



IX-API SDK Reference



• • • • •

May 2000 Software Release SDK 3.0
Document Revision 2.3
Part Number 6750003

This document as well as the software described in it is furnished under license and may be used or copied only in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express permission of Intel Corporation.

Intel Corporation might have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give any license to these patents.

Copyright © 2000 Intel Corporation. All rights reserved.

Intel and the Intel logo are registered trademarks and Internet Exchange, NetBoost, NCL, and the NetBoost logo are trademarks of Intel Corporation in the United States and other countries.

ARM and StrongARM are trademarks of Advanced RISC Machines, Ltd.

InstallShield is a registered trademark and service mark of InstallShield Software Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the US and other countries.

Windows NT is a registered trademark of Microsoft Corporation.

*Other third-party brands and names are the property of their respective owners.

This product includes software developed by parties other than Intel. **See the back page of this document for a list of copyrights and license agreements.**

• • • • •

Intel Corporation
1350 Villa Street
Mountain View, CA 94041-1126
Tel: 650.567.9800
Fax: 650.567.9810
www.intel.com

Contents

.....

About This Referencexix

Audience xix

In This Reference xix

Other Sources of Information xx

Typographical Conventions xxi

Syntax Example xxii

Contacting Intel xxii

Web and Internet Sites xxii

Customer Support Technicians xxiii

Chapter 1

Overview1

About the Host System 2

About the Policy Accelerator 2

About the Application Programming Interface (API) of the SDK 2

Parts of an IX-API SDK Application 3

Packet Processing Units 4

Messages Between the Host and Policy Accelerator 5

Asynchronous Calls and Callback Functions 6

Creating an IX-API SDK Application 6

Creating the Host Module 6

Creating the Accelerator Module 7

For More Information 7

Chapter 2

System Types and Methods9

Overview 9

Include Files 9

Data Types 10

Byte Order Issues 10

Byte Order Classes 11

Byte Order and Intermodule Communication 12

Upcalls, Downcalls, and Byte Order	13
Operators and Byte Order	14
The Data Type API	15
nuint16 Class.	.16
nuint16 Constructor	17
htons Method	18
ntohs Method	18
raw_ Member	19
swaps Method	19
nuint32 Class.	.20
nuint32 Constructor	21
htonl Method	22
ntohl Method	22
raw_ Member	23
swapl Method	23

Chapter 3 **Host API. 25**

Overview	25
Include Files	25
Host API Class Organization	26
Object Pairing on the Host and Policy Accelerator	27
Dictionary Names	27
Application and ACE Management Classes	28
Message Support Classes	29
Base Class	30
Error Handling in the Host API	30
Host API Reference	31
Include Files	31
API Classes	31
AceGroup Class33
AceGroup Constructor	35
AceManager Class37
AceManager Constructor	40
getCompilerErrorMessages Method	43
getDropTarget Method	44
getPassTarget Method	44
getTag Method	45
load Method	45
releaseCompilerErrorMessages Method	47
releaseMessage Method	47

CrosscallHandlerManager Class48
CrosscallHandlerManager Constructor	50
CrosscallManager Class52
CrosscallManager Constructor	54
Downcall Class56
Downcall Constructor	59
call Method	61
Message Class.62
Message Constructor	63
getBuffer1 Method	64
getBuffer2 Method	64
getLen1 Method	64
getLen2 Method	65
MessageBlock Class.66
MessageBlock Constructor	67
NBApp1 Class68
NBApp1 Constructor	70
bind Method	72
getTag Method	74
getStackDriverName Method	76
link Method	77
unbind Method	79
unlink Method	80
NBError Class.81
getErrorCode Method	82
NBObject Class83
getId Method	83
getType Method	83
TargetManager Class85
TargetManager Constructor	86
UpcallHandler Class88
UpcallHandler Constructor	91
getUpcallFunction Method	93

