NITROX III CN35xx Processor Family Microcode Instruction Set Manual

IPSec/IKE Microcode

Main Microcode: CNN35x-MC-IPSEC-0002
Boot Microcode: CN35xx-MC-BOOT-0001

Document Number: CNN35xx-MC-IPSec-0002-001.pdf
August, 2012

About This Document

This document describes and specifies the features and instruction set (IS) provided by CN35xx IPSec/IKE microcode file CNN35x -MC-IPSEC-0002, where 0002 represents the microcode version number. This document also specifies the features and instruction set for the associated N3 boot microcode file CN35xx-MC-BOOT-0001.

The intended audience for this document is hardware engineers, software engineers and managers tasked with implementing software designs using the N3 macro-processor.

PUBLISHED BY
Cavium, Inc
2315 N. First Street,
San Jose, CA 95131
Phone: 408-943-7100
Email: sales@cavium.com
Web: http://www.cavium.com

Chapter Overviews

Chapter 1 IPSec/IKE Feature Set: lists and describes the feature set implemented by the specific microcode version(s) listed above.

Chapter 2 NITROX III Microcode General Overview: Reviews the general structure of a NITROX III instruction, and describes the general instruction execution flow.

Chapter 3 NITROX III Microcode Loading Process: Describes what “Boot” Microcode is, what “Main” microcode is, describes the layout of a microcode binary file, and describes the microcode loading and initialization process.

Chapter 4 IPSec/IKE Microcode Users Guide: A primer describing the intended usage of NITROX IPSec/IKE instructions, to implement a typical IPSec/IKE application.

Chapter 5 Boot Microcode Instruction Set: A detailed reference description of the NITROX III Boot Microcode instruction set. These instructions can be used to load the IPSec/IKE microcode and various runtime data structures into the NITROX III chip during device initialization. This instruction set may also provide various device debug instructions that may be used during system development, bring up, and/or troubleshooting.

Chapter 6 IPsec/IKE Microcode Instruction Set: A detailed reference description of the NITROX III IPSec/IKE microcode instruction set. These instructions can be used to accelerate a software implementation of the IPsec and IKE protocols.

Appendix A Instruction Set Quick Reference: list of all instructions (and associated opcodes) implemented by the specific microcode versions listed above.

Appendix B Completion Code Quick Reference: A quick reference table of all completion codes returned by this microcode API set.

Appendix C Microcode Change History: A brief review of feature changes since previous microcode release.

Prerequisites

This document assumes that the reader has a good understanding of the following:

- NITROX III chip hardware architecture, and typical usage (see NITROX III Hardware Manuals)
- IPsec and IKE security protocols
- SRTP protocol
- Asymmetric and symmetric cryptography

Document Revision History

Below is a history of the creation and revision of this document. It is not a history of microcode version revisions. That history can be found in the Appendix of this document.

Revision	Date	Author	Change Description
	10/14/11	Dhana	Initial Release created from Document Number: CNN35xx-MC-IPSec-0001-001.pdf
001	8/25/12	Mike S	Basic editing to clean up document formatting Updated Scatter/Gather Mode Description Added Appendix listing SHA HMAC Initial Values

Table of Contents

About This Document 2
Chapter Overviews 2
Prerequisites 3
Document Revision History 3
Table of Contents 4
Chapter 1 IPSec/IKE Feature Set 6
Chapter 2 NITROX III Microcode General Overview 9
2.1 NITROX III Instruction/Data Flow 9
2.2 Basic Instruction Format 10
Instruction Format Rules: 10
2.3 DMA Modes 11
Direct DMA Mode (Opcode<7>==0) 12
Scatter/Gather DMA Mode (Opcode<7>==1) 12
2.4 Error Address/Error Codes 14
2.5 Typical NITROX III Instruction Execution Flow 15
In-Place Processing 16
Chapter 3 NITROX III Microcode Loading Process 17
3.1 Microcode Loading Fundamentals 17
3.2 Microcode Binary File Layout 18
3.3 Microcode Initialization 19
3.4 Boot Initialization Parameters 19
3.5 Debug Instructions 20
Chapter 4 IPSec/IKE Microcode Users Guide 21
4.1 Basic API Notation 21
4.2 Internet Key Exchange (IKE) 21
IKE Notation 21
4.3 IPSec Packet Processing 23
Setting up an IPSec SA 23
4.4 IPSec SA Bundles [Transport Adjacency] 24
4.5 SA Tear down 25
4.6 IPv6 Extension Headers 25
Overview 26
Outbound Processing 29
Inbound Processing 33
Additional Specifications 36
4.7 IPv4 Options Header 37
4.8 UDP Enapsulation (NAT Traversal) 37
Chapter 5 Boot Microcode Instruction Set 38
5.1 Microcode Initialization Instructions 38
Opcode BOOT_INIT 38
5.2 Debug Instructions 41
Opcode READ_INTERNAL_DATAPATH 41
Opcode CHECK_ENDIAN 42
Opcode ECHO_CMD 43

Opcode ROLL_CALL 44
Chapter 6 IPsec/IKE Microcode Instruction Set: 45
6.1 IPSec Packet Processing Instruction Group 45
Opcode WRITE_IPSEC_OUTBOUND_SA 45
Opcode WRITE_IPSEC_INBOUND_SA 54
Opcode ERASE_CONTEXT 60
Opcode PROCESS_OUTBOUND_IPSEC_PACKET 61
Opcode PROCESS_INBOUND_IPSEC_PACKET 72
6.2 General Cryptography Instruction Group 76
Opcode RANDOM 76
Opcode MOD_EX 78
6.3 AES_CTR(SRTP) Instruction Group 80
Opcode AES_CTR(SRTP)_ENCRYPT 80
Opcode AES_CTR(SRTP)_DECRYPT 83
Appendix A Instruction Set Quick Reference: 86
Boot Microcode Instruction Set 86
IPSec/IKE Microcode Instruction Set 86
Appendix B Completion Code Quick Reference 88
Appendix C SHA Hash Initial Values 92
Appendix D Microcode Change History 94

Chapter 1 IPSec/IKE Feature Set

Lists and describes the feature set implemented by the specific microcode version(s) listed in the introduction.

ESP

- Tunnel and Transport Modes
- Encryption Algorithms:
  - AES CBC (128, 192, 256), AES CTR (128, 192, 256), AES-GCM (includes auth)
  - DES, 3DES CBC
  - NULL
- Authentication Algorithms
  - MD5, SHA1, SHA2 (256, 384, 512)
  - NULL (NULL encryption and NULL authentication are mutually exclusive)
- Traffic Flow Confidentiality Service (tunnel mode only)
  - Dummy Packet Generation
  - Support for Arbitrary Padding
- Extended (64-bit) Sequence Number support

AH

- Tunnel and Transport Modes
- Supported algorithms:
  - MD5, SHA1

UDP Encapsulation (NAT traversal)

- In IPSec Transport Mode, incremental checksum computed on TCP, UDP (cannot re-compute complete TCP/UPD checksums).
- ESP packets only (not supported with AH)

IPv4 and IPv6

- Tunnel and Transport modes
- ESP, AH, and bundles

SA Bundling (a.k.a Transport Adjacency)

- AH as outer SA, and ESP as inner SA
- Transport mode only. Not available in tunnel mode.
- AH and ESP processing in single pass of data across host interface bus

Erase Context

- "Zero out" an existing IPSec context memory location.

Outbound Pre−Fragmentation
- Fragments outbound packets before encrypting
- Tunnel Mode only
- ESP or AH only. Not supported with bundles.

Outbound Post−Fragmentation
- Fragments outbound packets to equal sizes after encryption
- Requires software reassembly on inbound, prior to decryption
- ESP or AH only. Not supported with bundles.

Selector Checking
- Optionally verifies expected IP Source & Destination Address and Port (4-tuples) on Inbound packets.
- Optionally verifies expected Protocol field (5th tuple)
- If Tunnel Mode, checks inner-packet 4 or 5-tuple.

IPComp Support
- On outbound, encrypts IPCOMP packets or IP packets (Tunnel mode only)
- On inbound, detects and reports IPCOMP packet after decryption.
- Fragmentation not supported for IPCOMP packets
- Discreet ESP or AH only. Supported with SA Bundles.

Modular Exponentiation
- Up to 4096 bits

AES CTR encrypt/decrypt
- General purpose crypto-API

Any-byte Alignment
- Input and output data buffers may be aligned at any byte address

In-place Processing
- Packets may be processed “in-place” in host system memory. In this case, output data (at Rptr) will be written to the same memory buffer where input data was previously read (at Dptr). Refer to Section 2.5 Typical NITROX III Instruction Execution Flow for more information on how to use this feature.

Scatter−Gather DMA
- Provides support for multiple non-contiguous buffers for input and output data
- S/G list must be a contiguous, 8 −byte aligned buffer
- S/G list is minimum of 88 bytes
- S/G Pointers can be arbitrary (any-byte aligned)
- Gather−Only also supported (multiple input buffer to single output buffer)

Interrupt upon Completion

- When enabled in the instruction Opcode, NITROX will generate an interrupt after completion of instruction processing.

Interrupt Coalescing

- When enabled in Interrupt Enable Register, NITROX III will generate an interrupt after completion of a preset number of successful commands has completed, or when a preset timer value counts down to 0, whichever happens first.
- Interrupt Counter and Timer values configured via NITROX III registers
    - GENINT_COUNT_THOLD: Interrupt Counter Threshold
    - GENINT_COUNT_INT_TIME: Timer Threshold. Timer value is in units of NITROX III core clock cycles. For Nitrox_N3, 400000000 clock cycles ~= 1 second.

Chapter 2 NITROX III Microcode General Overview

Reviews the general structure of a NITROX III instruction, and describes the general instruction execution flow.

2.1 NITROX III Instruction/Data Flow

As described in the CN16xx and CN15xx hardware manuals, NITROX processors “pull” (DMA read) one or more 32-byte NITROX instructions from a software instruction queue configured in host system memory. If a given instruction requires input data, NITROX will also pull the variable-sized input data structure from host system memory. If the instruction produces output data, NITROX will “push” (DMA write) the variable-sized output data structure to host memory during processing.

NITROX notifies the host system of instruction completion by writing a 1-byte completion-code to an instruction-specific “completion address” in host memory. NITROX will write one completion code to host memory for each instruction executed. The 1-byte instruction completion code reports either successful completion (CCODE==0x00), or various instruction error codes (CCODE!=0x00). Specification of all instruction-specific CCODES is included with each instruction in this manual. NITROX instructions may also be configured to optionally notify the host system of instruction completion via host system interrupt.

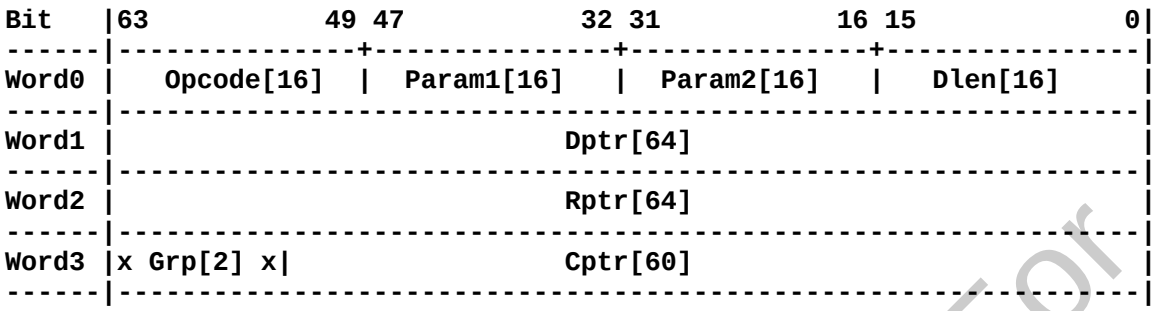
In the event of a serious instruction syntax or data error, resulting in NITROX microcode being unable to calculate an appropriate completion address, NITROX will notify the host system of the error condition by writing an error code to a preconfigured “error address” shared by all NITROX cores. Refer to section 3.3 Microcode Initialization for additional information about microcode-detected error conditions and responses.

Host software initiates one or more NITROX instructions by writing the associated instruction(s) and corresponding input data structure(s) to a queue in host system memory, and then writing to a NITROX “doorbell” register with the number of instructions to be executed. Host software then polls its local completion address memory location(s), looking for a valid completion code for each instruction. When the completion address is written for a given instruction, this indicates that NITROX has completed processing for that instruction.

NITROX is a parallel processing device. There are multiple identical cores, each executing its own microcode. Each core executes instructions independently of other cores, and instruction execution is not guaranteed (in fact is not likely) to complete in the order submitted to the device. Host driver and application software must therefore be designed to utilize the multi-core NITROX efficiently.

2.2 Basic Instruction Format

The general format of all NITROX III instructions is presented below.



Instruction Format Rules:

Opcode[16] - specifies the instruction type. All instruction opcodes are comprised of a Major Group, a Minor Group, and flags (control bits) used to configure various general processing options. Opcode[16] bits are specified as follows:

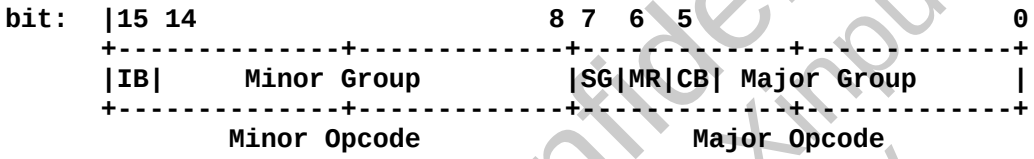


Table 1: NITROX III Opcode[16] Bit Fields

Field	Name	Function
4:0	Major Group	Identifies the major operational group.
5	CB; Completion Bit	Used for “In-place Processing”. Upon instruction completion, this bit will be written to the most significant bit of the completion code byte that is written at the completion address. Set this bit to the inverse value of the most significant bit of the byte of the input data buffer, at the calculated completion address. Refer to section **2.5 Typical NITROX III Instruction Execution Flow** for additional information about In-place Processing.
6	MR; More Requests	Used in legacy “NITROX Plus” microcode images only. Ignored otherwise. When set, indicates to “Plus-Mode” microcode that another instruction should be processed by this core, once this instruction is completed. 0 = No More Instructions 1 = More Instructions

7	SG; Scatter Gather	Use scatter-gather DMA Mode 0 = Direct Mode 1 = Scatter/Gather Mode Refer to Section 2.3 DMA Modes for additional information about this feature.
14:8	Minor Group	Identifies the operation within the major group.
15	IB; Interrupt Bit	Enable Interrupt-on-completion for this instruction. 0 = Do not set interrupt status on completion of this instruction 1 = Set interrupt upon completion of this instruction Refer to Section 2.1 NITROX III Instruction/Data Flow for a detailed description of this feature.

Param1[16] - instruction parameter field. Function varies with instruction type.

Param2[16] - instruction parameter field. Function varies with instruction type.

Dlen[16] – specifies total length (in 8-byte quadwords) of input data structure pointed to by the Dptr field of the instruction.

Dptr[64] - In direct DMA mode, this is the address of instruction input data in host system memory. In scatter/gather mode, this is the address of the scatter/gather descriptor list in host system memory. Refer to Section 2.1 NITROX III Instruction/Data Flow below for additional information about DMA modes.

Rptr[64] - In direct DMA mode, this is the address of the instruction output (response) data buffer in host system memory. In scatter/gather DMA mode, this is the completion address, where the completion code will be written for this instruction.

Cptr[60] - the host memory address where an instruction's "session context" information is stored (session keys and/or other session related data). This pointer must start on an 8-byte boundary (bits 2:0 must be zero).

Grp[2] – specifies the NITROX III core group to submit this specific instruction to. Refer to the NITROX III Hardware Manuals for additional information about how to configure up to 4 different core "groups".

2.3 DMA Modes

NITROX processors execute instructions that perform complex operations on input data structures, producing corresponding output (response) data structures. Input data structures are read from host system memory into the NITROX chip on behalf of the NITROX cores, by shared DMA circuitry of the NITROX Host System Interface (see the NITROX Hardware Manuals for a description of cores and the Host System Interface). Output data structures are written to host system memory on behalf of NITROX cores, by this same shared NITROX DMA logic.

There are two DMA modes available with NITROX III, Direct mode and Scatter/Gather mode. DMA mode selection is made via bit 7 of the Opcode field of the instruction. When Opcode<7>==0, Direct DMA mode is

used. When Opcode<7>==1, Scatter/Gather DMA mode is used.

Direct DMA Mode (Opcode<7>==0)

In Direct DMA mode the input data referenced by Dptr, and the response (output) data buffer referenced by Rptr must exist in host memory as two contiguous blocks of memory. Refer to chapters 5 or 6 of this document as needed, to determine the size of the input and output buffers needed for any given instruction.

Scatter/Gather DMA Mode (Opcode<7>==1)

In Scatter/Gather DMA Mode, input and output data may exist in host memory in scattered memory fragments. NITROX will gather all input data according to a gather list, and scatter all output data according to a scatter list. In this mode, Dptr points to the gather and scatter lists in host memory, instead of pointing to the input data itself.

In this mode, input and output data buffers can each consist of several independent memory fragments. Each fragment can be located anywhere in host system memory. Fragments are not required to be contiguous to one another, nor do they have alignment restrictions of any kind; data within any given fragment can begin and/or end on any byte-address boundary. Data within each fragment must be contiguous.

