBytes).

5.1.7 Sleep Image Section

Sleep image is the code loaded from NVDevice on initial Power-on. Section header of the sleep image consists of the following. There can be more than one sleep image section for scattered loading of the image into SRAM and IRAM as defined in the load address of each sleep section headers.

1. **Section Length**, the total length in bytes of the section (4 Bytes).
2. **Section Load Address,** the location to load this section into memory (4 Bytes).
3. **Section Checksum**, a simple checksum to validate the image (4 Bytes).
4. **Section Pointer,** pointer to the next section if available, or else null (1 Pad Byte + 3 Bytes)
5. **Section Body**, the contents of the section (Targeted to be less than 8KBytes).

5.2 Diagnostics

Various diagnostic capabilities will be available for the Adapter. If the NV-Device on the Adapter is not programmed the status will be indicated in a general purpose register at power up. The Host will then have NV-Device read/write capability in order to program and verify the NV-Device. The Host will have direct access to the NV-Device through the SPI registers. The boot loader will support a command set allowing the Host to load the NV-Device (by indicating address/data) and access memory.

If the NV-Device is programmed then the Adapter will load the Boot Image from the NV-Device to IRAM and then transfer execution to IRAM. POST will perform tests on the internal MAC, PCI registers, external SRAM, Sidewinder, NV-Device, PHY, etc. and save the result to a [TBD] location in SRAM and indicate the result to a general purpose I/O register. After successful completion of POST the Adapter will copy the run-time image from the NVDevice and execute the run-time image. At this time the Host could run the following tests:

[TBD] The tests are currently being identified, however, an indication of tests currently run on previous products is given below:

- Register access test
- BIST [TBD] - (may report the result from POST or run a subset of the POST tests) External SRAM test, Sidewinder verification, etc.
- Program the NV-Device – this will require a [TBD] test mode, allows read/write of each location
- Verify the NV-Device checksum(s) – calculate checksum and compare against the stored value. There will be [TBD] a checksum for each image in the NV-Device
- Register access test – need to identify all registers which allow each bit to be set/cleared and read back
- FIFO Loopback test – need to define a mode which allows transmit data to be moved into the receive buffer instead of being transmitted
- [TBD] Loopback tests will need to be incorporated which allow the data to be encrypted/decrypted by the Sidewinder
- Internal Loopback test – data is looped in the MAC
- MII Read/Write test
- ENC/DEC Loopback test
- 10Mbs Echo Test – (over 100 meter cable? - find an echo server, create an echo server request packet, transmit the packet, wait for the packet to return from the echo server, save the echo server's address and then send 100 echo packets and compare each received echo packet with the packet sent)
- 100Mbs Echo Test
- Interrupt test [TBD]
- Remote wake-up test [TBD]
- IIC Test [TBD]

Appendix A – Enumerated Values

1. Host to Adapter Commands

The following data structures are taken from the Typhoon header files **cmd.h** on June 7, 2000 that define the command enumeration. The names do not exactly match the commands described earlier, but they are close enough to figure them out.