Chapter 4 **Action Services Library 95**

Overview	95
TCP/IP Support	96
Environmental Restrictions	96
Include Files	96

Initialization	97
Action Functions	97
Packet Moving Classes	97
String Search Classes	98
String Search Management	98
Initiating and Continuing Searches	98
Search Operating Modes	99
String Search Classes	99
For More Information	99
Message Support Classes	99
For More Information	100
Time Support Classes	100
Statistical Support Class	101
Set Management Classes	101
Declaring Sets	101
Searches on Sets	101
Set Elements	102
Search Key Format	102
Set, Search, and Element Classes	103
For More Information	103
Memory Management Classes and Functions	103
Controlling Memory Usage	104
Monitoring Memory Usage	104
Interface Management Classes	105
Base Classes	105
Memory Allocation	105
Name Space	106
The Action Services Library (ASL) API	107
Include Files	107
Classes and Functions	107
Ace Class	110
Ace Constructor	113
drop Method	113
pass Method	114
Action Functions.	115
action_drop Function	116
action_pass Function	116
Custom Action Functions	117
Backlog Class.	119
est Method	120

names Method	120
now Method	121
size Method	122
Buffer Class	123
append Method	126
busy Method	126
decref Method	127
headerBase Method	128
headerType Method	128
incref Method	129
interfaceNum Method	129
interfaceType Method	131
new Operator	131
next Method	132
packetPadHeadSize Method	133
packetPadTailSize Method	133
packetSize Method	133
prepend Method	134
rxTime Method	134
takable Method	135
takable_clr Method	135
takable_max Method	136
takable_min Method	136
takable_set Method	137
trim_head Method	137
trim_tail Method	138
txTime Method	138
Crosscall Class	139
Crosscall Constructor	142
call Method	143
CrosscallHandler Class	144
CrosscallHandler Constructor	147
direct Method	148
DowncallHandler Class	150
DowncallHandler Constructor	153
direct Method	154
Dualobj Class	155
Dualobj Constructor	156
ace Method	156
Dynamic Class	157
delete Operator	158
new Operator	158

Element Class	159
Elt_setname Class	160
Elt_setname Constructor	162
Elt_setname Destructor	162
cancel Method	163
delete Operator	163
expire Method	164
new Operator	165
Event Class	166
Event Constructor	168
Event Destructor	169
cancel Method	169
curr Method	169
direct Method	170
schedule Method	170
Initialization Function	172
init_actions Function	172
Linked Class	174
Linked Constructor	175
Linked Destructor	175
link Method	175
next Method	176
orphan Method	176
prev Method	176
unlink Method	177
Memory Management Functions	178
getmemstatvalues Function	178
mstats Function	179
Message Class	180
Message Constructor	182
Message Access Methods	183
Message Completion Methods	183
MessageBlock Class	184
MessageBlock Constructor	186
Name Class	191
Name Constructor	192
find Method	192
here Method	193
Named Class	194
Named Constructor	195

Named Destructor	195
find Method	196
name Method	196
NBInterfaceProp Class	197
NBInterfaceProp Constructor	200
GetProperty Method	200
GetPropertyList Method	201
NBFIF_GET_SET_PROP_ITEM Structure	201
NBFIF_PROP_CAP_ITEM Structure	202
NBFIF_PROP_CAPS Structure	202
NBFIF_PROP_ITEM Structure	203
SetProperty Method	204
NBLinkwatch Class	205
NBLinkwatch Constructor	206
checkLinks Method	207
NBRmon Class	208
NBRmon Constructor	211
NBRmon Destructor	212
Init Method	212
GetRmonCounters Method	213
GetRXTXStats Method	214
GetQueryRate Method	215
SetQueryRate Method	215
NBStringMatchReport Class	216
NBStringMatchReport Constructor	217
end Method	218
len Method	218
matches Method	219
reports Method	219
sid Method	219
start Method	220
tag Method	221
NBSearchContext Class	222
NBSearchContext Constructor	225
ActiveStrings Method	225
SchedDelete Method	226
SchedReset Method	226
SetOpt Method	227
SetPerBufferCallback Method	228
SetPerMatchCallback Method	230
SetPerResetCallback Method	232
NBStringSearchEngine Class	233