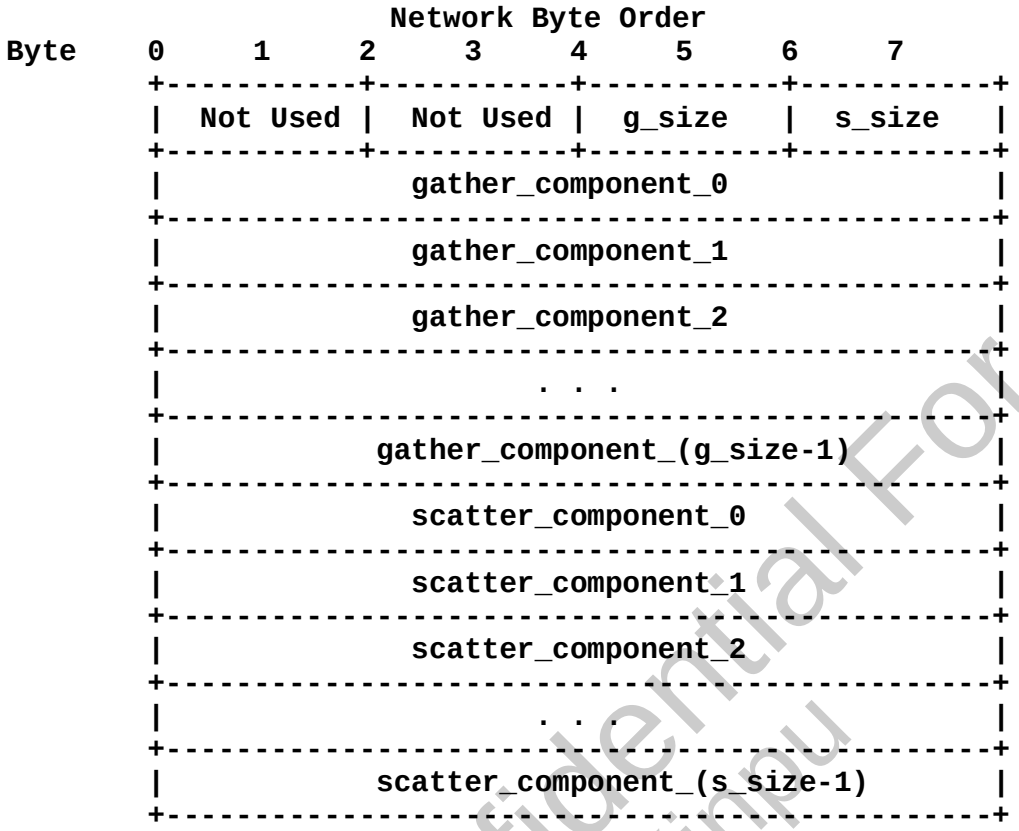
Most of the fields of the general NITROX 32Byte instruction format described above, are the same for both direct and scatter/gather modes. The definition of some of the fields is mode-dependent however. In scatter/gather mode:

Dlen: indicates the size (in 64-bit words) of the scatter/gather list, instead of the size of the actual input data structure. The maximum total number (gather+scatter) of scatter/gather list pointers is 100. As will be shown below, this mandates that for Scatter/Gather mode the max value of Dlen is 1008 bytes (0x03f0).

Dptr: points to a scatter/gather list in host system memory, rather than directly to input data. This list describes the structure of both input data buffers and allocated output buffers in host memory. This list must be in a single, contiguous, 8-byte aligned buffer.

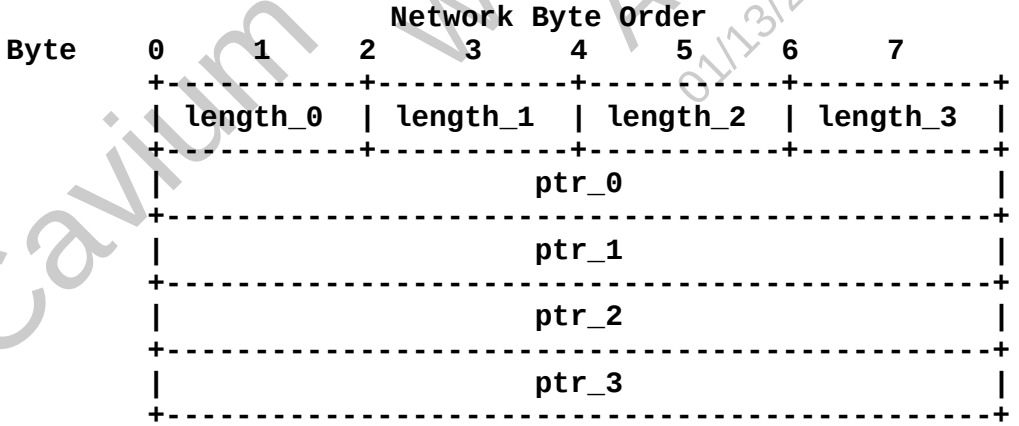
Rptr: points to the instruction completion address, rather than directly to the output/response data buffer (unless using gather-only DMA, as described below). i.e. in scatter/gather mode the only data written to Rptr is the one-byte completion code.

The structure of the Scatter-Gather list is described as follows:



The fields s_size and g_size are each 2 −byte values representing the total number of individual buffers/pointers in the scatter and gather lists. The range of both s_size and g_size is 1 to 96.

Each gather or scatter component is a 40 byte data structure describing the lengths and addresses of four discrete buffers in host memory. Below is the structure of individual Scatter/Gather List components:



Scatter/Gather Mode Constraints:

- g_size range: 1 to 96
- s_size range: 1 to 96
- Maximum total buffers/ptrs (gather + scatter): 100
- Maximum total g_size + s_size: 25 (100 total pointers/4 pointers per s/g component)
- Max Dlen: 1008 [8B + (25 * 40B) s/g components]
- Ptr(n) buffer size range: 0<=size<=2^15 −1
- Ptr(n) must point to host memory, and bit 63 must be 0.
- Ptr(n) byte alignment may apply. Refer to instruction spec for details.

2.4 Error Address/Error Codes

Under normal circumstances, NITROX will write a completion code (CCODE) to the Completion Address in host memory, as described in section **2.1 NITROX III Instruction/Data Flow.**

In the event of a severe error (one where microcode is unable to calculate a reliable Completion Address), microcode will generate and assert a Microcode General Interrupt. This interrupt bit indicates that either a severe error was detected by microcode during instruction processing, or that the instruction has completed while in "Interrupt-on-Completion" mode (refer to Interrupt Status registers in the NITROX III hardware manual).

In addition to generating the above interrupt upon detection of a severe error, microcode will also write an 8-byte Error Code (ECODE) to host memory at the Error Address (ERROR_ADDR) specified by the host system during the microcode loading process. Refer to sections **3.4 Boot Initialization Parameters** and **5.1 Microcode Initialization Instructions** for information about how to initialize ERROR_ADDR.

When using Interrupt-on-Completion mode, both the Error Address and the normal Completion Address should be monitored for completion status, to prevent potential ambiguity between a severe error and a normal (non-error) completion event.

The format of the 8-byte ECODE value written to ERROR_ADDR is shown in the figure below:

CCODE: 8-bit Completion Code
COREID: Number of the core that reported the severe error
RPTR6: Lower 6 bytes of Microcode Instruction word2 (Rptr). This can be used to assist in uniquely identifying which specific instruction caused the error. This assumes that each instruction has a unique result location within some known period of time.

2.5 Typical NITROX III Instruction Execution Flow

To execute a NITROX III instruction, host software (an API and Driver running on the host CPU) typically performs the following operations:

- Mark the output buffer as “clean” by writing 0xFF to the completion address. NOTE: if in-place processing is used, this step is not done. Refer to section **In-Place Processing** below for additional information.
- Add one or more NITROX instructions to the end of the host instruction queue. As described above, instruction queue entries are 32 bytes, and may point to an input data buffer, an output response/data buffer, and/or a context data buffer.
- Write the number of new instructions added to the host queue, to the doorbell register in the Instruction Queue Manager (IQM) of the NITROX processor. In this flow example only one instruction will be added to the queue, so a 0x00000001 would be written to the Doorbell Register.
- Poll for the instruction(s) to complete. A common method for polling would be to use a poll-sleep-wake-poll-sleep scheme. However there are many ways to implement efficient polling mechanisms. The topic of polling mechanisms is beyond the scope of this document, and it is not described in detail herein.

In response to having its doorbell register written, NITROX will perform the following operations:

- The IQM adds the desired number of entries written to its doorbell register, to the IQM’s count of valid host queue entries to be read.
- The IQM issues a DMA READ command to the Host System Interface (HSI). This generates a DMA read to ‘pull’ one or more instructions from the host instruction queue into the internal instruction queue. Up to eight instructions (256 bytes) can be burst from host memory into the internal queue in one DMA operation.
- The resulting data from the DMA READ above is written to the IQM. Once completed, the IQM decrements its count of valid host queue entries, by the number of instructions copied from the external to the internal queue.
- The IQM writes the next available instruction in its internal (hardware) instruction queue to the register file of the next available NITROX processor core.
- The NITROX core begins processing the instruction by decoding the opcode and parameter fields. If required, the NITROX core reads input data from host memory at the address specified in the instruction, by issuing DMA READ commands to the HSI block.
- If required, the core then reads relevant context or keys from the address specified in the Cptr field of the instruction, by issuing DMA reads to the HSI block.

As processing produces output data during instruction execution, the core issues DMA WRITE commands to the HSI, writing output data from the core to host memory at the address specified in the instruction. The core also updates context memory state if needed, by issuing DMA WRITE commands to the HSI block.
The core then concludes instruction execution by writing the completion code (CCODE) to the completion address calculated during instruction execution, and informing the IQM that it is available to process another instruction.

Host software (NITROX driver) will see CCODE change from the initialized value of 0xFF, to 0x00 (in most cases), and notifies the application that the instruction has completed.

There are multiple independent processor cores in a NITROX III processor. The IQM assigns instructions to the cores in the order received into its queue. However because different types of instructions may take

different amounts of time to complete, instructions may (in fact are likely to) complete out of order. For example, an RNG instruction will be performed much faster than a modular exponentiation. Even two modular exponentiation operations may take different amounts of time to complete.

Driver and API software in the host system can be written to maintain processing order if required.

In-Place Processing

Software may set Rptr to the same value as Dptr, resulting in output data being written to the same memory area where input data is being read. This is referred to herein as “In-Place Processing”. In general, NITROX III DMA processing will create a 128 byte “cushion” between input data reads and corresponding output data writes.

Generally speaking, in-place processing is always enabled on NITROX III. When NITROX III microcode is finished processing an instruction, it reads the value of bit Instruction:Word0:Opcode:CB (see Opcode Field Definition), and copies it to CCODE:CS (bit 7) of the completion code (CCODE). Microcode then writes the resulting CCODE to the completion address.

For example, if the completion code for a given instruction was 0x00 (SUCCESS), and Instruction Word0:Opcode:CB for that instructions was set to 0 by the host processor, NITROX III would write 0x00 to the completion address when the instruction completed. If however, Instruction:Word0:Opcode:CB was 1, NITROX III would write 0x80 to the completion address. This is always done, without regard to whether or not in−place processing is being used. This provides two options for the host processor to implement instruction completion notification.

The host processor may initialize the completion address (the location where it expects to see the completion code (CCODE)) to 0xff, and keep Instruction:word0:Opcode:CB clear. The host will then poll this completion code address in host memory until its value changes from 0xff to a valid completion code (e.g. 0x00), indicating that NITROX III has successfully completed the request. With in-place processing though, this type of simple scheme may fail because initializing the completion address to 0xff may corrupt the input data. After all, we are using the same buffers for input and output.

To prevent this situation, the host should not initialize the completion address to 0xff for in-place processing. Instead, the host should read the value of the CCODE:CS bit (bit 8) at the completion address within the input buffer (which is also the response buffer) and write that bits compliment to Instruction:Word0:Opcode:CB bit of the instruction. The host can then poll the completion address to check for toggle of bit CCODE:CS, indicating that instruction processing has completed. Once the CCODE:CS bit toggles, the value at the completion address can be assumed to be a valid completion code.

For example, if the value of the input buffer at the completion address was 0xAA, then CCODE:CS (bit 8) = 1. The host should then set Instruction:Word0:Opcode:CB to 0. The host should then poll the completion address and check for CCODE:CS to toggle from 1 to 0, indicating that this byte of the input buffer has been over-written. The host processor then checks the value of the CCODE to identify status of the completed event.

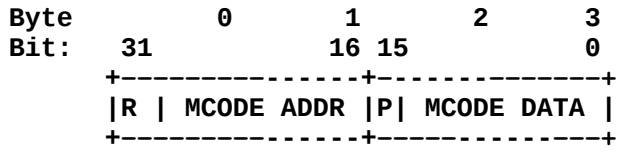
As an optimization, if the output data is always larger than the input data, the host does not need to worry about corrupting the input buffer since the completion code will be written at an address that is beyond the last byte of the input data. Initializing the completion code to some value does not corrupt the input buffer.

Chapter 3 NITROX III Microcode Loading Process

This chapter describes what “Boot” Microcode is, what “Main” microcode is, the layout of a microcode binary file, and the microcode loading and initialization process.

3.1 Microcode Loading Fundamentals

At the basic hardware level, microcode is loaded into any disabled NITROX core by performing PCI target mode writes to the chip’s UCODE_LOAD register (BAR0/4 + 18h). UCODE_LOAD is a 32-bit register consisting of an address field, a data field, and a data parity field, as per the diagram below.

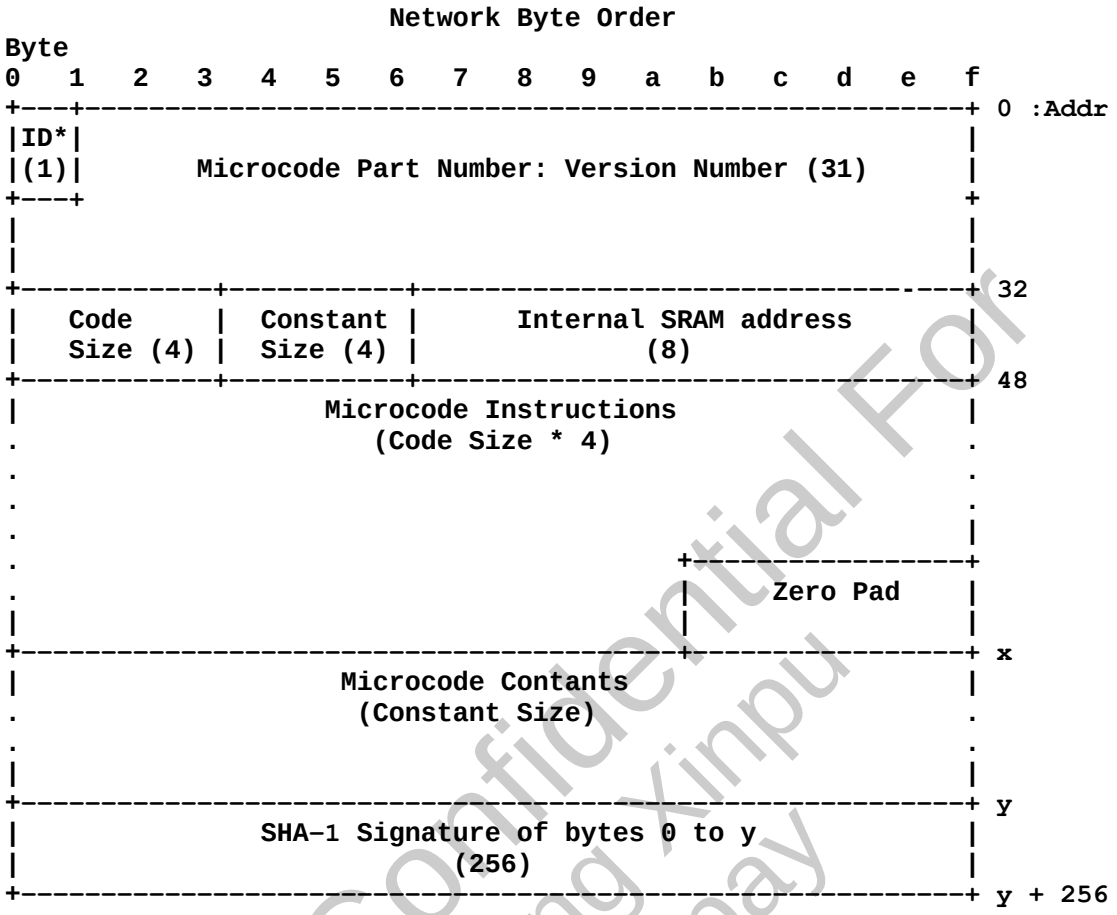


<31:30>: R (Reserved)
<29:16>: MCODE ADDR
<15:15>: P (MCODE Parity)
<14:00>: MCODE DATA

Any time this register is written by the host system, NITROX will write the data+parity fields to the microcode control store memory, at the specified address. Again, only disabled NITROX cores will be written to.

Cavium provides various NITROX microcode binary files. These files contain, among other things, an array of 32-bit address+data+parity values. This array is what must be written to the UCODE_LOAD register by the host system, at chip initialization time. MCODE parity is pre-calculated and included in this array. Parity is then checked by hardware on every read of the microcode control store, in every NITROX core. In the event of a parity error on any core, that core is automatically disabled by hardware and the error is reported by the ***CORE INSTRUCTION PARITY ERROR*** bit in the INTERRUPT_STATUS register.

3.2 Microcode Binary File Layout



Location x = 48 + ROUNDUP16(Code Size * 4)
Location y = x + Constant Size
Values in parenthesis indicate size in bytes.
*ID is the microcode ID

Microcode Part Number: Version Number - a text value identifying the version of the microcode binary file. It can be easily read by opening the binary file with a standard text editor. Most of the binary data will be displayed as “binary gibberish”, but this field will display as standard ASCII text. For example, “CNN3-MC-IPSec-0001”.

Code Size – specifies the number of bytes in the Microcode Instructions field

Constants Size – specifies the number of bytes in the Microcode Constants field

Internal SRAM Address – specifies the address in NITROX III internal SRAM, to store the pointer to where the constants are stored as a hex array in host memory.

Microcode Instructions – an array of microcode instructions, as per description in previous section of this document.

Microcode Constants – an array of 8−byte "constant" values, such as 0x00000000 0000000, 0x5c5c5c5c 5c5c5c5c, or 0x36363636 36363636, etc. These constants are used in various NITROX instructions to perform things like HMAC calculations, checksums, zero−padding, etc.

3.3 Microcode Initialization

NITROX III microcode utilizes various fixed addresses and constants during runtime. These fixed values must be initialized on the chip and in host memory at device/driver initialization time (i.e. any time a NITROX device driver is loaded, and the chip is initialized/configured). NITROX microcode configuration is performed by executing a small, specialized set of configuration instructions prior to executing any other "normal" NITROX instructions.

In addition, various specialized NITROX device and system debug instructions have been implemented in NITROX microcode.

Because these configuration and debug instructions consume valuable NITROX control store memory resources, and because they are only used during boot-up and/or for hardware troubleshooting, Cavium has implemented this small set of "management" instructions into a separate "boot" microcode binary file. During system initialization, this boot microcode is loaded into the device first, before the primary, or "main" microcode is loaded onto the chip. Boot-time configuration instructions are then executed by the host system to initialize the NITROX device for normal operation. Once initialized, the primary or "main" NITROX microcode can be loaded onto the chip (resulting in the boot microcode being overwritten and thus discarded).

3.4 Boot Initialization Parameters

Below is a list and description of the items that are initialized by instructions in the boot microcode.

Core watchdog timer value:
NITROX cores implement a programmable hardware watchdog timer that resets a NITROX core and generates an interrupt if that core's timer counts down to 0 from a programmable initial value. The timer starts counting down at the beginning of execution of any instruction, and is reset to (held at) the initial value any time a core is idle. The timer's start value is configured on the chip, via a parameter provided to an instruction in the boot microcode instruction set.

Core scratchpad base address, size:
In NITROX IPSec/IKE microcode, each NITROX core requires its own scratchpad area in host system memory. This scratchpad memory must be configured as a contiguous block of host system memory. The base address of the scratchpad area, and the size of each scratchpad, is initialized on the chip by parameters provided to an instruction in the boot microcode instruction set. Each core then uses the following formula to locate its scratchpad.