Commands

```
CMD_reset =                 0,
CMD_tx_enable =             1,
CMD_tx_disable =            2,
CMD_rx_enable =             3,
CMD_rx_disable =            4,
CMD_rx_filter_write =       5,
CMD_rx_filter_read =        6,
CMD_stats_read =            7,
CMD_stats_cycle =           8,
CMD_stats_clear =           9,
CMD_memory_read =          10,
CMD_memory_write =         11,
CMD_var_section_read     =  12,
CMD_var_section_write    =  13,
CMD_static_section_read  =  14,
CMD_static_section_write =  15,
CMD_image_section_prgm   =  16,
CMD_nvram_page_read      =  17,
CMD_nvram_page_write     =  18,
CMD_xcvrsel  =             19,
CMD_testmux =              20,
CMD_phyloopback_enable =   21,
CMD_phyloopback_disable =  22,
CMD_mac_cntrl_read =       23,
CMD_mac_cntrl_write =      24,
CMD_max_pkt_size_read =    25,
CMD_max_pkt_size_write =   26,
CMD_media_status_read =    27,
CMD_media_status_write =   28,
CMD_network_diags_read =   29,
CMD_network_diags_write =  30,
CMD_physical_mngt_read =   31,
CMD_physical_mngt_write =  32,
CMD_var_param_read =       33,
CMD_var_param_write =      34,
CMD_goto_sleep_l =         35,
CMD_firewall_control =     36,
CMD_mcast_hash_mask_read = 37,
CMD_station_addr_write =   38,
CMD_station_addr_read =    39,
CMD_station_mask_write =   40,
CMD_station_mask_read =    41,
CMD_vlan_ether_type_read = 42,
CMD_vlan_ether_type_write=  43,
CMD_vlan_mask_read =        44,
CMD_vlan_mask_write =       45,
CMD_bcast_throttle_write =  46,
CMD_bcast_throttle_read  =  47,
CMD_dhcp_prevent_write   =  48,
CMD_dhcp_prevent_read    =  49,
CMD_recv_buffer_control  =  50,
CMD_software_reset =        51,
CMD_create_sa =             52,
CMD_delete_sa =             53,
CMD_diag_test_complete =    54,
CMD_diag_arm2hostcomm =     55,
CMD_diag_pcidmadoneint =    56,
CMD_diag_pingpong =         57,
CMD_diag_host2armcomm =     58,
CMD_diag_pcidmaint =        59,
CMD_diag_rxdmadoneint =     60,
CMD_diag_txdmadoneint =     61,
CMD_diag_swdmadoneint =     62,
CMD_diag_swint =            63,
CMD_diag_mactxcomplete =    64,
CMD_diag_oneshot =          65,
CMD_diag_freetimer =        66,
CMD_version_read =          67,
CMD_transmit_waveform =     68,
CMD_filter_define =         69,
CMD_add_wakeup_pkt =        70,
CMD_add_sleep_pkt =         71,
CMD_enable_sleep_events =   72,
CMD_enable_wakeup_events =  73,
CMD_get_ip_address =        74,
CMD_tcpSegmentResponse =    75,
CMD_read_pci_reg_l =        76,
CMD_write_pci_reg_l =      77,
CMD_read_offload_capability_l = 78,
CMD_write_offload_l =       79,
CMD_LM75_read_temp =        80,
CMD_smb_host_receive =      81,
CMD_smb_send =              82,
CMD_smb_stop =              83,
CMD_sos_read =              84,
CMD_get_linkStatus_l =      85,
CMD_mfg_wol =               86,
CMD_hello_response =        87,
CMD_enable_rx_filter =      88,
CMD_rx_filter_capability =  89,
CMD_pwr_dbg_log_read =      90,
CMD_pwr_dbg_log_reset =     91,
CMD_pwr_dbg_log_index =     92,
CMD_halt =                  93,
CMD_read_ipsec_info_l =     94,
CMD_reserved2 =             95,
CMD_reserved3 =             96,
CMD_reserved4 =             97,
CMD_reserved5 =             98,
CMD_reserved6 =             99,
CMD_adapter_data_write =    100,            // novel ECB only
CMD_multicast_addr_write =  101,            // novel ECB only
CMD_asf_command =           102,

CMD_get_ipsec_enable =      103,
CMD_snipit_download =       104,

CMD_Last_Value                // always must be the last value
                              // determines the size of the table
```

Unsolicited Responses

CycleStatistics = 8
MediaStatusRead = 27
TestComplete = 54
TcpSegmentResponse = 75