NBStringSearchEngine Constructor	235
AddString Method	236
ChangeOpMode Method	239
OpMode Method	240
RemoveString Method	241
SchedDelete Method	242
SearchBuffer Method	243
Reporting Matches	244
Single- or Multiple-Buffer Searches	244
Pool Class	245
Pool Constructor	245
Pool Destructor	246
free Method	247
take Method	247
Rate Class	248
Rate Constructor	249
add Method	250
count Method	250
clear Method	250
Search Class	251
hit Method	253
insert Method	253
miss Method	254
ran Method	254
toElement Method	254
Set Class	255
Set_setname Class	256
Set_setname Constructor	259
first Method	260
locate Method	260
next Method	261
Tagged Class	262
free Method	263
take Method	264
Target Class	265
Target Constructor	267
take Method	267
Time Class	268
Time Constructor	269
curr Method	270
Access Methods	270

Builder Methods	271
Assignment Operators	272
Conversion Operator	272
Upcall Class	273
Upcall Constructor	276
call Method	276

Chapter 5 ASL Extensions for TCP/IP 279

Classes and Constants in the ASL TCP/IP Extensions	279
General Checksum Support	279
IP Support	280
UDP Support	280
TCP Support	280
Network Address Translation (NAT)	281
Using the TCP/IP Classes	283
Using Header Classes	283
Using Header Classes and NAT	284
Using IP Datagram Classes	287
IP Constant Definitions	293
IP Fragmentation	293
IP Service Type	293
IP Precedence	294
IP Option Definitions	294
IP Options Field Offsets	295
TCP Constant Definitions	295
TCP Control Bits	296
TCP Options	296
TCP Session State	296
TCP Return Codes	298
ASL TCP/IP Extension API	300
Internet Class	302
apasum Method	303
apsasum Method	303
apsum Method	304
apssum Method	304
asum Method	305
cksum Method	305
incrcksum Method	306
psum Method	306
IP4Addr Class	308
IP4Addr Constructor	308
bcast Method	309

mcast Method	309
IP4Datagram Class	310
IP4Datagram Constructor	311
IP4Datagram Destructor	311
checkcksum Method	312
complete Method	312
fragment Method	313
fragmented Method	313
head Method	314
insert Method	314
len Method	315
nfrags Method	315
IP4DNat Class.	316
IP4DNat Constructor	316
rewrite Method	317
IP4Fragment Class	318
IP4Fragment Constructor	319
IP4Fragment Destructor	320
hdr Method	320
buf Method	320
complete Method	320
datalen Method	321
first Method	321
fragment Method	321
next Method	322
optcopy Method	322
payload Method	323
prev Method	323
IP4Header Class	324
cksum Method	325
datalen Method	325
dst Method	325
hl Method	326
hlen Method	326
id Method	326
len Method	327
offset Method	327
optbase Method	327
payload Method	328
proto Method	328
psum Method	328
src Method	329
tos Method	329

ttl Method	329
ver Method	330
vhl Method	330
IP4Mask Class.	331
IP4Mask Constructor	331
bits Method	332
leftcontig Method	332
IP4NAT Base Class.	333
rewrite Method	334
IP4SDNat Class.	335
IP4SDNat Constructor	335
rewrite Method	336
IP4SNat Class.	337
IP4SNat Constructor	337
rewrite Method	338
ReassemblyQueue Class.	339
ReassemblyQueue Constructor	340
add Method	340
clear Method	341
empty Method	341
read Method	341
TCPDNat Class.	343
TCPDNat Constructor	343
rewrite Method	344
TCPEndpoint Class.	346
TCPEndpoint Constructor	347
TCPEndpoint Destructor	347
init Method	347
process Method	348
reset Method	348
state Method	349
TCPHeader Class.	350
ack Method	351
cksum Method	351
dport Method	351
flags Method	351
hlen Method	352
off Method	352
optbase Method	352
payload Method	353
seq Method	353