Scratchpad Address = scratchpad_base_address + (core_number * scratchpad_size)

Error Address (ERROR_ADDR):
If NITROX is presented with an instruction and/or input data that it cannot decode sufficiently to generate an expected completion address, it will write an error code to a preconfigured "error address" in host system memory. That address is configured in NITROX internal SRAM, and is provided as a parameter to an instruction in the boot microcode instruction set. Refer to section **2.4 Error Address/Error Codes** for additional information about ERROR_ADDR usage.

Context window base address and SPI key constants:
These are constants used by NITROX for inbound Ipsec packet processing. They are used to calculate the absolute address of IPSec inbound SA's stored in host system memory. The formula used is: Absolute_address = context_window_base + (SPI_key XOR SPI_value).

Constants Address:
As per the description in the previous section, the microcode binary file contains an array of constants, and also a 64-bit field containing a 16-bit NITROX Internal SRAM Address. A NITROX driver must store a copy of this constants array in a hex array in host system memory. It must then provide the host memory address of this array, and also the 16-bit Internal SRAM Address as parameters to the boot instruction set. This NITROX instruction will store this specified host Constants Address at the specified Internal SRAM Address in the hardware. Then, any NITROX core containing "main" NITROX microcode, when it is first enabled, will read the internal SRAM address to ID the host memory address of the constants array, and then DMA read this array into its own register file.

3.5 Debug Instructions

The Boot microcode includes a small set of instructions intended for use in debugging NITROX III hardware. Refer to section **5.2 Debug Instructions** for a list and details of boot debug instructions.

Chapter 4 IPSec/IKE Microcode Users Guide

A primer describing intended usage of NITROX IPSec/IKE instructions, to implement a typical IPSec/IKE application.

4.1 Basic API Notation

Below is a summary list of the IPSec/IKE APIs implemented in the version of NITROX III IPSec microcode specified by this manual, along with the associated C-API prototypes one might use to access the microcode API in a typical Cavium Networks IPSec SDK.

C-API	Microcode OPCODE
Csp_Random()	RANDOM
Csp_Me()	MOD_EXP
Csp_Write_outbound_sa()	WRITE_IPSEC_OUTBOUND_SA
Csp_Write_inbound_sa()	WRITE_IPSEC_INBOUND_SA
Csp_Process_outbound()	PROCESS_OUTBOUND_IPSEC_PACKET
Csp_Process_inbound()	PROCESS_INBOUND_IPSEC_PACKET

4.2 Internet Key Exchange (IKE)

Internet Key Exchange (IKE) is a protocol used to setup a Security Association (SA) for IPSec by exchanging handshake messages. The IKE message exchange is CPU intensive. Using NITROX API calls for the Modular Exponentiation and Random Number Generation used in the Diffie-Hellman key exchange will significantly enhance system performance.

IKE Notation

The below notation is taken from RFC 2409, "The Internet Key Exchange (IKE)" (*www.ietf.org/**rfc/rfc2409**.txt*)

- HDR is an ISAKMP header whose exchange type is the mode. When written as HDR* it indicates payload encryption.
- SA is an SA negotiation payload with one or more proposals. An initiator MAY provide multiple proposals for negotiation; a responder MUST reply with only one.
- SAi_b is the entire body of the SA payload (minus the ISAKMP generic header).
- IKE is the key exchange payload which contains the public information exchanged in a Diffie-Hellman exchange.
- Nx is the nonce payload; x can be: I or r for the ISAKMP initiator and responder respectively.
- Idx is the identification payload for "x". x can be: "ii" or "ir" for the ISAKMP initiator and responder respectively during phase one negotiation; or "ui" or "ur" for the user initiator and responder respectively during phase two.
- HASH (and any 21erivative such as HASH(2) or HASH_I) is the hash payload. The contents of the hash are specific to the authentication method.
- ---> signifies "initiator to responder" communication (requests).
- <-- signifies "responder to initiator" communication (replies).

- [x] indicates that x is optional.

The IKE SA (ISAKMP SA) is similar to the IPSEC SA except that it is bi-directional. It is established after the IKE Phase-I exchange, and it protects the IKE Phase-2 message exchanges.

IKE Phase 1 – Main Mode

The description below shows how CAVIUM API's can be used within the IKE message exchanges. In this example consider two peers, with one trying to establish an Ipv4–transport-mode IPSec communication using pre-shared keys, and the other through IKE.

In the message exchange shown, the first two messages negotiate the policy, the next two messages exchange Diffie-Hellman values and nonces (ancillary data), and the last two messages authenticate the Diffie-Hellman exchange. A bi-directional ISAKMP SA is established as a result of phase-1.

Phase 1 : Main mode authenticated with a Pre-shared Key

INITIATOR Operations	Msg Direction	RESPONDER Operations
HDR, SA	---->	
	<----	HDR, SA
HDR, KE, Ni /* initiator nonce */ Ni_b = ***CspRandom();*** /* private key */ x = ***CspRandom();*** /* public key X = g^x mod n */ X = ***CspMe(g,x,p);***	---->	
	<----	HDR, KE, Nr /* responder nonce */ Nr_b = ***CspRandom();*** /* private key */ y = ***CspRandom();*** /* public key Y = g^y mod p */ Y = ***CspMe(g,y,p);*** /* compute shared secret K = Y^x mod p */ K = ***CspMe(Y,x,p);***
HDR*, IDi, HASH_I /* compute shared secret K' = X^y mod p */ K' = ***CspMe(X,y,p);***	---->	
	<---	HDR*, IDir, HASH_R

IKE Phase 2 – Quick Mode

Quick mode is an SA negotiation process used to negotiate policy and derive keying material for the actual IPSec SA. The information exchanged will be protected by (within) the ISAKMP SA.
An optional Key exchange payload can be used to have an additional Diffie-Hellman exchange.

Phase 2 : Quick mode

INITIATOR Operations	Msg Direction	RESPONDER Operations
HDR*, HASH(1), SA , Ni , [KE], [IDci, Idcr] /* initiatior nonce */ Ni_b = ***CspRandom();*** /* private key */ x = ***CspRandom();*** /* public key X = g^x mod n */ X = ***CspMe(g,x,p);***	---->	
	<----	HDR*, HASH(2), SA, Nr, [KE] [IDci, IDcr] /* responder nonce*/ Ni_r = ***CspRandom();*** /* private key */ y = ***CspRandom();*** /* public key X = g^y mod p */ Y = ***CspMe(g,y,p);*** /* compute shared secret K = Y^x mod p */ K= ***CspMe(Y,x,p);***
HDR*, HASH(3) /*compute shared secret K' = X^y mod p*/ K' = ***CspMe(X,y,p);***	---->	

4.3 IPSec Packet Processing

After successful completion of an IKE phase-2 negotiation, IPSec SA's are established in the initiator and responder systems.

Setting up an IPSec SA

Below are the steps required to establish an IPSec SA. **References below to software structures are taken from the linux 2.6.11 kernel source (www.kernel.org).** The 2.6.11 kernel patch in the NITROX III IPSec SDK is taken as reference.

- Find the type of the destination address (*id.daddr.a4* of the *xfrm_state* structure) using the function *inet_addr_type()*. If the address is RTN_LOCAL (a local address) then the direction is inbound, else outbound.
- Call API ***Csp1AllocContext()*** to allocate a context and return a context handle, which is stored in the *context* variable of the corresponding *xfrm_state* structure.
- Extract information from the *xfrm_state* structure like:
  - Protocol (ESP/AH), from the *id.proto* field of *xfrm_state*
  - Encryption algorithm to be used, from the *props.ealgo* field of *xfrm_state*
  - Authentication algorithm to be used, from the *props.aalgo* field of *xfrm_state*
  - etc...

- The information from the previous step is provided as parameters to the NITROX APIs ***Csp_write_inbound_context()*** or ***Csp_write_outbound_context()***. These APIs write SA context data into context memory structures in kernel space. For the case of an inbound IPSec SA, call ***Csp_write_inbound_context()*** with the required parameters. This API writes the SA into the context in the kernel space. For the outbound SA case, call ***Csp_write_outbound_context()*** with the required parameters.

Once the SA's are setup on both the initiator and responder, IPSec traffic can be sent and received on both sides.

Inbound Packet Processing – IPSec to IP

The steps below are required for inbound packet processing.

- The IP transport protocol is detected by the Ethernet stack. The IP packet is then extracted, and passed to the IP stack by functions *ip_rcv()* and *ip_rcv_finish()* respectively. Inbound IPSec packets are detected by examining the next header value in the outer IP header. They are then passed the *ip_local_deliver()* function.
- Once an inbound IPSec packet is detected, an IPSec SA lookup is performed. Decoding of IPSec packets is done by *xfrm_rcv()*
- IPSec packet authentication headers (AH) are tested by *ah_input()*. IPSec packet decryption (and optional authentication) is performed by *esp_input()*.Within *ah_input()* and *esp_input()*, NITROX API ***Csp_inbound_process()*** is called with necessary parameters derived from the *xfrm_state* and *sk_buff* structures related to the packet.

Once the above steps are completed, the resulting fully decoded (i.e. decrypted and/or authenticated) IP packet is returned to the IP layer for further processing by the network stack.

Outbound Packet Processing – IP to IPSec

The steps below are required for outbound packet processing.

- IP Packets from upper protocol layer are received by *ip_queue_xmit*().
- Policies are applied to determine if the packets need IPSec processing.
- If the IP Packet needs to be encrypted (and optionally authenticated), encryption is done by *esp_output()*. If the data is to be authenticated, authentication is performed by *ah_output()*. In *ah_output ()* and *esp_output()*, NITROX API ***Csp_outbound_process()*** is called with the necessary parameters derived from the *xfrm_state* and *sk_buff* structures related to the packet.

Once the above steps are completed a complete IPSec encoded IP packet is returned to IP for further processing by the networking stack.

4.4 IPSec SA Bundles [Transport Adjacency]

NITROX III microcode automatically handles SA bundles (e.g. ESP+AH) in order to support and accelerate transport adjacency. A bundle is defined here as a pair of (2) SA contexts. The first SA context is identified by the Cptr of the PROCESS_INBOUND_IPSEC_PACKET or PROCESS_OUTBOUND_IPSEC_PACKET instructions. The *Next SA* bit in the control field of this SA must be set, and the *Next SA* field must point to the second context. NITROX III microcode supports bundles of only two SAs. The *Next SA* bit in the control field of the second SA must be cleared (0).

This feature has been implemented to accelerate SA bundling. However bundling can also be done using two separate calls.

NOTE: When using bundling, the NITROX III scratch pad size in host memory should be configured, and MUST be larger than the biggest packet size plus 1k bytes. The scratch pad size and location are configured in the BOOT_INIT instruction.

Outbound Bundles

- Input request Cptr points to ESP SA. [Inner SA]
- Next SA bit in the ESP SA will be set and Next SA pointer will point to AH SA [Outer SA].
- Next SA bit in the AH SA should be cleared (0).

Inbound Bundles

- Input request Cptr points to AH SA. [Outer SA]
- Next SA bit in the AH SA will be set and Next SA pointer will point to ESP SA [Inner SA].
- Next SA bit in the ESP SA should be cleared (0).

Supported bundle features

Outer SA should be:

- AH
- Transport mode
- No selector check support
- No UDP encapsulation

Inner SA should be:

- ESP
- Transport/Tunnel
- Selector check [Inbound Only]
- Header Preservation [only on Outbound Tunnel mode]
- UDP encapsulation

Unsupported bundles features

- Pre and Post frag

4.5 SA Tear down

The following steps are required for tearing down an IPSec SA. **References below to software structures are taken from the linux 2.6.11 kernel source (www.kernel.org).**

- In the *xfrm_state_delete()* function : Get all the required parameters to call the API ***Csp_write_inbound_sa()***. Set the control bytes to '0' in the API to invalidate the SA.

After returning from the API call the ***CspFreeContext ()*** API with the context field of the xfrm_state as argument. This frees the context of the SA in system memory.

4.6 IPv6 Extension Headers

NITROX III IPSec microcode supports IPv6 input packets with/without extension headers. The NITROX-N3 device will add AH/ESP headers required during the IPSec encryption processing. The NITROX-N3 device will remove AH/ESP headers during the IPSec decryption processing.

Overview

Extension Headers Supported

The NITROX-N3 supports the following extension headers in any order and any number of times except the hop-by-hop header which can occur only once and if present should be the first header.

- Hop-by-Hop Header
- Destination Header *
- Routing Header (type-0)
- Fragment Header
- Authentication Header
- Encapsulating Security Payload
- Final Destination Header *
- Mobility extension header
- Routing type-2 header

* Destination Header is one which comes before AH/ESP header and Final Destination Header is one which comes after AH/ESP header.

The IPv6 extension header processing stops once it hits “no next header”, if present.

Mobility extension header (MH) is always protected using ESP or AH protocol along with the payload data as per RFC 3776. So, this header should be last extension header, if present.

Routing type-2 header is having the type filed value as “2” and segments left field as “1” in the routing header.

This header is restricted to carry only one IPv6 address. If both type-0 and type-2 headers are present, then routing type-2 header should follow the routing type-0 header.

IPv6 modes of Operation

IPv6 Extension Headers is supported in two modes: RFC2460 compliant mode and Override mode. The RFC2460 compliant or override mode is selectable on a per-packet basis via bit 14 of param1 of the instruction. When bit 14 is set to 1, override mode is used. When bit 14 is cleared to 0, RFC2460 compliant mode is used.

RFC2460 compliant mode

The extension headers hop-by-hop, destination and routing are inserted outside the AH/ESP header. The final destination header is considered as part of payload and is inserted inside the AH/ESP header. The fragment header is always outside the AH/ESP header.

Mobility extension header is not supported with RFC 2460 compliant mode.

Override mode

The extension headers hop-by-hop, destination, routing and final destination are inserted outside the AH/ESP header. The fragment header is always outside the AH/ESP header. This mode is useful for IPv6 mobility support according to RFC3775.

MH is always supported in override mode of operation. Along with mobility header, RFC 3775 defines one destination extension header option called as “Home address option”. This is used to carry the home address of the mobile node while away from home.

AH considerations

The classification of IPv6 extension headers is shown below, with regard to mutability for AH computation

Option/Extension Name	Notes
1. Hop-by-Hop and Destination options	Included in the ICV calculation
2. Mutable but Predictable Routing (Type 0)	Included in the ICV calculation
3. Fragmentation	Not applicable

The format of **hop-by-hop** and **destination options** is shown below.

Option type (8-bit)	Option data length (8-bit)	Option data (variable)

The third highest order bit in the option type indicates whether the option type is mutable. If this bit is set, the option type is mutable else immutable. If the mutability bit is set, the entire option data field must be treated as zero-valued octets for the calculation of ICV.

The format of the home address option carried by destination extension header is as follows according to RFC 3775:

Option type(8 bit)	Option data length (8-bit)	Home address (16 bytes)

The home address option must be placed as follows:

- After the routing header, if that header is present.
- Before the fragment header, if that header is present.
- Before the AH or ESP header, if either one of them is present.
- For each Ipv6 header, the home address option must not appear more than once. How ever an encapsulated packet may contain separate home address option.

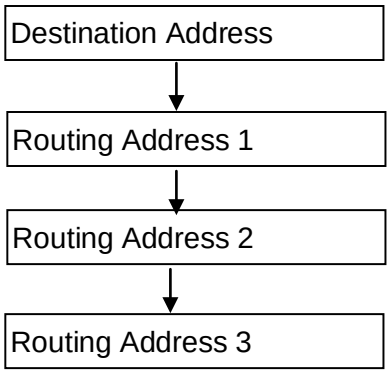
IPv6 Routing Header "Type 0" will rearrange the address fields within the packet during transit from source to destination. During header walk, last routing address is stored at dpRF location IPV6_DEST_ADDR_LOCATION. While muting the following steps are performed.

Last routing address at IPV6_DEST_ADDR_LOCATION is send to the hash.

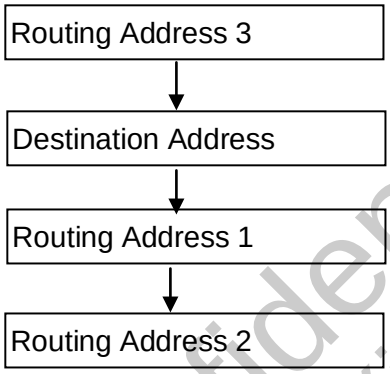
Packets original destination is send to the hash.

Subsequent routing addresses are send to the hash.

Last routing address is stored at original packets destination address location. This is used when next routing header exists i.e., for the multiple routing headers case. The rearrangement of address fields with routing header consisting of three nodes is shown in below figures.



Order of original IPv6 packets destination and routing addresses



Order of IPv6 destination and routing addresses on which ICV is calculated.

Fragmentation header is not considered for the purpose of ICV calculations. If exists, the NITROX-N3 will parse the header.

Outbound Processing

The NITROX-N3 device performs complete outbound IPSec packet processing (encryption) for IPv6. The Input Data for this operation is a complete, fully formed IPv6 packet with/without extension headers. The resulting output is a fully formed IPSec AH or ESP IPv6 packet.

The IPv6 extension header packet processing include

AH

Transport or Tunnel mode in override or rfc2460 mode.

ESP

Transport or Tunnel mode in override or rfc2460 mode.

Transport Mode

For transport mode microcode does the following:

- Encrypts and/or authenticates the payload of the incoming IPv6 packet.
- Adds the AH or ESP header between the IPv6 extension headers and the processed payload.
- Changes the next header field of IPv6 base header or last extension header to AH or ESP header.

Packet Format before applying AH/ESP in rfc2460 or override mode.

Original IPv6 header	Extension headers (if present)	TCP	Data

Packet Format after applying AH in rfc2460 mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment	AH	Final destination	TCP	Data

|<---------- authenticated except for mutable fields ---------->|

Packet Format after applying AH in override mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest.	AH	TCP	Data

|<---------- authenticated except for mutable fields ---------->|

Packet Format after applying ESP in rfc2460 mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment	ESP	Final destination	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

Packet Format after applying ESP in override mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest	ESP	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

Packet Format before applying ESP in override mode with Mobility extension header

Original Ipv6 header	Hop-by-hop, dest, routing, fragment,final dest	Mobility header	TCP	Data

Packet Format after applying ESP in override mode with Mobility extension header

Original Ipv6 header	Hop-by-hop, dest, routing, fragment,final dest	ESP	Mobility header	TCP	Data	ESP trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated---------->|

Packet Format before applying AH in override mode with Mobility extension header

Original Ipv6 header	Hop-by-hop, dest, routing, fragment,final dest	Mobility header	TCP	Data

Packet Format after applying AH in override mode with Mobility extension header

Original Ipv6 header	Hop-by-hop, dest, routing, fragment,final dest	AH	Mobility header	TCP	Data

|<----------authenticated except for mutable fields of ipv6 main header---------->|

Tunnel Mode

For tunnel mode, microcode does the following:

- Encrypts and/or authenticates the complete IPv6 packet with extension headers.
- Adds the AH or ESP header.
- Adds the Tunnel IPv6 header with the next header field of base header or last extension header indicating AH or ESP.