2. Typhoon Status

These values are written to the ARM2Host register 0 by the Typhoon to indicate the status of the software.

```
#define STATUS_ROM_CODE                          0x00000001
#define STATUS_ROM_EEPROM_LOAD                   0x00000002
#define STATUS_ROM_FAIL                          0x00008003
#define STATUS_EEPROM_CODE                       0x00000004
#define STATUS_EEPROM_FAIL                       0x00008005
#define STATUS_FIRST_INIT                        0x00000006
#define STATUS_WAITING_FOR_BOOT                  0x00000007
#define STATUS_SECOND_INIT                       0x00000008
#define STATUS_RUNNING                           0x00000009
#define STATUS_WAITING_FOR_HOST_REQ              0x0000000D
#define STATUS_FAIL                              0x0000800A
#define STATUS_PCI_TARGET_ABORT                  0x0000800B
#define STATUS_PCI_MASTER_ABORT                  0x0000800C
#define STATUS_POST_FAIL                         0x0000800E
#define STATUS_DIAGNOSTIC_MODE                   0x0000000F
#define STATUS_WAITING_FOR_SEGMENT               0x00000010
#define STATUS_SEGMENT_FAIL                      0x00008010
#define STATUS_SLEEPING                          0x00000011
#define STATUS_RESETTING                         0x00000012
#define STATUS_LANWORKS_CMD_COMPLETE             0x00000013
#define STATUS_LANWORKS_CMD_FAIL                 0x00008013
#define STATUS_HALTED                            0x00000014
#define STATUS_HALTING                           0x00000015
```

3. Host Status

These values are written to the Host2ARM register 0 by the driver to pass boot commands prior to the transfer of the boot record to the adapter.

```
#define HOST_CMD_BOOTING                  0x00000000
#define HOST_CMD_NULL                     0x00000000
#define HOST_CMD_BOOT_RECORD              0x000000FF
#define HOST_CMD_NVDEVICE_PROGRAM         0x000000FE
#define HOST_CMD_RUNTIME_IMAGE            0x000000FD
#define HOST_CMD_SEGMENT_AVAILABLE        0x000000FC
#define HOST_CMD_DOWNLOAD_COMPLETE        0x000000FB
```

```
#define HOST_CMD_WAKEUP                    0x000000FA
```

Appendix B - Packet Filtering

NOTE: For the first release, due to the memory size limitation on the adapter, the Packet Filter engine of Typhoon will reside with the NDIS driver on the host. It is the same engine that performs the Ipsec Selector matching against the Security Association Database. Filtering policy can be enforced by creating a special kind of Security Association with extension data to handle the filtering requirements.

1. Filter features

Typhoon provides a sophisticated set of security features, based on which a powerful high-level personal firewall may be efficiently supported. The focus of Typhoon's security features is to offload CPU-intensive filtering functions that are best implemented on the intelligent adapter card.

Typhoon supports the following services:

1) Dynamic specification of IP packet filters.
2) Capturing and reporting of specified packets.
3) Support for Standby (temporary suspension of all packet filters) and Blocking (temporary shutoff of all network traffic) mode.
4) Address translation (through OEM Extension)
5) Encryption/Decryption (through IPSec)

Due to its on-board processing power and strategic position on the protocol stack, Typhoon offers excellent filtering performance and is particular effective in protection against Denial of Service attack. However, high-level security protocols, such as Authentication, Key Management, HTTP URL filters, etc., must be implemented on the host.

1.1 Filtering Management Requests to Typhoon

Higher layer security protocol issues Typhoon requests to ask for supported security features.
Handling of packet filtering management requests has lower processing priority than the normal data packet.

The following types of request are supported:

REQUEST_BLOCK_ALL
This request asks for a temporary blocking of all IP traffic.

REQUEST_UNBLOCK
This request cancels the previously issued REQUEST_BLOCK_ALL.

REQUEST_FILTER_STANDBY
This request temporarily asks for a temporary suspension of all packet filters and allows all traffic to pass through the adapter.

REQUEST_FILTER_UNSTANDBY
This request cancels the previously issued REQUEST_FILTER_UNSTANDBY.

REQUEST_SET_FILTER_DEFAULT
This request sets default actions regarding the handling of packets not covered by any packet filters. This request specifies the direction of packet flow (transmit/receive), actions (pass-through/drop), and reporting options (report to the host or not).

REQUEST_FILTER_RESET_ALL
This request resets all filters previously defined. This is to prepare for defining a new round of filter scheme.