sport Method	353
urp Method	354
win Method	354
window Method	354
TCPNat Base Class	355
TCPNat Constructor	356
rewrite Method	356
ports Method	357
seqs Method	357
TCPSDNat Class	359
TCPSDNat Constructor	360
rewrite Method	361
TCPSeqInfo Class	362
data Method	362
endseq Method	363
flags Method	363
len Method	363
next Method	363
prev Method	364
segment Method	364
startseq Method	364
TCPSeq Class	365
TCPSeq Constructor	366
image Method	366
val Method	367
TCPSeq Operators	367
TCPSession Class	368
TCPSession Constructor	368
TCPSession Destructor	369
process Method	369
client Method	370
server Method	370
TCPSNat Class.	371
TCPSNat Constructor	371
rewrite Method	373
UDPDNat Class.	374
UDPDNat Constructor	374
rewrite Method	375
UDPHeader Class	376
cksum Method	376
dport Method	377

len Method	377
payload Method	377
sport Method	377
UDPNat Base Class	378
UDPNat Constructor	379
rewrite Method	379
ports Method	380
UDPSDNat Class	381
UDPSDNat Constructor	381
rewrite Method	382
UDPSNat Class.	384
UDPSNat Constructor	384
rewrite Method	385

Chapter 6 **Network Classification Language 387**

Overview	387
See Also	388
NCL Rules File Structure and Elements	388
Include Files	389
Examples	389
Symbolic Constants	389
Value Formats	390
Comments	390
Names	391
Keywords	392
Operators	392
Arithmetic Operators	393
Logical and Relational Operators	393
Bit-wise Operators	394
Precedence	394
Protocol Definitions	395
Example Protocol Definition	395
Using the Built-in TCP/IP Protocol Definition	396
Intrinsic Functions	397
Defining Protocol Fields	398
Examples	399
Identifying Nested Protocols	399
Example	400
Extending Protocol Definitions	401
Adding Fields	401
Adding Predicates	401
Predicate Definitions	402

Example	403
Sets and Named Searches	403
Defining a Set	403
Choosing the Size Hint	404
Defining Named Searches	404
Executing Searches	405
Examples	406
Rules and Actions	406
Defining Rules	407
Passing Action Arguments	408
Example	409
Conditional Rule Execution	410
Example	410
Synchronizing NCL with Action Code	411
Generating Sets and Searches	411
Generating Field Accessors	412

Chapter 7	Command-Line Tools	413
	Tool Locations	413
	cecomp Command	414
	celink Command	417
	getaceid Command	418
	nbgcc Command	419
	nbgdb Command	421
	Stepping through Action Functions	422
	Shutting down the Debugger	422
	nbld Command	423
	odxloop Command	424
	pal00diag Command	425
	readport Command	426
	resolver Command	427
	Starting and Stopping the Resolver	428
	Stopping and Restarting Applications	428
Appendix A	IX-API SDK Host API Error Codes	429
	Alphabetical Listing	429
Appendix B	IX-API SDK File Types	439
	File Types and Extensions	439
Appendix C	Policy Accelerator Name Space	441
	Overview	441

Object Name Syntax	442
System Names for Policy Accelerator Interfaces	444
System ACE Names	444
Policy Accelerator Names	445
Policy Accelerator Interface Names	445
Example	446

Glossary.	447
------------------	------------

Index.	457
---------------	------------



About This Reference



This reference manual provides network application developers with detailed information about the Intel® IX-API SDK software developer's kit (SDK). It describes the components of the SDK, its classes and methods, and their syntax and use. Its companion document, *Developing Applications Using the IX-API SDK*, describes how to plan and create applications using the SDK.

Audience

This reference is for network application developers who are creating applications for use with the Intel IX-API SDK. It assumes that you are familiar with the following:

- C++ programming
- A development environment compatible with supported compilers
 - For the Windows NT platform, Microsoft Visual Studio®
 - For UNIX platforms, a compatible development environment of your choice
- Realtime network applications

In This Reference

This reference includes the following chapters and appendices:

- **Chapter 1, “Overview,”** provides a general introduction to IX-API SDK applications and the Intel system.
- **Chapter 2, “System Types and Methods,”** provides information on the system-wide numeric data types and support classes that you use in both the host and accelerator modules.