Packet Format before applying AH/ESP in rfc2460 or override mode.

Original IPv6 header	Extension headers (if present)	TCP	Data

Packet Format after applying AH in rfc2460 or override mode.

New IPv6 header	New Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest.	AH	Original IPv6 header	Original Ext Hdrs: Hop-by-hop, dest., routing, fragment, final dest, mobility header.	TCP	Data

|<---------- authenticated except for mutable fields in new IPv6 header ---------->|

Packet Format after applying ESP in rfc2460 or override mode.

New IPv6 header	New Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest.	ESP	Original IPv6 header	Orig. Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest, mobility header.	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

- If mobility extension header is present in the tunnel mode in the outer IPV6 header of the plain packet , an exception ERROP__IPV6_EXTENSION_HEADER_BAD will occur.

Examples of AH/ESP header insertion for override mode (Transport and Tunnel) and rfc2460 mode (Tunnel)

- hop + AH/ESP
- hop + dst + AH/ESP
- hop + dst + route + AH/ESP
- hop + dst + route + dst + AH/ESP

Example of AH/ESP header insertion for override mode in Transport for mobiltiy header

- hop + dst + route + dst + AH/ESP + mobility_hdr

Examples of AH/ESP header insertion for rfc2460 mode (Transport)

- hop + AH/ESP
- hop + AH/ESP + dst (dst is considered as part of payload and is immutable)
- hop + dst + route + AH/ESP
- hop + dst + route + AH/ESP + dst

Header size limitations

In transport mode maximum IPv6 header size is 248 (208(Extension Header) + 40(Base Header)) bytes including extension headers. If the IPv6 extension header length is more than 208 bytes, Nitrox-N3 will throw an exception " ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG(50) ".
In tunnel mode maximum IPv6 header size (template size) is 248 (208(extension header) + 40(Base Header)) bytes including extension headers. If the IPv6 extension header length is more than 208 bytes, Nitrox-N3 will throw an exception "ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG(50)".

Outbound exceptions

- If IPv6 processing encounters multiple hop-by-hop extension header or not the very first extension header then Nitrox-N3 will throw an exception "ERROP__IPV6_HOP_BY_HOP_ERROR (51)".

- In Transport mode, If IPv6 extension header length exceeds 208 bytes then Nitrox-N3 will throw an exception "ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG(50)".
- In Tunnel mode, If IPv6 extension header length exceeds 208 bytes then Nitrox-N3 will throw an exception "ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG(50)".
- If IPv6 routing header's extension header length is not divisible by 2 then Nitrox-N3 will throw an exception "ERROP__IPV6_RH_LENGTH_ERROR (53)".
- If Ipv6 routing header is present and copy address bit enabled in the context then Nitrox-N3 throws an exception "ERROP__IPV6_OUTBOUND_RH_COPY_ADDR_ERROR(54)".
- If ipv6 mobiliry extension header is present in the tunnel mode in the outer IPV6 header of the plain packet , then Nitrox_N3 throws an exception "ERROP__IPV6_EXTENSION_HEADER_BAD (55)" .

Inbound Processing

The NITROX-N3 device performs complete inbound IPSec packet processing (decryption) for IPv6. The Input Data for this operation is a complete, fully formed IPSec AH or ESP IPv6 packet. The resulting output is a fully formed plain IPv6 packet with/without extension headers.
The IPv6 extension header packet processing include

AH

Transport or Tunnel mode in override or rfc2460 mode

ESP

Transport or Tunnel mode in override or rfc2460 mode.

Transport Mode

For transport mode microcode does the following:

- Decrypts and authenticates the IPSec ESP packet, or authenticates the IPSec AH Packet.
- Removes the ESP or AH header.
- Copies the next header field of the AH or ESP header to the next header field of the original IPv6 base header or last IPv6 extension header.
- Returns the plain IPv6 Packet as output.

AH Input Packet Format before AH processing, in rfc2460 mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment	AH	Final destination	TCP	Data

|<---------- authenticated except for mutable fields ----------->|

IP Packet Format after AH processing, in rfc2460 mode.

Original Ipv6 header	Extension headers (if present)	TCP	Data

AH Packet Format before AH processing, in override mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest.	AH	TCP	Data

|<---------- authenticated except for mutable fields ---------->|

IP Packet Format after AH processing, in override mode.

Original Ipv6 header	Extension headers (if present)	TCP	Data

ESP Packet Format before ESP processing, in rfc2460 mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment	ESP	Final destination	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

IP Packet Format after ESP processing, in rfc2460 mode.

Original Ipv6 header	Extension headers (if present)	TCP	Data

ESP Packet Format before ESP processing, in override mode.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest.	ESP	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

IP Packet Format after ESP processing, in override mode.

Original Ipv6 header	Extension headers (if present)	TCP	Data

ESP Packet Format before ESP processing, in override mode with Mobility Header.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest.	ESP	Mobilty Header	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

IP Packet Format after ESP processing, in override mode.

Original Ipv6 header	Extension headers (if present) ,mobility header	TCP	Data

Packet Format before AH processing, in override mode with Mobility Header.

Original IPv6 header	Hop-by-hop, dest, routing, fragment, final dest.	AH	Mobilty Header	TCP	Data

<---------- authenticated except for mutable fields of ipv6 header ---------->|

IP Packet Format after AH processing, in override mode.

Original Ipv6 header	Extension headers (if present) ,mobility header	TCP	Data

Tunnel Mode

For tunnel mode microcode does the following:

- Decrypts and authenticates the IPSec ESP packet, or authenticates the IPSec AH packet.
- Removes the ESP or AH header.
- Removes the outer IPv6 header and extension headers.
- Returns the original IPv6 Packet as output.

AH Packet Format before AH processing, in rfc2460 or override mode.

New IPv6 header	New Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest.	AH	Original IPv6 header	Original Ext Hdrs: Hop-by-hop, dest., routing, fragment, final dest, Mobility header	TCP	Data

|<---------- authenticated except for mutable fields in new IPv6 header ---------->|

IP Packet Format after AH processing, in rfc2460 or override mode.

Original Ipv6 header	Extension headers (if present) , mobility header.	TCP	Data

ESP Packet Format before ESP processing, in rfc2460 or override mode.

New IPv6 header	New Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest.	ESP	Original IPv6 header	Orig. Ext Hdrs: Hop-by-hop, dest, routing, fragment, final dest, mobility header.	TCP	Data	ESP Trailer	ESP Auth

|<---------- encrypted ---------->|
|<---------- authenticated ---------->|

IP Packet Format after ESP processing, in rfc2460 or override mode.

Original Ipv6 header	Extension headers (if present), mobility header.	TCP	Data

Header size limitations

In transport or tunnel mode maximum extension header size is 208 bytes. If the extension header length is more than 208 bytes, Nitrox-N3 will throw an exception " ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG (50).

Inbound exceptions

- If IPv6 processing encounters multiple hop-by-hop extension header or not the very first extension header then Nitrox-N3 will throw an exception "ERROP__IPV6_HOP_BY_HOP_ERROR (51)".
- In Transport mode, If IPv6 extenstion header length exceeds 208 bytes then Nitrox-N3 will throw an exception "ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG (50)".
- In Tunnel mode, If IPv6 extension header length exceeds 208 bytes then Nitrox-N3 will throw an exception "ERROP__IPV6_EXTENSION_HEADERS_TOO_BIG(50)".
- If IPv6 routing header's segments left field is not equal to zero then Nitrox-N3 will throw an exception "ERROP__IPV6_DECRYPT_RH_SEGS_LEFT_ERROR (52)".
- If ipv6 mobiliry extension header is present in the tunnel mode in the outer IPV6 header of the inbound packet , then Nitrox_N3 throws an exception "ERROP__IPV6_EXTENSION_HEADER_BAD (55)" .

Additional Specifications

- If a IPv6 packet contains only destination header, Nitrox-N3 IPv6 rfc2460 mode will insert it after AH and IPv6 override mode will insert it before AH.
- If SG mode, The first scatter element size of each fragment must be greater than the length of (IPv6 Base header + IPv6 Extension Headers + AH/ESP header length).
- Minimum Fragmentation size limit is moved from 120 to 344 to consider the extension headers in the input packet. If parma1[10] is set i.e Minimum_Fragment_size is set then 120 Bytes otherwise 344 Bytes.

For Ipv6 extension headers parma1[10] must not set..

- Routing header with Header preservation is not supported.

4.7 IPv4 Options Header

Pre-Fragmentation is only supported in Tunnel mode. Therefore fragmentation will be done only for the inner header, and packets get fragmented before IPSEC processing. If any option does not carry forward beyond the first fragment, microcode will replace the option header with an Option NOP as per description above.

4.8 UDP Enapsulation (NAT Traversal)

Does incremental checksum on transport protocols (TCP, UDP) in TRANSPORT mode.
A problem occurs with IPSec encrypted packets crossing NAT devices. The IPsec Authentication Header (AH) protects entire IP packets, including IP headers, from modification. NAT modifies the IP header, causing an inherent incompatibility. The IPsec Encapsulating Security Payload (ESP) encrypts IP packets. NAT cannot modify TCP and UDP ports when these values are encrypted. NAT is therefore incompatible with ESP. ESP in tunnel mode can sometimes get through static NAT.

The solution for this problem is UDP Encapsulation, or NAT Traversal. UDP Encapsulation wraps an IPsec packet inside a UDP/IP header. This allows NAT to function, without modifying the encapsulated IPsec packet. UDP encapsulation is used only on ESP packets.

NAT traversal is provided by the UDP encapsulation feature. Transport payload checksum is recomputed using the incremental checksum method. The original source and destination address are required for this operation. The checksum correction will be done on an incoming packet's L4 protcol (TCP or UDP). If IPv6 address, then calculate the partial checksum sum (16-bit sum with carry's added) and place in the IPv4 address area. Be sure to clear the upper 16-bits.
Transport Mode - Perform incremental update of payload checksum if inner protocol is TCP or UDP.
Tunnel Mode - No requirements

Chapter 5 Boot Microcode Instruction Set

Provides a detailed reference description of the NITROX III Boot Microcode instruction set. These instructions can be used to load the WLAN microcode and various runtime data structures into the NITROX III chip during device initialization. This instruction set may also provide various device debug instructions that may be used during system development, bring up, and/or troubleshooting.

5.1 Microcode Initialization Instructions

Opcode BOOT_INIT

This must be the first opcode issued following the loading of the boot microcode. If this is not the first instruction perceived by chip, it will assert a general interrupt. This instruction performs the set of operations below, using input values specified in the input data structure at Dptr. Refer to section **3.4 Boot Initialization Parameters** for additional details on each of the values set below.

- Loads the boot constants array into the register file of the core currently executing this instruction. This array is then tested for correct layout and content. If the test does not pass, the value 0x0123456789abcdef is returned at Rptr, and a General Interrupt is asserted. This is to help with endian issues that might occur during intial bringup.
- Sets the Core Watchdog Timer value.
- Writes the error address (ERROR_ADDR) to internal NITROX SRAM.
- Stores the core scratchpad base address and size in internal NITROX SRAM. Only used for IPSEC microcode. Can be set to 0 for non-IPSec microcode
- Stores context window base address and SPI key constant values in internal SRAM. Only used for IPSEC microcode. Can be set to 0 for non-IPSec microcode
- Initializes the Plus-Mode Instruction Queue (PIQ) base address and size registers in NITROX Internal SRAM (PIQ also referred to as Control Transfer Point, or CTP).

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	BOOT_INIT (0x0000)
• Param1[15:00]:	PLUS Mode Instruction *queue_size* Set to number of 32-byte instruction structures
• Param2[15:00]:	Core scratchpad_size in bytes
• Dlen[15:00]:	Total *size* (in bytes) of input data structure pointed to by Dptr. *Size* = (40 + 8 * *const_len* bytes), where *const_len* is number of bytes in constants array.

Input Data (Dptr):

- Watchdog Timeout Value (4 bytes)
- SPI Key (4 bytes)
- Context Window Base Address (8 bytes)
- Error Address (8 bytes)
- Core Scratchpad Base Address (8 bytes)
- PlusMode Inst Queue (PIQ) base address (8 bytes)
- Constants (*const_len* bytes)

Output Data (Rptr):

- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode BOOT_SETUP_CONST

This instruction copies a 64-bit address located at Dptr, to the NITROX internal SRAM address specified in param2.

When any MAIN microcode first starts executing (when first enabled by setting corresponding bit in CORE_ENABLE register), it will read the constants address from internal SRAM at the address specified in Param2. It will then transfer constants array data from host memory at this address, into its own internal register file. The constants are an array of 8-byte values, with commonly used values such as 0x00000000 0000000, 0x5c5c5c5c 5c5c5c5c, or 0x36363636 36363636. These constants are used in various operations such HMAC calculation, checksum, zero-padding, etc.

The internal SRAM address that is used in param2 must match the corresponding value in the microcode binary file. It should be a 16-bit address field stored in a 64-bit field.

If a core detects that constants were not loaded properly, it will assert a General Interrupt.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	BOOT_SETUP_CONST (0x0016)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Internal SRAM address Set to value obtained from MAIN microcode binary file
• Dlen[15:00]:	Total *size* (in bytes) of input data structure pointed to by Dptr. Set to 8.

Input Data (Dptr):

- Host system memory address pointing to constants array (8 bytes)

Output Data (Rptr):

- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

No context data used in this instruction. Set Cptr to 0.

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

5.2 Debug Instructions

Opcode READ_INTERNAL_DATAPATH

Reads data from an internal hardware datapath register inside the chip. This register and the associated data are for debug purposes only, and are not specified the NITROX III Hardware Manual. Therefore this instruction is reserved for Cavium use only.

If Param2 is set to 0 in the instruction, the ***MICROCODE GENERAL INTERRUPT*** bit will be set in the INTERRUPT STATUS register, and if enabled in the INTERRUPT ENABLE register, an interrupt will be asserted. Refer to NITROX III Hardware Manual for more information on CORE ERROR STATUS Register.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	READ_INTERNAL_DATAPATH (0x1101)
• Param1[15:00]:	Reserved. Range = [0 – 255]
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	Reserved. Set to 0

Input Data (Dptr):
No input data used in this instruction. Set Dptr to 0

Output Data (Rptr):
- Datapath register information. (*8* bytes)
- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode CHECK_ENDIAN

Returns known sequence of bytes; specifically, the MD5 Hash Init pad value 0x0123456789abcdef. This can be used to check the endian of the return data to verify that DMA byte alignment is configured correctly. Byte 0x01 should be at byte address 0x0, 0x23 at byte address 1, etc.

Instruction Syntax :

OpCode/Params:

• Opcode[15:00]:	CHECK_ENDIAN (0x1201)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	Reserved. Set to 0

Input Data (Dptr):
No input data used in this instruction. Set Dptr to 0

Output Data (Rptr):

- 0x0123456789ABCDEF (8 bytes)
- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode ECHO_CMD

Reads data from Dptr and writes it back to Rptr. Can be used to verify DMA read and write operations are configured and performing correctly.

Instruction Syntax :

OpCode/Params:

• Opcode[15:00]:	ECHO_CMD (0x1301)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	Total *size* (in bytes) of input data structure pointed to by Dptr. Valid range = 1 to 880 (0x370)

Input Data (Dptr):

- User Data (*size* bytes)

Output Data (Rptr):

- User Data (*size* bytes)
- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode ROLL_CALL

Returns the Core ID plus an 8-byte user-defined data input. Used to ID which core is executing this instruction.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	ROLL_CALL (0x1401)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• [15:00]:	Total *size* (in bytes) of input data structure pointed to by Dptr. Set to 8.

Input Data (Dptr):

- User Defined Data (*8* bytes)

Output Data (Rptr):

- User Defined Data (*8* bytes)
- Core ID (1 byte)
- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Chapter 6 IPsec/IKE Microcode Instruction Set:

A detailed description of each NITROX III instruction. These instructions are used to implement the IPsec and IKE protocols.

6.1 IPSec Packet Processing Instruction Group

Opcode WRITE_IPSEC_OUTBOUND_SA

This instruction initializes an IPSec outbound SA context data structure. It must be called prior to calling outbound IPSec packet processing instructions. The instruction reads an input structure from host memory at Dptr, pre-processes various fields, and writes the resulting IPSec outbound SA Context data structure in host memory at Cptr.

NOTE: Irrespective of the inbound or outbound SA size defined below, the microcode will write 512 bytes of context structure to Rptr. Also, Inbound and Outbound IPSec SA's have a different format/structure.

For additional information and user guidance for this Opcode, refer to Additional User Guidelines section at end of this Opcode specification.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	WRITE_IPSEC_OUTBOUND_SA (0x4014)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- Input Data Word [0:0] // SA Control Word
  - o Control (2 bytes) // see description below
  - o Extended-Control (1 byte) // see description below
  - o Inter-frag pad size (1 byte) // When fragmenting packets during IPSec
    // outbound packet processing, this field specifies the number of
    // 8-byte words of zero-padding to insert between the fragments,
    // in that instruction's output data structure. This pad is intended to
    // allow for later insertion of additional packet information by the
    // host processor. For example, L2 packet header information.
  - o SPI (4 bytes) // IPSec Security Parameter Index value inserted into all
    // outbound packets

- Input Data Word [1:4] // Encryption Key
  - o Cipher key (32 bytes) // Packet encryption key – Key size will vary with algorithm
    // (e.g. NULL, AES, DES, 3DES). Unused space must be set to 0.

- Input Data Word [5:7] // Authentication Key (SHA1, MD5)
  - o Hmac Key (24 bytes) // Packet authentication key. Key size will vary with algorithm
    // used. (e.g. NULL, MD5, SHA-1). Unused space must be set to
    // 0. SHA2 or AES-XCBC authentication keys specified below.

- Input Data Word [8:8] // Next SA Pointer
  If (*Next_SA* ==1) // Specified in Control<15>. See description below
  - o Next SA Pointer (8 bytes) // Pointer to location of next SA Context in this bundle

  Else
  - o Reserved (8 bytes) – set to 0

- Input Data Word [9:40] // IP Tunnel Template and UDP encapsulation port #'s
  If (*Outer_IP_Version* == IPv4) { // Specified in Control<2>. See description below
  If (*IPSec_Mode* == Tunnel) // Specified in Control<4>. See description below
  - o Outer IPv4 header template (20 bytes)
    // See below for description of template structure.

  Else
  - o Reserved. Set to 0 (20 bytes)

  }

  If (*Outer_IP_Version* == IPv6) { // Specified in Control<2>. See description below
  If (*IPSec_Mode* == Tunnel) // Specified in Control<4>. See description below
  - o Outer IPv6 header template (40 + IPv6_Extension_Header_Length bytes).
    // See below for description of template structure

  Else
  - o Reserved. Set to 0 (40 bytes)

  }

  - o UDP encap src port (2 bytes) // Used in “UDP hole punching”, a form of NAT traversal. UDP
    // encapsulation is selected via context Control<6>. If not used,
    // set to 0

  - o UDP encap dest port (2 bytes) // Used in “UDP hole punching”, a form of NAT traversal. UDP
    // encapsulation is selected via context Control<6>. If not used,
    // set to 0.