REQUEST_DEFINE_FILTER
This request specifies a packet profile and defines the action to be taken when a qualified packet is identified. A packet filter is defined by specifying:

1) IP Protocol
2) IP Source Address (defined as a single IP Address, a range of IP Addresses, any IP Address, or defined to match a Net Mask).
3) IP Destination Address (defined as a single IP Address, a range of IP Addresses, any IP Address, or defined to match a Net Mask).
4) TCP Source Port
5) TCP Destination Port
6) Direction of filtered traffic (Transmit/Receive)
7) Action to take (Discard, Pass)
8) Reporting option (to report or not)

Note that filter defined here does not bear an ID. In order to redefine a filter, high level protocol needs to issue REQUEST_FILTER_RESET first and start out fresh.

Reporting of packets identified and requested by a filter are passed on to the host with the following response:

RESPONSE_FILTER_REPORT
The actual packet is part of the response.

See sections on IPSec and OEM Extension for additional security support.

Appendix C - OEM function extension API

1. Overview

The first release of the adapter will include embedded software to perform all of the basic features described in this document. One of the benefits of the intelligent Typhoon architecture is the ability to extend its functions by adding new software. To allow simple extensibility an API is defined

2. Goals

The goal of the API is to provide an interface to allow new embedded Typhoon software to be integrated into the existing core software. The API is not intended for the inexperienced, it is assumed that anyone programming to the API has a firm understanding of the Typhoon architecture and a knowledge of the core Typhoon software.

The API:

- Allows new software to be inserted into the transmit and receive data paths. This can filter, modify, prioritize, and discard packets at multiple points in a packets transition through the adapter.
- Allows new host interfaces and commands to be implemented.
- Allows access to the hardware to modify or implement new features, e.g. I^2C, encryption, compression.
- Allows background tasks to be added to perform non data path functions.

3. SDK

The API is accessed through a 3Com supplied Typhoon SDK. The SDK contains documentation on the Typhoon hardware and software architecture, API description, tool requirements and a binary software image of the core software.

The core binary is statically linked with user code that interfaces to it via the API. The resulting image is downloaded to the adapter in place of the standard 3Com supplied image.

Appendix D: Sleep Event Implementation

1. Introduction

This appendix covers the details of how the sleep events are implemented within the firmware. The simple events whose descriptions can be found in any introductory networking book will not be covered here. Only more the complex or obscure events, or those that take advantage of certain loopholes in the original specifications will be explained here.

2. Novell Watchdog Replies

Novell uses NCP Watchdog Frames to keep a connection active. The server polls each client station. The poll interval is determined by the server, and is not included as part of poll frame. There is a poll frame from each server the station is connected to. The poll frames are addressed directly to each individual station by MAC address.

The fields in bold can be used to filter the frame.

<u>Server Poll Station</u>

Destination Address	6 bytes	Field A
Source Address	6 bytes	Field B
Length Field	2 bytes	00h 20h
IPX Checksum	**2 bytes**	**FFh FFh**
IPX Length	2 bytes	00h 20h
IPX Transport Control	1 byte	Hop Count
IPX Packet Type	**1 byte**	**04h (IPX)**
IPX Destination Address		
Destination Network	4 bytes	Field C
Node Address	6 bytes	Field D
Socket Number	**2 bytes**	**40h 05h**
IPX Source Address		
Source Network	4 bytes	Field E
Node Address	6 bytes	Field F
Socket Number	**2 bytes**	**40h 01h**
NCP Connection Number	1 byte	Field G
NCP Signature	**1 byte**	**3Fh**

The reply to the poll looks like (with the fields from the request marked)