- **Chapter 3, “Host API,”** provides a complete reference to the IX-API SDK host API. It describes the classes you use to create and configure Action/Classification Engines (ACEs), bind targets, and pass messages to and from the Policy Accelerator. You use the host API to write the host module for an application.
- **Chapter 4, “Action Services Library,”** provides a complete reference to the ASL. It describes the C++ library classes and functions supplied with the IX-API SDK that you use to write the action part of an ACE in the accelerator module.
- **Chapter 5, “ASL Extensions for TCP/IP,”** provides a complete reference to the ASL TCP/IP extensions, a set of class definitions you can use to perform tasks common to TCP/IP-based network-oriented applications.
- **Chapter 6, “Network Classification Language,”** provides a complete reference for NCL, the language in which you write the classification part of an ACE in the accelerator module.
- **Chapter 7, “Command-Line Tools,”** provides a complete reference for executable command-line tools you use to compile, link, and debug the parts of an IX-API SDK application.
- **Appendix A, “IX-API SDK Host API Error Codes,”** provides a table of all IX-API SDK error codes in alphabetical order along with a description for each.
- **Appendix B, “IX-API SDK File Types,”** provides a table of the source and object code file types used by the IX-API SDK, as identified by their filename extensions.
- **Appendix C, “Policy Accelerator Name Space,”** provides a complete reference for the naming and reference conventions that the Resolver uses to manage relations between objects and entities on the host and on the Policy Accelerator.
- **“Glossary”** provides definitions of Intel and C++ programming terms.

Other Sources of Information

This guide is part of the Intel IX-API SDK documentation set, which also includes:

- *Developing Applications Using the IX-API SDK*, which provides programmers with conceptual descriptions and instructions on writing network policy-enforcement applications using the IX-API SDK
- *IX-API SDK Release Notes*, which lists information about the latest software release

- *Installing the IX-API SDK*, which describes how to install both the run-time and the development versions of the IX-API SDK
- *Installing a Policy Accelerator 100 Board*, which describes how to install a Policy Accelerator PCI board into a PC
- *Customizing a NIC Driver Using the ODX Protocol*, which describes how to use the optimal data exchange (ODX) protocol for PCI to customize your standard NIC driver for communication with the Policy Accelerator through a direct PCI bus interface

In addition, the Intel Web site provides valuable information on products, support, and the company. See “Contacting Intel” on page xxii.

Typographical Conventions

This document uses the following typographic conventions to help you locate and identify information:

Italic text Used for new terms, emphasis, and book titles; also identifies arguments in syntax descriptions.

Bold text Identifies keywords and punctuation in syntax descriptions.

`Courier font` Identifies file names, folder names, and text that either appears on the screen or that you are required to type.



NOTE: Provides extra information, tips, and hints regarding the topic.



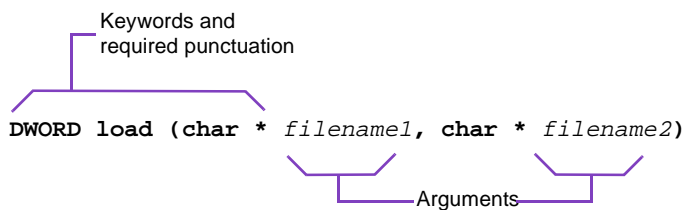
CAUTION: Identifies important information about actions that could result in damage to or loss of data or could cause the application to behave in unexpected ways.



WARNING! Identifies critical information about actions that could result in equipment failure or bodily injury.

Syntax Example

The following figure shows a sample syntax notation.



Contacting Intel

You can reach Intel's automated support services 24 hours a day, every day at no charge. The services contain the most up-to-date information about Intel products. You can access installation instructions, troubleshooting information, and general product information.

Web and Internet Sites

You can use the internet to download software updates, troubleshooting tips, installation notes, and more.