  - o Reserved. Set to 0 ((256 - UDP port size(4) - template size) bytes)
    // 256B allocated here to support variable template size.
    // Template size(Ipv4) = 20
    // Template size(Ipv6) = 40 + *IPv6_Extension_Header_Length*

- Input Data Word [41:56] // Authentication Key (SHA2 or AESXCBC)

If (*Authentication_Type*==SHA2*) { // Specified by Control<12:11> and Extended-Control<3:1>
If (Authentication_Type == SHA256) { // See description below.
- SHA2 Keys (64 bytes)
- Reserved. Set to 0 (64 bytes) // IV embedded in microcode constant's array for SHA256.

}
Else If (Authentication_Type == SHA384 or SHA 512) {
- SHA2 Keys (64 bytes)
- IV (64 bytes) // Refer to **Appendix C SHA Hash Initial Values**

}
Else
- Reserved. Set to 0 (128 bytes)

- Input Data Word [57:58] // Nonce & Explicit IV

If (*Explicit_IV_Mode == 1*) // Specified by Extended_Control<6>
If (*AES_CTR_Mode* == 1 or *AES_GCM_Mode* == 1) {
// Specified by Extended_Control<5:4>
- Nonce MSB (4 bytes) // Used in CTR & GCM/GMAC modes
- Explicit IV (8 Bytes)
- Counter IV. Set to 0 (4 bytes) // Used in CTR & GCM/GMAC modes

}
Else {
- Explicit IV MSB (for AES or TDES CBC, 8 bytes)
- Explicit IV LSB (for AES CBC, else reserved, set to 0. 8 bytes)

}

Else // *Explicit_IV_Mode* == 0
If (*AES_CTR_Mode* == 1 or *AES_GCM_Mode* ==1){
// Specified by Extended_Control<5:4>
- Nonce MSB (4 bytes) // Used in CTR & GCM/GMAC modes
- Reserved. Set to 0 (8 bytes)
- Counter IV. Set to 0 (4 bytes) // Used in CTR & GCM/GMAC modes

}
Else // Encryption_Type == TDES, or AES_CBC
- Reserved, set to 0 (16 bytes)

- Input Data Word [59:59] // Extended Sequence Number

If(*ESN_Mode*==1)
- Extended Sequence Number (4 bytes) // Lower 32 bits provided with Input data during calls
// to packet processing instructions
- Reserved, set to 0 (4 bytes)

Else
- Reserved (8 bytes)

- Input Data Word [60:60]:
  - Reserved, set to 0 (8 bytes)

Control[15:0]

Bit	Description	Value
0	*SA_Valid*	0: Invalid Context 1: Valid Context
1	*SA_Direction*	Set to 1 for Outbound
2	*Outer_IP_Version*	0: IPv4 1: IPv6
3	*Inner_IP_Version* (Tunnel_Mode only. Ignored for Transport Mode)	0: IPv4 1: IPv6
4	*IPSec_Mode*	0: Transport Mode 1: Tunnel Mode
5	*IPSec_Protocol*	0: AH 1: ESP
6	*Encapsulation_Type*	0: None 1: UDP
7	Reserved	Set to 0
10:8	*Encryption_Type*	000b: Null 001b: DES (CBC mode) 010b: DES3 (CBC mode) 011b: AES128. See Extended-Control[5:4] for mode 100b: AES192. See Extended-Control[5:4] for mode 101b: AES256. See Extended-Control[5:4] for mode Else undefined. Do not use.
12:11	*Authentication_Type*	00b: Null 01b: HMAC96-MD5 10b: HMAC96-SHA-1 11b: *Extended_Auth_Type*: Auth type specified in Extended Control<3:1>
13	*Copy_DF*	0: Set DF bit in outer IP header to value in header template in SA Context 1: Set DF bit in outer IP header to value copied from inner IP header
14	*Fragmentation_Type*	0: Post (after) encryption 1: Pre (before) encryption
15	*Next_SA*	0: Current SA is final 1: Read Next SA

Extended-Control[7:0]

Bit	Description	Value
7	Reserved	Set to 0
6	*Explicit_IV_Mode*	0: Disabled. IV used is Nitrox-generated random number. 1: Enabled. IV used is specified in outbound SA context memory Note: Typically enabled for testing purposes only. During Normal processing Nitrox will auto-generate a random IV.
5	*AES_CTR_Mode*	When Encryption_Type is set to AES in Control<10:8> 0: AES_CBC or AES_GCM used in transform. 1: AES_CTR used in transform. AES key size also specified in Control<10:8>. AES_GCM_Mode and AES_CTR_Mode select bits are mutually exclusive. Do not set both to 1.
4	*AES_GCM_Mode*	When *Encryption_Type* is set to AES in Control<10:8> 0: AES_CBC or AES_CTR used in transform 1: AES_GCM used in transform AES key size also specified in Control<10:8>. AES_GCM_Mode and AES_CTR_Mode select bits are mutually exclusive. Do not set both to 1.
3:1	*Extended_Auth_Type*	001 – AES-GMAC authentication algorithm 010 – HMAC-SHA2-256-128 authentication algorithm 011 – HMAC-SHA2-384-192 authentication algorithm 100 – HMAC-SHA2-512-256 authentication algorithm 101 – AES-XCBC-128 authentication algorithm. Else Reserved. Do not use.
0	*ESN_Mode*	Enables support for Extended Sequence Numbers compliant to RFC4303. 0: Normal Sequence Number used 1: Extended Sequence Number used. High-order 32 bits of ESN are copied into the ESP trailer, and ICV is calculated. For AES_GCM and AES_GMAC, prepare the AAD data with 64-bit sequence numbers.

Output Data (Rptr):

- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

- Context Structure Word [0:0]: // SA Control Word

- Control (2 bytes) // Copied from Dptr
- Extended-Control (1 byte) // Copied from Dptr
- Inter-frag pad size (1 byte) // Copied from Dptr
- SPI (4 bytes) – copied from Dptr

- Context Structure Word [1:4]: // Encryption Key
  - Cipher key (32 bytes) // Copied from Dptr

- Context Structure Word [5:10]: // Authentication Key (SHA1, MD5)
  - Hmac key Material (48 bytes) // Preprocessed HMAC Key

- Context Structure Word [11:11]: // Next SA Pointer

If (Next SA == 1)
- Next SA Pointer (8 bytes) // Copied from Dptr

Else
- Reserved (8 bytes) // Copied from Dptr

- Context Structure Word [12:43]: // Tunnel Header Template and UDP port

If (*Outer_IP_Version* == IPv4) {
If (*IPSec_Mode* == Tunnel)
- Outer IPv4 header template (20 bytes) // Copied from Dptr

Else
- Reserved (20 bytes) // Copied from Dptr

}

If (*Outer_IP_Version* == IPv6) {
If (*IPSec_Mode* == Tunnel)
- Outer IPv6 header template (40 + IPv6_Etension_Header_Length bytes) // Copied from Dptr

Else
- Reserved (40 bytes) // Copied from Dptr

}
- UDP encap src port (2 bytes) // Copied from Dptr
- UDP encap dest port (2 bytes) // Copied from Dptr

If (template size + UDP_ports (4 bytes) < 256)
- Reserved (256 - *template_size*+UDP port size (4 bytes))
  // Where *template_size* is the size of IP Tunnel Template
  // described above, for either IPv4 or IPv6.

- Context Structure Word [44:59]: // Authentication Key (SHA2 or AESXCBC)

If(*Authentication_Type*==SHA2*) { // Specified by combination of Control<12:11> and
// Extended-Control<3:1>
- OPAD partial hash (64 bytes) // Preprocessed partial hash
- IPAD partial hash (64 bytes) // Preprocessed partial hash

}
Else If (*Authentication_Type*==AESXCBC) {
- AES-XCBC-IV (16 bytes) // Calculated & written by Nitrox

- Calculated AES-XCBC-128 K1 (16 bytes) // Refer to IETF RFC 3566 for description
- Calculated AES-XCBC-128 K2 (16 bytes) // of IV and values K1, K2, and K3
- Calculated AES-XCBC-128 K3 (16 bytes)

}
Else
- Reserved (128 bytes)

- Context Structure Word [60:61]: // Nonce & Explicit IV

If (*Explicit_IV_Mode == 1*) // Specified by Extended_Control<6>
If(*AES_CTR_Mode ==1* or *AES_GCM_Mode == 1*) {
// Specified by Extended_Control<5:4>
- Nonce (4 bytes) // Copied from Dptr
- Explicit IV (8 Bytes) // Copied from Dptr
- Counter Initial Value (4 bytes) // Copied from Dptr

}
Else {
- Explicit IV MSB (8 bytes) // Copied from Dptr
- Explicit IV LSB (8 bytes) // Copied from Dptr

}

Else // *Explicit_IV_Mode* == 0
If (AES_CTR_Mode == 1 or AES_GCM_Mode ==1){
// Specified by Extended_Control<5:4>
- Nonce(4 bytes) // Copied from Dptr
- Reserved (8 bytes) // Copied from Dptr
- Counter Initial Value (4 bytes) // Copied from Dptr

}
Else // Not CTR or GCM mode
- Reserved (16 bytes)

- Context Structure Word [62:62]: // Extended Sequence Number

If (*ESN_Mode*==1)
- Extended Sequence Number (4 bytes) // Lower 32 bits provided with Input data during calls
// to packet processing instructions
- Reserved (4 bytes)

Else
- Reserved (8 bytes)

- Context Structure Word [63:63]: // Reserved Field
  - Reserved (8 bytes)

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x18	BAD_AUTH_TYPE	Auth type is other than SHA1, MD5, SHA2, GCM/GMAC or AES-XCBC

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

Tunnel Mode IP Header Template

When Tunnel Mode IPSec is selected, NITROX will automatically insert the Tunnel Header into each outbound packet, using the header template configured in the SA context during this instruction. The format of the template in the outbound SA context is as follows.

IPv4 Template:

IP version=4, hlen=5 to 0xf, service type, DF bit, TTL, Next Header (ESP, AH, UDP), tunnel source address, tunnel destination address

IPv6 Template:

IP version=6, flow label, hop limit, Next Header (ESP, AH, UDP, HOP*, DST *or ROUTE*), tunnel source address, tunnel destination address.
IPv6 Extension Headers: HOP/ROUTE/DST
HOP: Next Header (ESP, AH, DST, ROUTE,UDP), Extension Header Length, Options
ROUTE: Next Header (ESP, AH, DST, ROUTE, UDP), Extension Header Length, Routing Type, Segments Left, Destination Adress[1], Destination Address[2]
DST: Next Header (ESP, AH, DST, ROUTE, UDP), Extension Header Length, Options

The Next Header value in the template must use the appropriate protocol number. E.g. AH= 51,UDP =17, etc. Using the wrong protocol will cause errors that will not be detected until inbound processing at opposite end of the tunnel.

If the desired behavior is to set or clear the DF bit in the tunnel header, it should be set or cleared appropriately in the header template.

NITROX automatically calculates and inserts into the final tunnel header, the total packet length, traffic class and fragmentation header (IPv6), fragmentation flags, id, offsets, and checksum (IPv4).

Explicit IV

In some instances it may be desirable for the user to specify the IV to use for outbound packet processing, instead of having that IV generated automatically by Nitrox microcode. For example, when testing or certifying an algorithm implementation within the device the user must be able to provide an explicit IV in order to compare encrypted output data to a known value. To enable this capability, the user can write an explicit IV to context memory for later use by the Opcode ENCRYPT_IPSEC_OUTBOUND_PACKET.

Extended Sequence Numbers:

ESN is supported by setting the *ESN_Mode* bit in the Extended_Control field of the SA context, and configuring the high-order 32 bits of the sequence number in input data. This instruction will then copy these high-order bits of the sequence number from the input data to the appropriate field in the context data structure, for later use by the associated outbound packet processing instructions. Refer to **Opcode PROCESS_OUTBOUND_IPSEC_PACKET** for additional information about sequence numbers and extended sequence number use during packet processing.

Opcode WRITE_IPSEC_INBOUND_SA

This instruction initializes an IPSec inbound SA context data structure. It must be called prior to calling inbound IPSec packet processing instructions. The instruction reads an input structure from host memory at Dptr, pre-processes various fields, and writes the resulting IPSec inbound SA Context data structure in host memory at Cptr.

NOTE: Inbound and Outbound IPSec SA's have a different format/structure. Irrespective of the inbound/outbound SA size, the microcode will write the 512 bytes of context to Rptr.

For additional information and user guidance for this Opcode, refer to Additional User Guidelines section at end of this Opcode specification.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	WRITE_IPSEC_INBOUND_SA (0x2014)
• Param1[15:00]:	Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- Input Data Word [0:0] // SA Control Word
  - o Control (2 bytes) // See below for description
  - o Expected Protocol (1 byte) // Protocol field used for IPSec inbound selector checking.
  - o Extended-Control (1 byte) // See below for description
  - o SPI (4 bytes) // SPI value used in IPSec inbound selector checking
- Input Data Word [1:4] // Encryption Key
  - o Cipher key (32 bytes) // Packet decryption key – Key size will vary with algorithm
    // (e.g. NULL, AES, DES, 3DES). Unused space must be set to 0.
- Input Data Word [5:7] // Authentication Key (SHA1, MD5)
  - o Hmac Key (24 bytes) // Packet authentication key. Key size will vary with algorithm
    // used. (e.g. NULL, MD5, SHA-1). Unused space must be set to
    // 0. SHA2 or AES-XCBC authentication keys specified below.
- Input Data Word [8:8] // Next SA Pointer
  If (*Next_SA* ==1) // Specified in Control<15>. See description below
  - o Next SA Pointer (8 bytes) // Pointer to location of next SA Context in this bundle

  Else
  - o Reserved. Set to 0 (8 bytes)

- Input Data Word [9:10] // Inbound Selector Checks: UDP Src/Dest Addr & Port
  - UDP Encap IP Src Address (4 bytes) // Used in "UDP hole punching", a form of NAT traversal.
  - UDP Encap IP Dest Address (4 bytes) // UDP encapsulation is selected via context Control<6>.
  - UDP Src Port (4 bytes)
  - UDP Dest Port (4 bytes)

- Input Data Word [11:18] // Inbound Selector Check - Src/Dest Addresses
  If (*Inner_IP_Version* == IPv4) { // Specified in Control<3>. See description below.
  - Expected IP Source Addr (8 bytes)
  - Expected IP Destination Addr (8 bytes)
  - Reserved. Set to 0 (48 bytes)

  }
  Else { // *Inner_IP_Version* == IPv6
  - Expected IP Source Addr (32 bytes)
  - Expected IP Destination Addr (32 bytes)

  }

- Input Data Word [19:19] // Nonce & Explicit IV
  If (*AES_CTR_Mode* == 1 or AES_GCM_Mode == 1) {
  // Specified by Extended_Control<5:4>. See description
  // below.
  - Nonce MSB (4 bytes)
  - Reserved. Set to 0 (4 bytes)

  }
  Else
  - Reserved. Set to 0 (8 bytes)

- Input Data Word [20:21] // Extended Sequence Number
  If (*ESN_Mode*==1) // Specified by Extended_Control <0>. See description
  // below.
  - Extended Sequence Number (4 bytes) // Lower 32 bits provided with Input data during calls
  // to packet processing instructions

  Else
  - Reserved. Set to 0 (4 bytes)

  - Reserved. Set to 0 (12 bytes)

- Input Data Word [22:37] // Authentication Key (SHA2, AES_XCBC)
  If (*Authentication_Type*==SHA2*) { // Specified by Control<12:11> and Extended-Control<3:1>
  If (Authentication_Type == SHA256) { // See description below.
  - SHA2 Keys (64 bytes)
  - Reserved. Set to 0 (64 bytes) // IV embedded in microcode constant's array for SHA256.

  }
  Else If (Authentication_Type == SHA384 or SHA 512) {
  - SHA2 Keys (64 bytes)
  - IV (64 bytes) // Refer to **Appendix C SHA Hash Initial Values**

  }

}
Else If (*Authentication_Type*==AESXCBC)
o Reserved. Set to 0 (128 Bytes)

Control[15:0]

Bit	Description	Value
0	*SA_Valid*	0: Invalid context 1: Valid context
1	*SA_Direction*	Set to 0 for Inbound
2	*Outer_IP_Version*	0: IPv4 1: IPv6
3	*Inner_IP_Version* (Tunnel Mode only. Ignored for transport mode)	0: IPv4 1: IPv6
4	*IPSec_Mode*	0: Transport Mode 1: Tunnel Mode
5	*IPSec_Protocol*	0: AH 1: ESP
6	*Encapsulation_Type*	0: NULL 1: UDP
7	Reserved	Set to 0
10:8	*Encryption_Type*	000b: Null 001b: DES (CBC mode) 010b: DES3 (CBC mode) 011b: AES128. See Extended-Control[5 :4] for mode 100b: AES192. See Extended-Control[5 :4] for mode 101b: AES256. See Extended-Control[5 :4] for mode else undefined
12:11	*Authentication_Type*	00b: Null 01b: HMAC96-MD5 10b: HMAC96-SHA-1 11b: *Extended_Auth_Type*: Auth type specified in Extended-Control<3:1>
13	*Selector_Check_Enable*	0: Disable selector checking on inbound IP packet 1: Verify source addr/port and dest addr/port on inbound IP packet
14	*Protocol_Selector_Disable*	0: Verify Protocol field of inbound IP packet (only when *Selector_Check_Enable* is set) 1: Disable selector check of Protocol field on inbound IP packet
15	*Next_SA*	0: Current SA is final 1: Read Next SA

Extended-Control[7:0]

Bit	Description	Value
[7:6]	Reserved	Must be set to Zero
5	*AES_CTR_Mode*	When *Encryption_Type* is set to AES in Control<10:8> 0: AES_CBC or AES_GCM used in transform. 1: AES_CTR used in transform. Key size also specified in Control<10:8>. *AES_GCM_Mode* and *AES_CTR_Mode* are mutually exclusive. Do not set both to 1.
4	*AES_GCM_Mode*	When *Encryption_Type* is set to AES in Control<10:8> 0: AES_CBC or AES_CTR used in transform 1: AES_GCM used in transform AES key size also specified in Control<10:8>. AES_GCM_Mode and AES_CTR_Mode select bits are mutually exclusive. Do not set both to 1.
3:1	*Extended_Auth_Type*	001 – AES-GMAC authentication algorithm 010 – HMAC-SHA2-256-128 authentication algorithm 011 – HMAC-SHA2-384-192 authentication algorithm 100 – HMAC-SHA2-512-256 authentication algorithm 101 – AES-XCBC-128 authentication algorithm. Else Reserved. Do not use.
0	*ESN_Mode*	Enables support for Extended Sequence Numbers compliant to RFC4303. 0: Normal Sequence Number used 1: Extended Sequence Number used. High-order 32 bits of ESN are copied into the ESP trailer, and ICV is calculated. For AES_GCM and AES_GMAC, prepare the AAD data with 64-bit sequence numbers.