Destination Address	6 bytes	Field B from Request

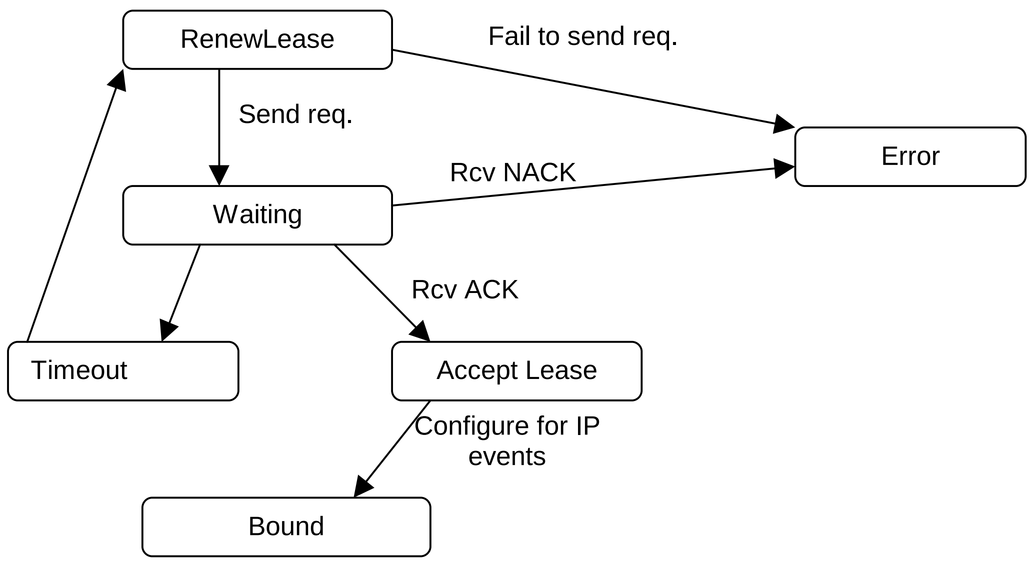
Source Address	6 bytes	Field A from Request
Length Field	2 bytes	00h 20h
IPX Checksum	2 bytes	FFh FFh
IPX Length	2 bytes	00h 20h
IPX Transport Control	1 byte	00h
IPX Packet Type	1 byte	04h
IPX Destination Address		
Destination Network	4 bytes	Field E from Request
Node Address	6 bytes	Field F from Request
Socket Number	2 bytes	40h 01h
IPX Source Address		
Destination Address	4 bytes	Field C from Request
Node Address	6 bytes	Field D from Request
Socket Number	2 bytes	40h 05h
NCP Connection Number	1 byte	Field G from Request
NCP Connection Active	1 byte	59h

3. DHCP Lease Renewal

When the Typhoon is put to into a low power state, it will start the DHCP state machine in the sleep image.

The possible DHCP states are:

- RENEW_LEASE - issuing a DHCPREQUEST message
- WAITING_FOR_OFFER - waiting for a server to respond with DHCPACK or DHCPNACK
- TIMEOUT - didn't get a server response in a reasonable time
- ACCEPT_OFFER - received a DHCPACK, resetting lease time, saving parameters
- LEASE_ERROR - any error in the renewal process lands us here. Abort lease renewal and abort any DHCP dependent features such as Wake-on-Ping.
- BOUND - not currently in the lease process. Also an indicator that the IP address stored internally is valid and may be used by other sections of the firmware.



Typhoon DHCP States

The Typhoon only checks for incoming DHCP messages while in the WAITING_FOR_OFFER state. There are no unsolicited DHCP messages, so it need only look a packet is expected.

The request packet sent out in the RENEW_LEASE state consists of a standard DHCPREQUEST packet with the “Client ID(61)” and “Request Parameter(55)” options. The IP address and Client ID used in the request are passed down by the driver with the EnableSleepEvents command. The Client ID is operating system dependent.

Typhoon expects to get back a DHCPACK packet with the “Lease Time(51)”, “SubnetMask(1)” and “BroadcastAddr(28)” options. Other options may be included in the DHCPACK message, but Typhoon will ignore them.

See RFCs 2131 and 2132 for details on the DHCP lease renewal process and for the formats of a DHCPREQUEST and a DHCPACK packet with options. Typhoon implements most of the "Re-using a previously allocated network address" algorithm in section 3.2 of RFC 2131.

Appendix E: SOS Pins on Typhoon

Typhoon has three general purpose input pins, SOS0, SOS1, SOS2. Each of these pins has a pull-up on the board, if left unconnected the default state is high. The firmware implementation is for engineering tests purposes, and would require modification if Typhoon had an OEM customer.

The SOS pins are currently polled in sleep mode only and polling is only done if the SOSOption parameter in the variable section indicates that polling is enabled. In order to individually test these pins they have been given the following assignments.