- General online support services are on the World Wide Web at:

<http://support.intel.com>

- Online support services for the Policy Accelerator 100 are on the World Wide Web at:

<http://support.intel.com/support/network/adapter/pa/pa100/>

For specific types of information and services, go to the following Web and internet sites:

- **Corporate:** <http://www.intel.com>
- **Network Products:** <http://www.intel.com/network>
- **Intel IX Information:** <http://developer.intel.com/design/ixa/>
- **IX-API SDK:** <http://developer.intel.com/design/ixa/software/index.htm>
- **Policy Accelerator:** <http://developer.intel.com/design/ixa/pa100/index.htm>
- **ASIC:** <http://128.11.21.45/scripts/mardev/product/ixe100.asp>
- **FTP Host:** download.intel.com

**Customer
Support
Technicians**

- **FTP Directory:** /support/network/adapter

- **United States and Canada:** 1-916-377-7000 (7:00 - 17:00 M-F Pacific Time)

- **Worldwide Access:** Intel has technical support centers worldwide. Many of the centers are staffed by technicians who speak the local languages. For a list of all Intel support centers, their telephone numbers, and the times they are open, go to:

<http://support.intel.com/support/9089.htm>

Contacting Intel

Chapter 1

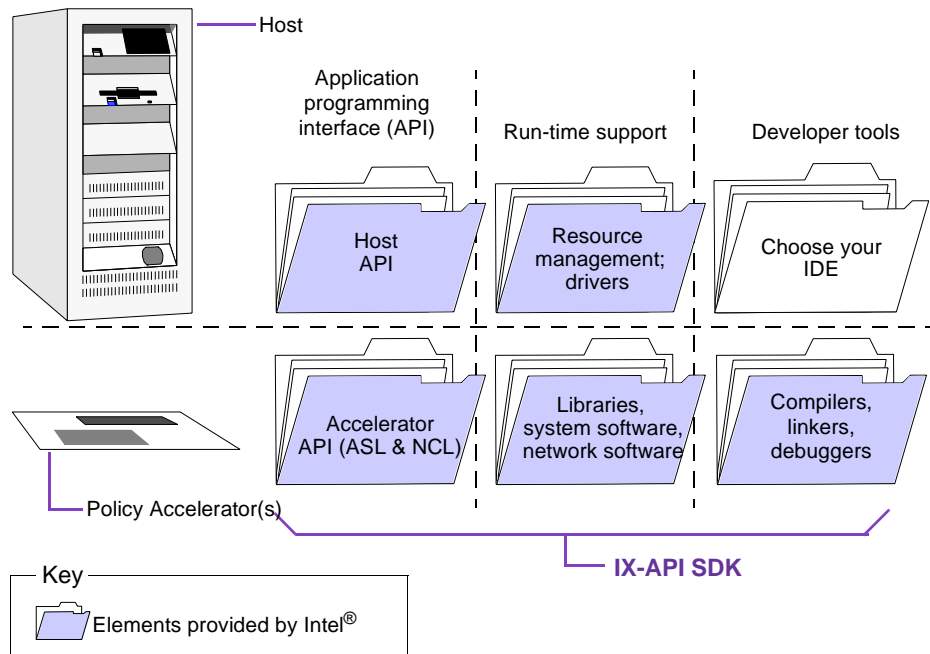
Overview



The Intel® IX-API SDK consists of both hardware and software components. The main components are:

- Policy Accelerator boards
- IX Software Developer's Kit (SDK)

As an application developer, you use the SDK to customize network applications to work on the Policy Accelerator. The Policy Accelerator enables network applications to process packets at wire speed. IX-API SDK application code is distributed between the host computer and the resident Policy Accelerator. You develop separate but related modules using the SDK.



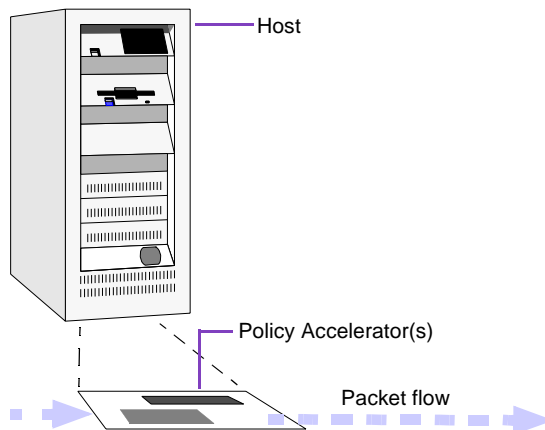
About the Host System

The host is a computer system in which you have installed the Policy Accelerator board. The host system runs and controls a policy-enforcement application through a resource manager that runs as a background process. This resource manager, called the Resolver, starts automatically when you start the host computer, and is always running in the run-time environment. It need not be running in the development environment.