Output Data (Rptr):

- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

- Context Structure Word [0:0]: // SA Control Word
  - Control (2 bytes) // Copied from Dptr
  - Expected Protocol (1 byte) // Copied from Dptr
  - Extended-Control (1 byte) // Copied from Dptr
  - SPI (4 bytes) – copied from Dptr // Copied from Dptr
- Context Structure Word [1:4]: // Encryption Key
  - Cipher key (32 bytes) // Copied from Dptr
- Context Structure Word [5:10]: // Authentication Key (SHA1, MD5)
  - Hmac key Material (48 bytes) // Preprocessed HMAC Key

- Context Structure Word [11:11]: // Next SA Pointer

If (*Next_SA* == 1)
  - Next SA Pointer (8 bytes) // Copied from Dptr

Else
  - Reserved (8 bytes) // Copied from Dptr

- Context Structure Word [12:13]: // UDP port & IP address selector checks
  - UDP Encap IP Src address (4 bytes) // Copied from Dptr
  - UDP Encap IP Dest address (4 bytes) // Copied from Dptr
  - UDP Src Port (4 bytes) // Copied from Dptr
  - UDP Dest Port (4 bytes) // Copied from Dptr

- Context Structure Word [14:21]: // IPSec tunnel header selector checks

If (*Inner_IP_Version* == IPv4) {
  - IP Src Addr (8 bytes) // Copied from Dptr
  - IP Dest Addr (8 bytes) // Copied from Dptr
  - Reserved (48 bytes) // Copied from Dptr

}
Else If (*Inner_IP_Version* == IPv6) {
  - IP Src Addr (32 bytes) // Copied from Dptr
  - IP Dest Addr (32 bytes) // Copied from Dptr

}

- Context Structure Word [22:22]: // Nonce & Explicit IV

If (*AES_CTR_Mode* == 1 or *AES_GCM_Mode* == 1) {
  - Nonce MSB (4 bytes) // Copied from Dptr
  - Reserved (4 bytes) // Copied from Dptr

}
Else
  - Reserved (8 bytes) // Copied from Dptr

- Context Structure Word [23:24]: // Extended Sequence Number

If (*ESN_Mode*==1)
  - Extended Sequence Number (4 bytes) // Lower 32 bits provided with Input data during calls

Else // to packet processing instructions
  - Reserved (4 bytes)

  - Reserved (12 bytes) // Copied from Dptr

- Context Structure Word [25:40]: // Extended Sequence Number

If(*Authentication_Type*==SHA2*) { // Specified by combination of Control<12:11>
// and Extended-Control<3:1>
  - Calculated OPAD (64 bytes) // Preprocessed partial hash
  - Calculated IPAD (64 bytes) // Preprocessed partial hash

}
Else If (*Authentication_Type*==AESXCBC) {
  - AES-XCBC-IV (16 bytes) // Calculated & written by Nitrox
  - Calculated AES-XCBC-128 K1 (16 bytes) // Refer to IETF RFC 3566 for description

- Calculated AES-XCBC-128 K2 (16 bytes) // of IV and values K1, K2, and K3
- Calculated AES-XCBC-128 K3 (16 bytes)

}

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x18	BAD_AUTH_TYPE	Auth type is other than SHA1, MD5, SHA2, GCM/GMAC or AESXCBC.

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

Extended Sequence Numbers:

ESN is supported by setting the *ESN_Mode* bit in the Extended_Control field of the SA context, and configuring the high-order 32 bits of the sequence number in input data. This instruction will then copy these high-order bits of the sequence number from the input data to the appropriate field in the context data structure, for later use by the associated inbound packet processing instructions. Refer to **Opcode PROCESS_INBOUND_IPSEC_PACKET** for additional information about sequence numbers and extended sequence number use during packet processing.

Opcode ERASE_CONTEXT

Zero out Context memory. Context Size < 256 bytes.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	ERASE_CONTEXT (0x0014)
• Param1[15:00]:	*Size* – amount of context memory area to erase (in 8 bytes words. (Context Size >> 3) -1). Range = 1 to 32 (8 byte words)
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	Reserved. Set to 0

Input Data (Dptr):

No input data used in this instruction. Set Dptr to 0

Output Data (Rptr):

- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

- Zeroized Memory (Size bytes)

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode PROCESS_OUTBOUND_IPSEC_PACKET

Performs complete IPSec outbound packet processing. Input to this instruction is a complete IP packet (header and payload). Output is a complete IPSec packet, ready for transport.

IPSec outbound packet processing includes encryption and/or authentication of IP packet, insertion of padding, insertion of IPSec (AH/ESP) header(s), including tunnel IP header if tunnel mode is used, and insertion of UDP encapsulation (if selected).

If selected, this instruction will fragment the input packet into multiple output packets. Fragmentation can be selected to be performed either before or after packet encryption.

If IPCOMP is required by the application, data compression must be performed on the packet payload and IPCOMP header applied prior to submitting the packet to this instruction for IPSec processing. NITROX does not implement data compression. Support of IPCOMP packets is only provided for Tunnel mode IPSec, with fragmentation disabled. If IPCOMP support is selected in combination with either Transport mode or Fragmentation, NITROX will return an error.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	PROCESS_OUTBOUND_IPSEC_PACKET (0x0011)
• Param1[15:00]:	<15>: Reserved. Set to 0<14>: *RFC/Override mode* **-** used in IPv6 Extension Header Support 0: RFC Mode 1: Override Mode <13>: *TFC_Dummy_Packet* 0: Perform Normal IPSec Packet Processing 1: Generate Traffic Flow Confidentiality (TFC) “Dummy Packet” as per RFC4303. <12>: *TFC_Pad_Enable* 0: Size field in Param2 specifies IPComp packet length 1: Size field in Param2 specifies TFC pad length. IPComp not supported when this bit is set. <11>: *Per_Packet_IV* 1: Use IV from Input 0: Use IV from Context if *Explicit_IV_Mode*, else generate internally <10>: *Minimum_Fragment_Size* 1: Minimum Fragment Size is 120B selected 0: Minimum Fragment Size is 344B selected *<9:4>:* Reserved – must be set to 0 <3:0>: *Frag_Num* – Specifies the number of fragments to break input IP packet into when creating output IPSec packet(s). 0x0 – no fragmentation

	0x1 – RESERVED. Do not use 0x2 – 2 fragments … 0xf – 15 fragments This number is calculated by host software, based on size of input IP packet and network MTU. *Frag_Num* must be selected such that Original IP Packet Length/*Frag_Num* > If (Min_Fragment_size == 1) 120B else 344B. Input packets will be fragmented into approximately equal-sized output IPSec packets.

• Param2[15:00]:	If (*TFC_Pad_Enable*==0) then
	IPCOMP_Size (bytes) – value includes combined size of IPCOMP header & payload. Valid range = 0 to 0xFFEC (2^16 – 20). A value of 0 specifies that the input data is an IP packet, not an IPCOMP packet.
	Else
	TFC pad length (bytes). IPComp not supported in this mode.

• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- ID (4 bytes) // Copy of *Identification* field of input IP packet (only
  // least-significant 2 bytes are used for IPv4. Set unused
  // bytes to 0)
- Sequence number (4 bytes)

If (*Per_Packet_IV* == 1) { // Controlled by Param1<11>

- IV (16 bytes) // 16B for AESCBC. 8B for DES/ AESCTR/AESGCM. Remaining 8B
  // must be 0. For AESCTR/AESGGM, 4B Salt must be provided in
  // Context Structure

}

If (*IPCOMP_Size* != 0) { // IPCOMP packet

- Reserved. Set to 0 (6 bytes)
- DF bit (1 byte) // Copy of DF bit in IP header of input packet. Legal values: 0x00 or 0x01
- TOS (1 byte) // Copy of TOS (Type of Service) byte from IP header of input IP packet

}

- IP packet to be IPSec encoded (variable length)

If (*TFC_Pad_Enable*==1)

- TFC Padding (length specified in Param2)

Output Data (Rptr):

- IPSec packet (*Rlen* bytes) // IP packet/fragment format and *Rlen* depend on IPSec mode and
  // options configured in Cptr structure. See below for *Rlen*
  // calculation
- Completion Code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
Context data structure for this instruction is initialized under NITROX control, via instruction WRITE_IPSEC_OUTBOUND_CONTEXT above. Refer to that instruction for a description of this Context data structure.

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x0d	BAD_SCATTER_LENGTH	Total scatter length is less than *rlen*
0x17	BAD_IP_VERSION	IP version is neither IPv4 or IPv6
0x1e	BAD_IPSEC_CONTEXT	Context information is invalid

0x26	BAD_FRAGMENT_SIZE	Depends on Minimum_Fragment_Size bit param1[10].
		If(Minimum_Fragment_size == 1)
		Fragment size is less than 120 bytes.
		Else
		Fragment size is less than 344 bytes.

0x29	BAD_INLINE_DATA	IPCOMP packet and Transport Mode and/or fragmentation is selected

0x2a	BAD_FRAG_SIZE_CONFIGURATION	Postfrag or prefrag,
		If (Minimum_Fragment_Size == 1) {
		Ipv6 packet has extension headers
		(or)
		Ipv4 packet has options
		}

0x2e	BAD_MIN_FRAG_SIZE_AUTH_SHA384_512	If (min_fragment_size == 1) i.e parma1[10] Auth = SHA384/SHA512.
0x32	IPV6_EXTENSION_HEADERS_TOO_BIG	IPv6 Extension Header length exceeds 320bytes in Transport mode (or) 248bytes in Tunnel mode

0x33	IPV6_HOP_BY_HOP_ERROR	HOP BY HOP Header is not the very first extension header or multiple occurrence of HOP-BY_HOP header.
0x35	IPV6_RH_LENGTH_ERROR	IPv6 Routing Header Length is not divisible by 2
0x36	IPV6_OUTBOUND_RH_COPY_ADDR_ERROR	IPv6 Routing Header with Header Preservation is not Supported
0x38	BAD_AUTH_TYPE_GCM_GMAC	*Authentication_Type* != 11b and *AES_GCM_Mode* == 1
0x39	BAD_ENCRYPT_TYPE_CTR_GCM	*Encryption_Type* == *DES or NULL and (*AES_CTR_Mode* ==1 or *AES_GCM_Mode* ==1)
0x3a	AH_NOT_SUPPORTED_WITH_GCM_GMAC_ SHA2	*IPSec_Protocol* == AH and Authentication_Type == AES_GMAC, SHA256, SHA384 or SHA512 Note: AH with SHA2 or GMAC not supported
0x3b	TFC_PADDING_WITH_PREFRAG_NOT_ SUPPORTED	TFC padding enabled and fragmentation type in SA context is specified as prefrag.

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

This instruction performs IPSec processing on outbound IP packets. i.e. it transforms IP packets into IPSec packets. The type of IPSec transform performed is selected by the user via the combination of instruction input parameters, and the configuration of a preexisting IPSec SA context data structure stored at Cptr. Therefore, before executing this instruction the user must initialize the associated SA context by executing opcode WRITE_IPSEC_OUTBOUND_SA, to select the desired IPSec processing options for this SA.

Rlen Calculation:

Define the following variables:

```
L: Input Packet Length (including IP header)
        If (IP version == IPv4)
                L = IPv4 packet length (bytes)
        Else // version == IPv6
                L = IPv6 packet length (bytes)
H: Transport Mode Header Length
        If (IPSec_Mode == tunnel)
                H = 0 (bytes)
        Else // (IPSec_Mode == transport)
                If ((version == IPv6)
                        H = 40 +  IPv6_Extension_Header_Length (bytes)
                Else // version == IPv4
                        H = 20 (bytes)
T: Tunnel Mode Header Length
        If (IPSec_Mode == transport)
                T = 0 (bytes)
        Else // (IPSec_Mode == tunnel)
                If (IP version == IPv6)
                        T = 40 + Outer IPv6_Extension_Header_Length (bytes)
                Else // (IP version == IPv4)
                        T = 20 (bytes)

E(h): ESP Header Length
        E(h) = 8 (bytes)

E(t): ESP Trailer Length
        E(t) = 2 (bytes)

A: AH Header Length
        A = 12 (bytes)
D: MAC Length
        If (Auth Type == NULL)
                D = 0 (bytes)
        Else if(Auth Type == MD5 or SHA)
                D = 12 (bytes)
        Else if (Auth Type == SHA256 || AES_GCM_flag == 1 || AES_GMAC_flag == 1)
                D = 16 (bytes)
```

Else if (Auth Type == SHA 384)
D = 24 (bytes)
Else if (Auth Type == SHA512)
D = 32 (bytes)
Else if (Auth Type == AESXCBC)
D = 12 (bytes)

I: IV Length
If (encrypt type == NULL)
I = 0 (bytes)
Else if (encrypt type == DES)
I = 8 (bytes)
Else // (encrypt type == AES)
I = 16 (bytes)

U: UDP encapsulation length
If (encaps type == UDP)
U = 8 (bytes)
Else // (encaps type == NULL)
U = 0 (bytes)

ENC_FUNC:
Encrypt == NULL: ROUNDUP4
Encrypt == DES: ROUNDUP8
Encrypt == AES: ROUNDUP16

Then:

For ESP:
Rlen = T + H + E(h) + I + ENC_FUNC(L - H + E(t)) + D + U

For AH:
Rlen = T + H + A + D + L

Note: For SA Bundles use the above rlen calucluation with outer SA supported cipher and authentication.

If (*Frag_Num* != 0), then:
For IPv4, *Rlen* = *Rlen*(L) + (*Frag_Num*-1)*(20) bytes.
For IPv6, *Rlen* = *RLen*(L) + (*Frag_Num*-1)*(40 + IPv6_Extension_Header_Length) bytes.

Where *Rlen*(L) = *Rlen* of the original, non-fragmented outbound packet from calculation above.

Example:
Input: 1500 byte IPv4 packet (L = 1500)
Output: IPSec packet, encoded with ESP(AES+SHA MAC), Tunnel Mode, no UDP encapsulation.

The resulting IPSec packet size, *Rlen*, will be:
Rlen(L) = T + H + E(h) + I + ENC_FUNC(L - H + E(t)) + D + U
= 20 + 0 + 8 + 16 + ROUNDUP16(1500 – 0 + 2) + 12 + 0
= 1560 bytes
NOTE:

- In case of IPv6 Extension Headers, special care is to be taken while generating first scatter element size of each scatter entry (each fragment). The first scatter element size of each fragment must be greater than the length of (IPv6 Base header + IPv6 Extension Headers + AH/ESP header length).
- When selecting fragmentation NITROX microcode requires the minimum fragment payload size for the first fragment depends on param1[10] bit, if (Minimum_Fragment_Size == 1) > 120 Bytes otherwise >=344 bytes.

Packet Fragmentation:

Fragmentation is selected on a packet-by-packet basis by entering the number of desired fragments for the current packet, in field *Frag_Num* (Param1[3:0]). NITROX microcode does not maintain the MTU size for the network connection, in order to calculate when fragmentation is needed. Therefore fragmentation count must be calculated externally and provided as part of the instruction parameters. No fragmentation is performed on the current packet when the *Frag_Num* field is set to 0x0. A maximum of 15 fragments is allowed by this instruction. When fragmentation is enabled, the Completion Code for the instruction will be at the end of the last fragment.

Non-fragmented packets can be of any size. However when selecting fragmentation, NITROX microcode requires the minimum fragment payload size for the first fragment (depends on Param1[10] bit, if param[10] == 1, Minimum Fragment size is 120 bytes otherwise 344 bytes. This requirement ensures that the minimum network MTU of 576 bytes is supported, while still delivering acceptable fragmentation performance. *Frag_Num* must be selected to ensure this minimum fragment size. E.g. Original IP Packet Length/Frag_Num >(120B or 344B).

When NITROX fragments a packet into multiple packets, the resulting output written to Rptr is multiple fragmented packets. NITROX microcode provides the capability of adding n * 8-bytes of zero-pad between fragments, by entering a value in the *inter-frag pad size* field of the corresponding Outbound SA context memory. By adding this padding, it enables host software to add additional network layer information (e.g. L2 header/trailer) to the fragments, without having to move them to another, larger memory buffer. Inter-frag padding is only supported in Direct DMA mode, or Gather-Only DMA mode (Scatter/Gather DMA mode, with s_size set to 0)

Two modes of packet fragmentation are supported during outbound IPSec packet processing. The type of fragmentation is selected in the Control field of the IPSec Outbound SA context data structure. Select *pre-fragmentation* to fragment the IPsec payload prior to performing IPSec processing (encryption, authentication, header insertion). Select *post−fragmentation* to first process the payload for IPSec, and then fragment the resulting processed IPSec payload.

The difference is subtle but important. With pre-fragmentation, HMAC processing (if any) will be performed on each fragmented packet, and the MAC field of the AH or ESP header will be specific to that fragment. Also, with encryption the IV will be across individual fragments. With post-fragmentation, all encryption and HMAC processing is performed on the payload prior to fragmenting it. Thus, the IV and MAC fields will be across the entire fragment set. The result is that upon processing of inbound packets, reassembly will either need to be performed before or after IPSec processing, for post-fragmentation and pre-fragmentation respectively.

Note that overall packet processing performance is reduced when fragmentation is used.

Frag_Num Calculation – Post-Fragmentation in Direct DMA Mode

Fragments created in this mode do not overlap, and the size of each fragment is a multiple of 8 bytes in length. To calculate the number of fragments needed, use the formula:

Frag_Num = ROUNDUP[PayloadSize/MaxPayloadSize]

Where:

- *PayloadSize* = (L-20) bytes = size of IPSec *payload* after IPSec processing
  - L = total number of bytes in the outbound IPSec packet (including outer IP header)
  - 20 bytes account for the outer IPv4 header
- *MaxPayloadSize* = ROUNDDOWN8(MTU−20).
  - MTU is the maximum transmission unit (bytes) for the network
  - 20 bytes account for the outer IPv4 header
  - Any MaxPayloadSize formula can be used here, as long as the first fragment created does not exceed the MTU and is not smaller than 120 bytes for Minimum_Fragment_size bit set otherwise 344 bytes.

Example:

Using the same 1500 byte IP packet example from above,

Rlen(L) = 1560 bytes

IPSec *PayloadSize* = 1560-20 = 1540 bytes.