SOS0 – BIOS didn't complete
SOS1 – OS didn't boot
SOS3 – Case Intrusion

Traps are generated in sleep mode, provided that the polling has been enabled and the pin has been asserted low for a specified amount of time. If the SOS0 or SOS1 pins are asserted low for at least five minutes, a trap for the corresponding pin will be generated. A trap for the SOS3 pin will be generated if the pin has been asserted low for at least two seconds. None of the traps will be generated if the runtime image is downloaded before the timer expires.

The traps are generated with the following format detailed below. For test purposes the trap is currently sent as a broadcast frame. The variable section contains parameters for SOS destination IPAddress, SubnetMask, and Gateway (these parameters can be configured by Touchdown) which can be used if an implementation is done for an OEM.

For information on the trap format refer to the document *Frame Formats For Platform Event Traps,* V 0.94, by Carl First.

<u>SOS0 (BIOS didn't complete)</u>

*Ethernet Header

Destination Address	6 bytes	Broadcast
Source Address	6 bytes	Typhoon MAC address
Type Field	2 bytes	0x0800

*IP Header

Headerlen/Version	1 byte	0x45
TOS	1 byte	0x00
Total length	2 bytes	0x8B
Identification	2 bytes	0x0000
Flags/Fragment offset	2 bytes	0x0000
Time to Live	1 byte	0x80

Protocol	1 byte	0x11
IP Checksum	2 bytes	0xXXXX
Source IP	4 bytes	0xXXXXXXXX
Destination IP	4 bytes	0xXXXXXXXX

*UDP Header

Source Port	2 bytes	0x00A2
Destination Port	2 bytes	0x00A2
Length	2 bytes	0x0077
UDP Checksum	2 bytes	0xXXXX

*SNMP Trap PDU

Message type	2 bytes	0x306D
Version number	3 bytes	0x020100
Community String	8 bytes	0x0406"public"
PDU Trap Type	2 bytes	0xA460
Enterprise	11 bytes	0x06092b06010401986f0101
Agent address	6 bytes	0x4004XXXXXXXX
Generic trap	3 bytes	0x020106
Specific trap	5 bytes	0x0203236F00
Time stamp	6 bytes	0x430400000000
Sequence of	4 bytes	0x303F303D
OID	12 bytes	0x060a2b06010401986f010101
GUID	16 bytes	0x00000000000000000000000000000000
Sequence #	2 bytes	0x0000
Local Timestamp	4 bytes	0x00000000
UTC Offset	2 bytes	0xffff
Trap Source type	1 byte	0x50 (NIC)
Event Source type	1 byte	0x50 (NIC)
Event Severity	1 byte	0x10
Sensor type	1 byte	0xff
Sensor number	1 byte	0xff
Entity ID	1 byte	0x23(BIOS)
Entity Instance	1 byte	0x00
Event Data 1	1 byte	0xC0
Event Data 2	1 byte	0x02
Event Data 3	1 byte	0xff
Event Data 4	1 byte	0x00
Event Data 5	1 byte	0x00
Event Data 6	1 byte	0x00
Event Data 7	1 byte	0x00
Event Data 8	1 byte	0x00
Language Code	1 byte	0xff
Manufacture ID	4 bytes	0x00000000
System ID	2 bytes	0x0000
OEM Custom	1 byte	0xC1

<u>SOS1 (OS didn't boot)</u>

*Ethernet Header

Destination Address	6 bytes	Broadcast
Source Address	6 bytes	Typhoon MAC address

Type Field	2 bytes	0x0800

*IP Header

Headerlen/Version	1 byte	0x45
TOS	1 byte	0x00
Total length	2 bytes	0x8B
Identification	2 bytes	0x0000
Flags/Fragment offset	2 bytes	0x0000
Time to Live	1 byte	0x80
Protocol	1 byte	0x11
IP Checksum	2 bytes	0xXXXX
Source IP	4 bytes	0xXXXXXXXX
Destination IP	4 bytes	0xXXXXXXXX

*UDP Header

Source Port	2 bytes	0x00A2
Destination Port	2 bytes	0x00A2
Length	2 bytes	0x0077
UDP Checksum	2 bytes	0xXXXX