The host provides basic services to your application. It handles exceptions and communicates set-up and modification information about packet handling to the Policy Accelerator. Each host can be equipped with one or more Policy Accelerators.

About the Policy Accelerator

A Policy Accelerator uses rules, which you define to implement policies, to classify packets and to determine the corresponding actions. Packets flow through the Policy Accelerator, which automatically applies the rules and performs the actions.



About the Application Programming Interface (API) of the SDK

The Intel IX API includes the following, which this document describes:

- Global types and methods

Data types and converters to ensure compiler-independent data integrity across network connections. Use these in both the host and Policy Accelerator portions of your application. See Chapter 2, “System Types and Methods.”

- **Host API**
A set of C++ classes for the host portion of your application. These classes provide the ability to load software onto, initialize, and manage the Policy Accelerator. See Chapter 3, “Host API.”
- **Action Services Library (ASL) for the Policy Accelerator**
A set of C++ classes and functions for the Policy Accelerator portion of your application. This library provides efficient implementation for common packet-manipulation tasks. See Chapter 4, “Action Services Library.”
- **ASL Extensions for TCP/IP for the Policy Accelerator**
A set of class definitions that help with tasks common to TCP/IP-based applications, such as accessing parts of packets, IP fragment reassembly and TCP stream reconstruction. See Chapter 5, “ASL Extensions for TCP/IP.”
- **Network Classification Language (NCL™) for the Policy Accelerator**
A special-purpose language in which you define rules for implementing your company’s policies. Your NCL rules classify packets and direct the actions to be taken. See Chapter 6, “Network Classification Language.”
- **Programming tools**
A set of utilities for compiling, linking, and debugging the different parts of an IX-API SDK application. See Chapter 7, “Command-Line Tools.”

Parts of an IX-API SDK Application

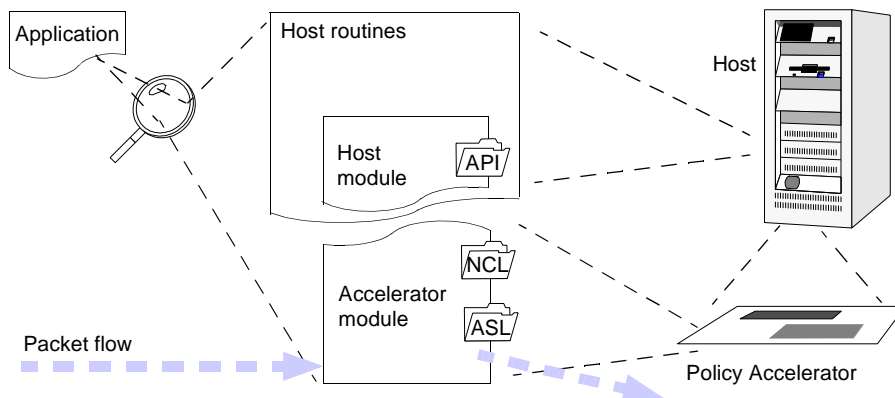
When you create an application for the Policy Accelerator, you use the API in the following modules:

- The *host module*, a C++ application that runs on the host and does the following:
 - Uses the IX-API SDK host application programming interface (API) to initialize and communicate with the Policy Accelerator
 - Uses standard host services for other operations

You compile this portion of the application with Microsoft Visual C++ (on NT) or a standard ANSI C++ compiler (on UNIX).
- The *accelerator module*, which contains two parts that run on the Policy Accelerator:
 - *Classification code* uses the Network Classification Language (NCL) to classify packets. This portion of the application is compiled at run time in the Policy Accelerator.