Using a common MTU size of 1500 bytes,

Frag_Num = ROUNDUP((1560-20)/ROUNDDOWN8(1500-20))
= ROUNDUP(1540/1480)
= 2

The resulting output of the operation (written at Rptr) would then be structured as follows:

- IP Fragment (L'+ ((IP hlen & 1) ? 4:0) bytes)
- Inter Fragment Padding (Inter-frag pad size bytes)
- IP Fragment (L'+ ((IP hlen & 1) ? 4:0) bytes)
- Inter Fragment Padding (Inter-frag pad size bytes)
- IP Fragment (L(last) bytes) + (L(last) % 8) (bytes)
- Completion code (1 byte)

Where:

- L' = unit size of each fragment except the last, as calculated below
- IP hlen = IP Header Length (32-bit words).
- ((IP hlen & 1) ? 4:0) = if IP hlen is odd then add 4 bytes, else add 0 bytes
- L(last) = size of last fragment, as calculated below:
- (L(last) % 8) bytes) = don't-care data written at end of last fragment, to force completion code to be written on an 8-byte address boundary.
- Inter-frag pad size = initialized as part of the Outbound SA Context data structure for this IPSec SA

Fragments that are created will be of approximately equal size. The transmission unit size, L', of each fragment except the last one is:

L' = ROUNDUP8[(*Rlen*(L) − 20)/(*Frag_Num*) + 1] + 20 (bytes)

The last fragment size is the remainder of the division using the new transmission size:

L(last)= [(*Rlen(L)*−20)%(L'−20)] + 20 (bytes)

Example:

Again, starting from the 1500B IP packet example used above:

L' = ROUNDUP8[(1560-20)/2 + 1)] + 20
= 776 + 20
= 796 bytes

L(last) = [(1560-20) % (796-20)] + 20
= 764 + 20
= 784 bytes

Finally, recalculate *Rlen* for the data structure above, to find the final size of output buffer pointed to by Rptr.

Example

Let *InterIP Frag Size* == 1 (8 bytes), and let *hlen* = 5 (20 bytes = minimum for an IPv4 header). Then:

Rlen(fragmented) = IP Frag Size[frag #1] + 4 (bytes)
+ InterIP Frag Size * 8 (bytes)
+ IP Frag Size[frag last] (bytes)

Rlen(fragmented) = (796 + 4) + 8 + 784 bytes
= 1596 bytes

Frag_Num Calculation – Post−fragmentation in Scatter−Gather DMA Mode

In the scatter−gather dma mode, special care needs to be taken by software, to ensure that each new packet starts at the desired location. The inter −fragment padding parameter is ignored in this mode.

Example:

Direct mode−dma
MTU = 800
Fragmentation = post−frag
Protocol = ESP(3DES / SHA1)
Mode = Transport
UDP encaps
Original IP packet length, L = 1550 bytes
Inter−fragment padding = 18 bytes
MaxPayLoadSize = ROUNDDOWN8(800 – 20) = 776 bytes
Rlen(L) = T(0) + H(20) + E(8) + I(8) + ROUNDUP8(L −H+2)(1536) + D(12) + U(8) = 1592 bytes
# of fragments = PayloadSize(1592 – 20) / Max Payload size(776) = 3
L' = RoundUp8((1592−20)/3 +1) + 20 = 528 + 20 = 548
Llast = (1592 – 20)%(548−20) + 20 = 516

IPCOMP processing

NITROX IPSec microcode supports use of IPCOMP packets as an input, instead of IP packets. **NITROX *does not* perform data compression, or IPCOMP header processing.** Therefore another agent must perform any

required IPCOMP compression and header insertion, and then pass the resulting IPCOMP packet to NITROX for IPSec processing.

IPCOMP is only supported when the IPSec SA is configured in tunnel mode. Transport mode IPCOMP packets are not supported. To select IPCOMP, enter the length (in bytes) of the IPCOMP packet in the param2[*IPCOMP_Size*] field of the instruction. A zero in this field indicates that the packet is a regular IP packet, not an IPCOMP packet. When IPCOMP is selected, the DF bit & TOS bits from the original uncompressed IP packet MUST be provided as input data along with the IPCOMP packet to be processed. These fields will be used along with the Tunnel Header template in the associated context, to create the Tunnel IP header in the resulting output packet(s).

TFC Arbitrary Padding Support:
Supported *ESP tunnel mode* only, as a combination of host software and microcode. Not supported in AH, or in ESP Transport Mode. *TFC Arbitrary Padding* and *Pre-fragmentation* are mutually exclusive. They are not to be used together.

Host software should generate and append TFC padding bytes to the original IP packet prior to presenting it to Nitrox for IPSec packet processing. The length of the TFC padding should be specified in the Param2 input field, and the TFC_Pad_Enable bit should be set the param1. The total length including TFC padding should also be reflected in the “Dlen” calculation. Microcode will add this length to the original IP length, encrypt the complete IP payload including TFC padding and do normal IPSec payload padding as well. HMAC will be calculated on the entire packet and appended as authentication MAC trailer.

Dlen and Rlen values should be increased by the size of TFC padding.

TFC dummy packet support:
Dummy packet processing is supported in both random and fixed lengths. The length is decided by the length field in the IP header of the input request. This feature is supported *only in ESP tunnel mode.*

Outbound processing is done by setting the TFC_Dummy_Packet bit in param1 of the instruction (bit 13). When set, microcode generates a dummy packet and puts it in the payload field of the ESP packet. The next_header field of the ESP trailer is assigned a value of 0x3b (59). (i.e No next header). Fragmentation is not supported in this mode.

Extended Sequence Numbers:
ESN is supported by setting the *ESN_Mode* bit in the Extended_Control field of the SA context, and configuring the high-order 32 bits of the sequence number in context memory.

If the *ESN_Mode* bit is set, then microcode will append the high-order 32-bits of ESN to the ESP trailer, immediately following the next_header field, and will calculate the ICV value as per RFC 4303 specification.

For AES_GCM and AES_GMAC, prepare the AAD data with 64-bit extended sequence number as per the respective RFC's. (AES GCM: RFC 4106. AES GMAC: RFC 4543). The high order 32 bits of extended sequence number are not transmitted. Host software must write and update the ESN high-order bits in Nitrox context memory as needed.

Minimum Fragment Size :

Minimum Fragment Size is set by enabling 10th bit of Parma1 of Outbound Packet Processing. If this bit set minimum fragment size allowed is 120Bytes else 344Bytes

Note: 1. Ipv4 and Ipv6 Base headers only supported.
2. SHA384/512 is not supported.

Opcode PROCESS_INBOUND_IPSEC_PACKET

Performs complete IPSec inbound packet processing. Input to this instruction is a complete IPSec (AH/ESP) packet. Output is a complete IP packet, an SA index, and this packet's sequence number. SA index and sequence number are returned to allow for optional anti-replay processing in software.

IPSec inbound packet processing includes authentication and/or decryption of IPSec payload, removal of padding, removal of IPSec header(s), including tunnel header if tunnel mode is used, removal of UDP encapsulation (if selected), and all required IPSec packet/protocol integrity checks.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	<7:0> PROCESS_INBOUND_IPSEC_PACKET (0x10) <15:8> *Context_Address_Select* – selects whether to use explicit context memory address specified by Cptr, or implicit address decoded as a function of the packet's SPI value 0x00: Context address specified by Cptr 0x01: Context address calculated from SPI
• Param1[15:00]:	<15:14> Reserved. Set to 0 <*13*> *UDP_Checksum* 0: Verify the Input IPv4 checksum 1: No checksum verification <12:0> Reserved. Set to 0
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- IPSec packet (Variable length) // Packet format/size depends upon IPSec options configured // in context data structure.

Output Data (Rptr):

- SA index (4 bytes) // For use by host software, for anti-replay check
- Sequence number (4 bytes) // For use by host software, for anti-replay check
- IP Packet (***Rlen*** bytes) // IP packet format and *Rlen* depend on IPSec mode and // options configured in Cptr structure. See below for explanation // of *Rlen* calculation.
- Completion code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

Context data structure for this instruction is initialized under NITROX control via instruction WRITE_IPSEC_INBOUND_CONTEXT above. Refer to that instruction for a description of this Context data structure.

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x14	BAD_IPSEC_PROTOCOL	*Next Header* field in IP packet is something other than ESP, AH, or UDP
0x15	BAD_IPSEC_AUTHENTICATION	MAC compare failed. Packet has been modified
0x16	BAD_IPSEC_PADDING	ESP pad count and padding bytes are not in correct decremented order, due to bad padding or bad cipher key.
0x17	BAD_IP_VERSION	IP version in the IP header is not IPv4 or IPv6
0x1c	BAD_IPSEC_SPI	SPI value in packet did not match SA SPI value stored at Cptr
0x1d	BAD_CHECKSUM	IP header checksum field is non-zero, and is different than internally calculated checksum
0x1e	BAD_IPSEC_CONTEXT	*Valid* bit in specified Context is not set (context is not valid)
0x1f	BAD_IPSEC_CONTEXT_DIRECTION	Direction bit in the context is outbound, not inbound
0x20	BAD_IPSEC_CONTEXT_FLAG_MISMATCH	Context *encapsulation* and *protocol* fields do not match with input packet
0x21	IPCOMP_PACKET_DETECTED	Completed with no error, and resulting output is an IPCOMP packet.
0x2c	DUMMY_PAYLOAD	Next header field in the received ESP trailer was 59 (0x3b), indicating packet received was a TFC dummy packet. This code will be returned for dummy packets even if packet authentication MAC miscompare was detected. Selector and protocol checks are also not performed for dummy packets.
0x2d	BAD_IP_PAYLOAD_TYPE	Next header of ESP/AH is not IPv4, IPv6, IPCOMP.
0x32	IPV6_EXTENSION_HEADERS_TOO_BIG	IPv6 Extension Header length exceeds 248bytes in Transport mode (or) 248bytes in Tunnel mode

0x33	IPV6_HOP_BY_HOP_ERROR	HOP BY HOP Header is not the very first extension header and multiple occurrence of HOP BY HOP header.
0x34	IPV6_DECRYPT_RH_SEGS_LEFT_ERROR	IPv6 Routing Header's segment left field is not equal to zero
0x35	IPV6_RH_LENGTH_ERROR	IPv6 Routing Header Length is not divisible by 2
0x38	BAD_AUTH_TYPE_GCM_GMAC	*Authentication_Type* != 11b and *AES_GCM_Mode* == 1
0x39	BAD_ENCRYPT_TYPE_CTR_GCM	*Encryption_Type* == *DES or NULL and (AES_CTR_Mode ==1 or AES_GCM_Mode ==1)
0x3a	AH_NOT_SUPPORTED_WITH_GCM_GMAC_SHA2	*IPSec_Protocol* == AH and Authentication_Type == AES_GMAC, SHA256, SHA384 or SHA512 Note: AH with SHA2 or GMAC not supported

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

Rlen Calculation

Define the following variables:

L: IPSec Packet Length
- If (IP version == IPv4)
  - L = Total IPv4 IPSec packet length, in bytes (header + payload)
- Else // version == IPv6
  - L = Total IPv6 IPSec packet length, in bytes (header + payload)

T: Tunnel Mode Header Length
- If *IPSec_Mode* == transport
  - T = 0 (bytes)
- Else // tunnel mode header
  - If (IP version == IPv4)
    - T = 20 (bytes)
  - Else // (IP version == IPv6)
    - T = 40 + extension header length (bytes)
- If(Header_Select == 0x01)
  - T = T (bytes)

E: ESP Header/Trailer Length
- E = 8 (bytes)

A: AH Header Length
- A = 12 (bytes)

D: MAC Length
- If (Auth Type == NULL)
  - D = 0 (bytes)
- Else if(Auth Type == MD5 or SHA)

D = 12 (bytes)
Else if (Auth Type == SHA256 || AES_GCM_flag == 1 || AES_GMAC_flag == 1)
D = 16 (bytes)
Else if (Auth Type == SHA 384)
D = 24 (bytes)
Else if (Auth Type == SHA512)
D = 32 (bytes)
Else if (Auth Type == AESXCBC)
D = 12 (bytes)

I: IV Length
If (encrypt type == NULL)
I = 0 (bytes)
Else if (encrypt type == DES)
I = 8 (bytes)
Else // (encrypt type == AES)
I = 16 (bytes)

U: UDP encapsulation length
If (encaps type == UDP)
U = 8 (bytes)
Else // (encaps type == NULL)
U = 0 (bytes)

Then:

For AH:
Rlen = L - T - A - D

For ESP:
Rlen = L - T - E - I - D – U

TFC Arbitrary padding support:
Supported *ESP tunnel mode* only, as a combination of host software and microcode. Not supported in AH, or in ESP Transport Mode. *TFC Arbitrary Padding* and *Pre-fragmentation* are mutually exclusive. They are not to be used together.

Microcode decrypts complete packet and sends to the host software, including TFC padding. Original packet can be recovered by host software, from the length in the IP header.

Important Note: While this feature has been tested to the best of Cavium Networks ability, no standard test vectors are available for this feature at the time of this writing.

TFC dummy packet support:
Dummy packet processing is supported in both random and fixed lengths. The length is decided by the length field in the IP header of the input request. This feature is supported *only in ESP tunnel mode.*

Microcode decrypts the input packet and if it encounters a next header field of 0x3b (59), it sets the completion code to 0x2c (44) to identify this as a dummy packet. Packet authentication, and selector and protocol checks are skipped when a dummy packet is detected.

Extended Sequence Numbers:
ESN is supported by setting the *ESN_Mode* bit in the Extended_Control field of the SA context, and configuring the high-order 32 bits of the sequence number in context memory.

If the *ESN_Mode* bit is set, then microcode will append the high-order 32-bits of ESN to the ESP trailer, immediately following the next_header field, and will calculate the ICV value as per RFC 4303 specification.

For AES_GCM and AES_GMAC, prepare the AAD data with 64-bit extended sequence number as per the respective RFC's. (AES GCM: RFC 4106. AES GMAC: RFC 4543). The high order 32 bits of extended sequence number are not transmitted. Host software must write and update the ESN high-order bits in Nitrox context memory as needed.

6.2 General Cryptography Instruction Group

Opcode RANDOM

Returns specified amount of random data, generated by the on-chip "True" Random Number Generator (TRNG). Refer to NITROX III Hardware Manual for a description of the TRNG implementation.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	RANDOM (0x0101)
• Param1[15:00]:	<15:0> *Size* (in bytes) of random data to produce. Valid range: 0 – 0xFFFF
• Param2[15:00]:	Reserved. Set to 0
• Dlen[15:00]:	Reserved. Set to 0

Input Data (Dptr):
No input data used in this instruction. Set Dptr to 0

Output Data (Rptr):
- Random Data (*Size* bytes)
- Completion code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Opcode MOD_EX

Performs Modular exponentiation [a^e MOD m]

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	MOD_EX (0x0004) MOD_EX_LARGE (0x0002: If modulus length > 1024 bits)
• Param1[15:00]:	*Modulus_Length* (in bytes). Min: 17 (136 bits) Max: 512 (4096 bits)
• Param2[15:00]:	*Exponent_Length* (in bytes) Min: 1 (8 bits) Max: < Mod length (4096 bits)
• Dlen[15:00]:	*Size* (in bytes) in input structure pointed to by Dptr. See Dptr data structure definition below. Min: 19 (17 + 1 + 1) Max: 1536 (512 + 512 + 512)

Input Data (Dptr):

- Modulus "m" (*Modulus_Length* bytes) // In byte array format
- Exponent "e" (*Exponent_Length* bytes) // In byte array format
- Data "a" (1 to *Modulus_Length* bytes) // In byte array format

NOTE: see below for description of byte array format

Output Data (Rptr):

- Result (*Modulus_Length* bytes) // In byte array format.
- Completion code (1 byte)

Context Data (Cptr - must be 8-byte aligned):

No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x17	BAD_LENGTH	Modulus length < 17, or Modulus length > 512, or Exponent length > Modulus length , or Input Data Length > Modulus length

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

Byte Array Format Example:

Let:
modulus = 0x3256c6e59fadee9cc8b3cd1ab0921ef5a1 (Modulus_Length = 17)
exponent = 0x0000000000000003 (Exponent_Length = 8)
Input data = 0x0003 (2 bytes)

Then;
modulus[0] =0x32
modulus[1] =0x56
.
.
.
modulus[16]=0xa1

exponent[0]=0x00
.
.
.
exponent[8]=0x03

data[0] =0x00
data[1] =0x03

6.3 AES_CTR(SRTP) Instruction Group

The Secure Real-time Transport Protocol (or SRTP) defines a profile of RTP (Real-time Transport Protocol), intended to provide encryption, message authentication (integrity), and replay protection to RTP data in both unicast and multicast applications. This group of instructions perform encrypt/decrypt/auth of the SRTP payload in a single pass of the data through the chip. Instructions in this group do not perform full SRTP protocol processing.

Refer to RFC 3711 for detailed information about SRTP.

The instructions below may also be used for simple AES CTR mode encryption, by setting Hash_Type, Protocol_Type, Header_Length, Index_Length, and Tag_Length 0.

Opcode AES_CTR(SRTP)_ENCRYPT

Encrypts SRTP payload with AES CTR mode, and authenticates with SHA1 or Null. May also be used for simple AES CTR mode decryption by setting Hash_Type, Protocol_Type, Header_Length, Index_Length, and Tag_Length 0.

NOTE: Encrypt and Decrypt instructions use the same Opcode value. Processing direction is selected by Opcode<8>. It is documented herein as 2 different instructions for clarity only.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	<7:0> SRTP_ENCRYPT (0x18) <15> Reserved. Set to 0 <14> *Protocol_Type* 0: SRTP 1: SRTCP <13:11> *HASH_Type* 0: Null 1: SHA1 <10:9> *AES_Type* 0: AES128 1: AES192 2: AES256 3: reserved <8> Reserved. Set to 0
• Param1[15:00]:	*Payload_Length* – size (bytes) of payload data, not including AES Key, IV, Auth Key, SRTP Header and SRTP Index. Min : 0 Max : [2^16 - IV length - Encryption Key_Length - Authentication Key

	Length - ROUNDUP8(*Header_Length*) - ROUNDUP8(*Index_Length*)]
• Param2[15:00]:	<15:12> *Tag_Length* (bytes) Min: 1 Max : 10 <11:8> *Index_Length* (bytes). For SRTP ROC value (used to derive Index). For SRTCP, E-bit and Sequence number. Note: MKI is not supported. Valid values: 0 or 4 only. <7:0> *Header_Length* (bytes) – length of SRTP/SRTCP header. Min: 0 Max: 72
• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- IV (16 bytes)
- AES key (16, 24, or 32 bytes)
- SHA1 key [24 bytes. Last 4 bytes must be zero]

If (*Protocol_Type*==SRTP)
- ROC (ROUNDUP8(*Index_Length*) bytes)

Else // (*Protocol_Type*==SRTCP)
- E-bit||Sequence Number (ROUNDUP8(*Index_Length*) bytes)

- SRTP/SRTCP header (ROUNDUP8(*Header_ Length*) bytes)
- Plaintext payload (ROUNDUP8(*Payload_Length*) bytes)

Output Data (Rptr):
- Encrypted payload (*Payload_Length* bytes)

If (*Protocol_Type*==SRTCP)
- E-bit||Sequence Number (*Index_Length* bytes)

If (*HASH_Type*== SHA1)
- Authentication Tag (*Tag_Length* bytes)

- Completion code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x18	BAD_AUTH_TYPE	Authentication algorithm is other than NULL and SHA1.
0x19	BAD_ENCRYPT_TYPE	Encryption algorithm is other than AES128, AES192 and AES256.