*SNMP Trap PDU

Message type	2 bytes	0x306D
Version number	3 bytes	0x020100
Community String	8 bytes	0x0406"public"
PDU Trap Type	2 bytes	0xA460
Enterprise	11 bytes	0x06092b06010401986f0101
Agent address	6 bytes	0x4004XXXXXXXX
Generic trap	3 bytes	0x020106
Specific trap	5 bytes	0x0203236F00
Time stamp	6 bytes	0x430400000000
Sequence of	4 bytes	0x303F303D
OID	12 bytes	0x060a2b06010401986f010101
GUID	16 bytes	0x00000000000000000000000000000000
Sequence #	2 bytes	0x0000
Local Timestamp	4 bytes	0x00000000
UTC Offset	2 bytes	0xffff
Trap Source type	1 byte	0x50 (NIC)
Event Source type	1 byte	0x50 (NIC)
Event Severity	1 byte	0x10
Sensor type	1 byte	0xff
Sensor number	1 byte	0xff
Entity ID	1 byte	0x24(OS)
Entity Instance	1 byte	0x00
Event Data 1	1 byte	0xC0
Event Data 2	1 byte	0x03
Event Data 3	1 byte	0xff
Event Data 4	1 byte	0x00
Event Data 5	1 byte	0x00
Event Data 6	1 byte	0x00
Event Data 7	1 byte	0x00
Event Data 8	1 byte	0x00
Language Code	1 byte	0xff
Manufacture ID	4 bytes	0x00000000

System ID	2 bytes	0x0000
OEM Custom	1 byte	0xC1

SOS2 (Case Intrusion)

*Ethernet Header

Destination Address	6 bytes	Broadcast
Source Address	6 bytes	Typhoon MAC address
Type Field	2 bytes	0x0800

*IP Header

Headerlen/Version	1 byte	0x45
TOS	1 byte	0x00
Total length	2 bytes	0x8B
Identification	2 bytes	0x0000
Flags/Fragment offset	2 bytes	0x0000
Time to Live	1 byte	0x80
Protocol	1 byte	0x11
IP Checksum	2 bytes	0xXXXX
Source IP	4 bytes	0xXXXXXXXX
Destination IP	4 bytes	0xXXXXXXXX

*UDP Header

Source Port	2 bytes	0x00A2
Destination Port	2 bytes	0x00A2
Length	2 bytes	0x0077
UDP Checksum	2 bytes	0xXXXX

*SNMP Trap PDU

Message type	2 bytes	0x306D
Version number	3 bytes	0x020100
Community String	8 bytes	0x0406"public"
PDU Trap Type	2 bytes	0xA460
Enterprise	11 bytes	0x06092b06010401986f0101
Agent address	6 bytes	0x4004XXXXXXXX
Generic trap	3 bytes	0x020106
Specific trap	5 bytes	0x0203056F00
Time stamp	6 bytes	0x430400000000
Sequence of	4 bytes	0x303F303D
OID	12 bytes	0x060a2b06010401986f010101
GUID	16 bytes	0x00000000000000000000000000000000
Sequence #	2 bytes	0x0000
Local Timestamp	4 bytes	0x00000000
UTC Offset	2 bytes	0xffff
Trap Source type	1 byte	0x50 (NIC)
Event Source type	1 byte	0x50 (NIC)
Event Severity	1 byte	0x10
Sensor type	1 byte	0xff
Sensor number	1 byte	0xff
Entity ID	1 byte	0x18(CASE)
Entity Instance	1 byte	0x00

Event Data 1	1 byte	0x00
Event Data 2	1 byte	0xff
Event Data 3	1 byte	0xff
Event Data 4	1 byte	0x00
Event Data 5	1 byte	0x00
Event Data 6	1 byte	0x00
Event Data 7	1 byte	0x00
Event Data 8	1 byte	0x00
Language Code	1 byte	0xff
Manufacture ID	4 bytes	0x00000000
System ID	2 bytes	0x0000
OEM Custom	1 byte	0xC1

4. Index