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

IV format:

For SRTP,
(session_salt * 2^16) XOR (SSRC * 2^64) XOR ((ROC||SEQ) * 2^16)

For SRTCP,
(session_salt * 2^16) XOR (SSRC * 2^64) XOR (SRTCP_Index * 2^16)

Opcode AES_CTR(SRTP)_DECRYPT

Decrypts SRTP payload with AES CTR mode, and authenticates with SHA1 or Null. May also be used for simple AES CTR mode decryption by setting Hash_Type, Protocol_Type, Header_Length, Index_Length, and Tag_Length 0.

NOTE: Encrypt and Decrypt instructions use same Opcode value. Direction is selected by Opcode<8>. It is documented herein as 2 different instructions to add clarity.

Instruction Syntax:

OpCode/Params:

• Opcode[15:00]:	<7:0> SRTP_DECRYPT (0x18) <15> Reserved. Set to 0 <14> *Protocol_Type* 0: SRTP 1: SRTCP <13:11> *HASH_Type* 0: Null 1: SHA1 <10:9> *AES_Type* 0: AES128 1: AES192 2: AES256 3: reserved <8> Reserved. Set to 1
• Param1[15:00]:	*Payload_Length* (bytes) – size of SRTP payload, not including AES KEY, IV, Auth Key, header and Index. Min : 0 Max : [2^16 - IV length – Encryption Key Length - Authentication Key Length - ROUNDUP8(Header_length) - ROUNDUP8(Index_length) – Authentication Tag Length]
• Param2[15:00]:	<15:12> *Tag_Length* (bytes) Min: 0 Max : 10 <11:8> *Index_Length* (bytes). For SRTP ROC value (used to derive Index). For SRTCP, E-bit and Sequence number. Note: MKI is not supported. Valid values: 0 or 4 only. <7:0> *Header_Length* (bytes) – length of SRTP/SRTCP header. Min: 0 Max: 72

• Dlen[15:00]:	*Size* (in bytes) of complete input data structure pointed to by Dptr. See Dptr data structure definition below to calculate appropriate value.

Input Data (Dptr):

- IV (16 bytes)
- AES key (16, 24, or 32 bytes)
- SHA1 key [24 bytes. Last 4 bytes must be zero]

If (*Protocol_Type*==SRTP)
- ROC (ROUNDUP8(*Index_Length*) bytes)

Else // (*Protocol_Type*==SRTCP)
- E-bit||Sequence Number (ROUNDUP8(*Index_Length*) bytes)

- Encrypted payload (ROUNDUP8(*Payload_Length*) bytes)
- Authentication tag [If SHA1] (*Tag_Length* bytes)

Output Data (Rptr):
- Decrypted plaintext payload (*Payload_Length* bytes)

If (*Protocol_Type*==SRTCP)
- E-bit||Sequence Number (*Index_Length* bytes)

- Completion code (1 byte)

Context Data (Cptr - must be 8-byte aligned):
No context data used in this instruction. Set Cptr to 0

Completion codes:

Value	Mnemonic	Description
0x00	SUCCESS	Completed with no error
0x18	BAD_AUTH_TYPE	Authentication algorithm is other than NULL or SHA1.
0x19	BAD_ENCRYPT_TYPE	Encryption algorithm is other than AES128, AES192 or AES256.
0x3d	BAD_SRTP_AUTH_TAG	The tag value in the received packet does not match calculated value.

NOTE: No other completion codes are defined for this instruction. Refer to **Appendix B Completion Code Quick Reference** for a description of other possible completion types, or non-completion events.

Additional User Guidelines

IV format:

For SRTP,
(session_salt * 2^16) XOR (SSRC * 2^64) XOR ((ROC||SEQ) * 2^16)

For SRTCP,
(session_salt * 2^16) XOR (SSRC * 2^64) XOR (SRTCP_Index * 2^16)

Appendix A Instruction Set Quick Reference:

Lists all instructions (and associated opcodes) implemented by the specific microcode version listed in the introduction.

Boot Microcode Instruction Set

Microcode CN35xx-MC-BOOT-0001

Instruction Name	Opcode	Page #
Microcode Initialization Instruction Group		
BOOT_INIT	0x0000	38
SETUP_CONST	0x0016	40
Debug Instruction Group		
READ_INTERNAL_DATAPATH	0x1101	41
CHECK_ENDIAN	0x1201	42
ECHO_CMD	0x1301	43
ROLL_CALL	0x1401	44

IPSec/IKE Microcode Instruction Set

Microcode CNN35x -MC-IPSEC-0002

Instruction Name	Opcode	Page #
IPSec Packet Processing Group		
WRITE_IPSEC_OUTBOUND_SA	0x4014	45
WRITE_IPSEC_INBOUND_SA	0x2014	53
ERASE_CONTEXT	0x0114	59
PROCESS_OUTBOUND_IPSEC_PACKET	0x0011	61
PROCESS_INBOUND_IPSEC_PACKET	0x0010 0x0110	72

General Cryptography Instruction Group		
RANDOM	0x0101	76
MOD_EX	0x0004 0x0002	78
AES_CTR Instruction Group		
AES_CTR(SRTP) Encrypt	0x0018	80
AES_CTR(SRTP) Decrypt	0x0118	83

Appendix B Completion Code Quick Reference

When microcode completes instruction processing, it writes a 1-byte code to the one of two locations in host system memory. The *Completion Address* is the byte address following the last byte of the output data structure. This address is written when the microcode is able to faithfully calculate Rlen (the expected size of the structure at Rptr). The *Error Address* is a location in host memory specified by the host driver, and written to the specified location in NITROX internal SRAM. Refer to TBD for information about the Error Address.

The value of the completion code written to the Completion or Error Address can vary depending on the instruction. Two instructions may use a common completion code value (e.g. 0x0f) to identify two completely different error types.

Below is a list of completion codes, the host memory location where they will be written (Completion or Error Address), the associated Mnemonic, a brief description of the completion code's meaning, and the instruction type(s) capable of generating the code.

Value	To Address	Mnemonic	Description	Instruction
0x00	Completion	SUCCESS	Completed with no error	All
0x01	Error	BAD_OPCODE	Invalid (Undefined) Opcode	All
0x01	Error	BAD_SCATTER_GATHER_LIST	Number of scatter elements are zero, or number of gather elements are zero, or Dlen is less than the SG size calculated from s_Len and g_size.	All(when SG mode is used)
0x0d	Completion	BAD_SCATTER_GATHER_ WRITE_LENGTH	Total scatter length is less than *rlen*	All(when SG mode is used)
0x0f	Error	BAD_LENGTH	Modulus length < 34, or Modulus length > 512, or Exponent length > Modulus length – 11, or Data Length > Modulus length – 11	MOD_EX
0x0f	Error	BAD_LENGTH	If the packet length goes negative by subtracting the custom header.	PROCESS_OUTBOUND_ IPSEC_PACKET
0x14	Error	BAD_IPSEC_PROTOCOL	*Next Header* field in IP packet is something other than ESP, AH, or UDP	PROCESS_INBOUND_ IPSEC_PACKET
0x15	Completion	BAD_IPSEC_AUTHENTICATION	MAC compare failed. Packet has been modified	PROCESS_INBOUND_ IPSEC_PACKET

0x16	Completion	BAD_IPSEC_PADDING	ESP pad count and padding bytes are not in correct decremented order, due to bad padding or bad cipher key.	PROCESS_INBOUND_ IPSEC_PACKET
0x17	Error	BAD_IP_VERSION	IP version in the IP header is not IPv4 or Ipv6, selector check version mismatch	PROCESS_INBOUND_ IPSEC_PACKET
0x17	Error	BAD_IP_VERSION	IP version in the IP header is not IPv4 or Ipv6	PROCESS_OUTBOUND_ IPSEC_PACKET
0x18	Error	BAD_AUTH_TYPE	Authentication algorithm is other than SHA1, MD5, SHA2, GCM/GMAC or AES-XCBC	WRITE_IPSEC_ OUTBOUND_SA WRITE_IPSEC_ INBOUND_SA
0x18	Error	BAD_AUTH_TYPE	Authentication algorithm nis other than NULL or SHA1.	AES_CTR(SRTP)_ ENCRYPT AES_CTR(SRTP)_ DECRYPT
0x19	Error	BAD_ENCRYPT_TYPE	Encryption algorithm is other than AES128, AES192 and AES256.	AES_CTR(SRTP)_ ENCRYPT AES_CTR(SRTP)_ DECRYPT
0x1c	Error	BAD_IPSEC_SPI	SPI value in packet did not match SA SPI value stored at Cptr	PROCESS_INBOUND_ IPSEC_PACKET
0x1d	Error	BAD_CHECKSUM	IP header checksum field is non-zero, and is different than internally calculated checksum	PROCESS_INBOUND_ IPSEC_PACKET
0x1e	Error	BAD_IPSEC_CONTEXT	*Valid* bit in specified Context is not set (context is not valid)	PROCESS_OUTBOUND_ IPSEC_PACKET PROCESS_INBOUND_ IPSEC_PACKET
0x1f	Error	BAD_IPSEC_CONTEXT_ DIRECTION	Direction bit in the context is outbound, not inbound	PROCESS_INBOUND_ IPSEC_PACKET
0x20	Error	BAD_IPSEC_CONTEXT_FLAG_ MISMATCH	Context *encapsulation* and *protocol* fields do not match with input packet	PROCESS_INBOUND_ IPSEC_PACKET
0x21	Completion	IPCOMP_PAYLOAD	Completed with no error, and resulting output is an IPCOMP packet.	PROCESS_INBOUND_ IPSEC_PACKET

0x23	Completion	BAD_SELECTOR_MATCH	If port or address of the selector check mismatch.	PROCESS_INBOUND_ IPSEC_PACKET
0x26	Error	BAD_FRAnGMENT_SIZE	Fragment size is less than expected.	PROCESS_OUTBOUND_ IPSEC_PACKET
0x29	Error	BAD_INLINE_DATA	IPCOMP packet and Transport Mode and/or fragmentation is selected, Ipcomp with pre-frag is enabled	PROCESS_OUTBOUND_ IPSEC_PACKET

0x2a	Error	BAD_FRAG_SIZE_ CONFIGURATION	Postfrag or prefrag,
			If (Minimum_Fragment_Size == 1) {
			Ipv6 packet has extension headers
			(or)
			Ipv4 packet has options
			}
				PROCESS_OUTBOUND_ IPSEC_PACKET

0x2c	Completion	DUMMY_PAYLOAD	Next header field in the received ESP trailer was 59 (0x3b), indicating packet received was a TFC dummy packet. This code will be returned for dummy packets even if packet authentication MAC miscompare was detected. Selector and protocol checks are also not performed for dummy packets.	PROCESS_INBOUND_ IPSEC_PACKET
0x2d	completion	BAD_IP_PAYLOAD_TYPE	Next header of ESP/AH is not IPv4, IPv6, IPCOMP.	PROCESS_INBOUND_ IPSEC_PACKET
0x2e	Error	BAD_MIN_FRAG_SIZE_ AUTH_SHA384_512	If (min_fragment_size == 1) i.e parma1[10] auth = SHA384/SHA512.	PROCESS_OUTBOUND_ IPSEC_PACKET
0x2f	Completion	BAD_ESP_NEXT_HEADER	NH = CustomHdr and CustomHdr bit is not set in param1[15].	PROCESS_INBOUND_ IPSEC_PACKET
0x32	Error	IPV6_EXTENSION_ HEADERS_TOO_BIG	IPv6 Extension Header length exceeds 320bytes in Transport mode (or) 248bytes in Tunnel mode	PROCESS_OUTBOUND_ IPSEC_PACKET, PROCESS_INBOUND_ IPSEC_PACKET

0x33	Error	IPV6_HOP_BY_HOP_ERROR	HOP BY HOP Header is not the very first extension header or multiple occurrence of HOP-BY_HOP header.	PROCESS_OUTBOUND_IPSEC_PACKET, PROCESS_INBOUND_IPSEC_PACKET
0x34	Error	ERROP__IPV6_DECRYPT_RH_SEGS_LEFT_ERROR	Decrypt IPv6 Routing Header Segment Left is non zero	PROCESS_INBOUND_IPSEC_PACKET
0x35	Error	IPV6_RH_LENGTH_ERROR	IPv6 Routing Header Length is not divisible by 2	PROCESS_OUTBOUND_IPSEC_PACKET
0x36	Error	IPV6_OUTBOUND_RH_COPY_ADDR_ERROR	IPv6 Routing Header with Header Preservation is not Supported	PROCESS_OUTBOUND_IPSEC_PACKET
0x37	Error	ERROP__IPV6_EXTENSION_HEADER_BAD	if Mobility Header is present in the outer header on case of tunnel mode.	PROCESS_INBOUND_IPSEC_PACKET, PROCESS_OUTBOUND_IPSEC_PACKET
0x39	Error	BAD_ENCRYPT_TYPE_CTR_GCM	*Encryption_Type* == *DES or NULL and (AES_CTR_Mode ==1 or AES_GCM_Mode ==1)	PROCESS_OUTBOUND_IPSEC_PACKET PROCESS_INBOUND_IPSEC_PACKET
0x3a	Error	AH_NOT_SUPPORTED_WITH_GCM_GMAC_SHA2	*IPSec_Protocol* == AH and Authentication_Type == AES_GMAC, SHA256, SHA384 or SHA512 Note: AH with SHA2 or GMAC not supported	PROCESS_OUTBOUND_IPSEC_PACKET PROCESS_INBOUND_IPSEC_PACKET
0x3b	Error	TFC_PADDING_WITH_PREFRAG_NOT_SUPPORTED	TFC padding enabled and fragmentation type in SA context is specified as prefrag.	PROCESS_OUTBOUND_IPSEC_PACKET
0x3d	Completion	BAD_SRTP_AUTH_TAG	The tag value in the received packet does not match calculated value.	AES_CTR(SRTP)_DECRYPT

Appendix C SHA Hash Initial Values

The following text was copied from NIST publication "FIPS 180-4: Secure Hash Standard (SHS)".

5.3 Setting the Initial Hash Value (H(0))

Before hash computation begins for each of the secure hash algorithms, the initial hash value, H(0), must be set. The size and number of words in H(0) depends on the message digest size.

5.3.1 SHA-1

For SHA-1, the initial hash value, H(0), shall consist of the following five 32-bit words (20 bytes), in hex:

$H_0(0)$ = 67452301
$H_1(0)$ = efcdab89
$H_2(0)$ = 98badcfe
$H_3(0)$ = 10325476
$H_4(0)$ = c3d2e1f0

5.3.2 SHA-224

For SHA-224, the initial hash value, H(0), shall consist of the following eight 32-bit words (32 bytes), in hex:

$H_0(0)$ = c1059ed8
$H_1(0)$ = 367cd507
$H_2(0)$ = 3070dd17
$H_3(0)$ = f70e5939
$H_4(0)$ = ffc00b31
$H_5(0)$ = 68581511
$H_6(0)$ = 64f98fa7
$H_7(0)$ = befa4fa4

5.3.3 SHA-256

For SHA-256, the initial hash value, H(0), shall consist of the following eight 32-bit words (32 bytes), in hex:

$H_0(0)$ = 6a09e667
$H_1(0)$ = bb67ae85
$H_2(0)$ = 3c6ef372
$H_3(0)$ = a54ff53a
$H_4(0)$ = 510e527f
$H_5(0)$ = 9b05688c
$H_6(0)$ = 1f83d9ab
$H_7(0)$ = 5be0cd19

These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

5.3.4 SHA-384

For SHA-384, the initial hash value, H(0), shall consist of the following eight 64-bit words (64 bytes), in hex:

$H_0(0)$ = cbbb9d5dc1059ed8
$H_1(0)$ = 629a292a367cd507
$H_2(0)$ = 9159015a3070dd17
$H_3(0)$ = 152fecd8f70e5939
$H_4(0)$ = 67332667ffc00b31
$H_5(0)$ = 8eb44a8768581511
$H_6(0)$ = db0c2e0d64f98fa7
$H_7(0)$ = 47b5481dbefa4fa4

These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.

5.3.5 SHA-512

For SHA-512, the initial hash value, H(0), shall consist of the following eight 64-bit words (64 bytes), in hex:

$H_0(0)$ = 6a09e667f3bcc908
$H_1(0)$ = bb67ae8584caa73b
$H_2(0)$ = 3c6ef372fe94f82b
$H_3(0)$ = a54ff53a5f1d36f1
$H_4(0)$ = 510e527fade682d1
$H_5(0)$ = 9b05688c2b3e6c1f
$H_6(0)$ = 1f83d9abfb41bd6b
$H_7(0)$ = 5be0cd19137e2179

These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

Appendix D Microcode Change History

Below is a brief list/review of feature changes between various microcode versions
Changes from build:

BUILD_0002:
Changes from previous build (0001)

==========

FEATURE UPDATE:

---

1. Added hmac_aes_decrypt_perf, hmac_des_decrypt_perf subroutines and modified calc_hmac_outer_hash subroutine to improve ipsec performance for AES/DES ciphers with MD5/SHA1 in the inbound.

2. Added hmac_encrypt_perf subroutine to improve ipsec performance for AES/DES ciphers with MD5/SHA1 in the outbound.

3. Performance improvement changes in both inbound and outbound.

4. Disabled 184 word DMA support, to improve the multi-core performance.

BUG-FIX:

---

1. Datapath write_check issue in AH outbound with mini_frag_size_bit set, in SG mode.

2. Fixed the usage of AUX_REG_17 in case of Mobililty header which causes the incorrect error.

BUILD_0001:
Changes from previous build (0000)
==========

FEATURE UPDATE:

1. Updated the DMA Read and Write to use the 184Words DO_DMA_IN_LEN for normal inbound and outbound processing.

2. Pre and Post fragmentation use the 152W as DO_DMA_IN_LEN.

3. Masking the group bits in the Cptr[63:61] to support the PLUS mode.

BUG-FIX:

1. During the Inbound processing, if ESP_NH is not p99(0x63) and p99 bit is set in param[15], then microcode returns with BAD_IPSEC_PROTOCOL at error address.

Updated error code as BAD_ESP_NEXT_HEADER (0x2f) in Completion Code(CC)

instead of writing at error address and continue processing.

```
if (CC != BAD_IPSEC_AUTHENTICATION)
   CC = BAD_ESP_NEXT_HEADER
else
   CC = BAD_IPSEC_AUTHENTICATION
```

2. Occurrence of the error BAD_MIN_FRAG_SIZE_AUTH_SHA384_512 in Inbound and this error should come in Outbound only. added condition check for inbound/outbound.

BUILD_0000:
Initial Version. From Px CNPx-MC-IPSEC-MAIN-0021_PRE_RELEASE. [BUILD_0020 + Mobility IPv6 support]