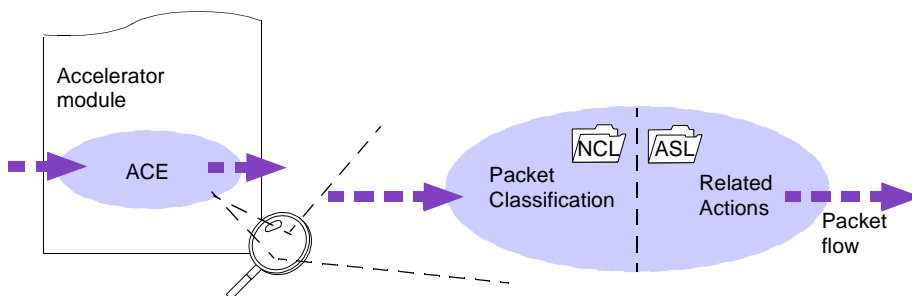
- *Action code* uses C++ with the Action Services Library (ASL) and its extensions to act on packets. You compile this portion of the application with a special version of the `gcc` compiler, `nbgcc`.

The following figure shows the structure of an IX-API SDK application:

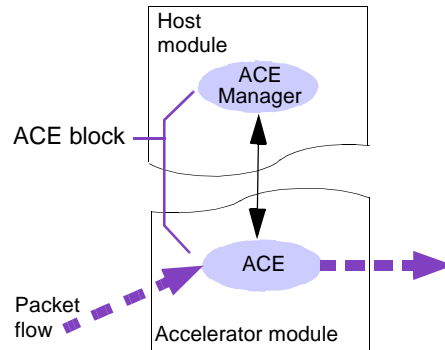


Packet Processing Units

The basic packet-processing unit of an IX-API SDK application is an Action/Classification Engine, or ACE. An application can have more than one ACE. Each ACE is managed by the single host module, but is associated with its own accelerator module files. When the host module initializes an ACE, it downloads the associated action and classification code files to the Policy Accelerator.



Each ACE is associated with an `AceManager` object on the host side, and an `Ace` object on the Policy Accelerator side (defined in the action file). The combination of an ACE and an ACE manager is known as an *ACE block*. For more information about managed objects, see “Object Pairing on the Host and Policy Accelerator” on page 27.

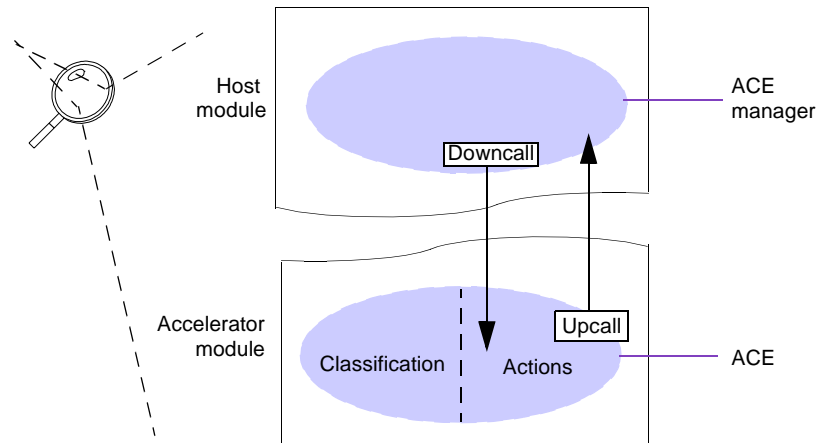


Packets flow into and out of an ACE according to the way you *bind* its *targets*. All ACEs (including the system-defined ACEs that represent the Policy Accelerator interfaces) contain two system-defined targets named `pass` and `drop`. When you bind an ACE's `pass` target to an interface ACE, the application's action functions can pass packets to that interface. You can define additional targets as well. A target, like an ACE, has a `TargetManager` object in the host module and a `Target` object in the accelerator module.

For more information about binding and targets, see Chapter 5, “Controlling Packet Flow,” in *Developing Applications Using the IX-API SDK*.

Messages Between the Host and Policy Accelerator

The ACE and its manager can communicate by passing messages to each other. You use *upcalls* and *downcalls* to share information in a manner similar to asynchronous remote procedure calls, as shown in the following figure:



Upcalls and downcalls are also represented by objects on both the host and Policy Accelerator sides of the application. For example, to send a message in an upcall you use an `Upcall` object in the accelerator module. To receive and process the message, you use a corresponding `UpcallHandler` object on the host.

You define ACE and ACE manager subclasses to contain references to these objects, as well as the methods that will create, send, and process messages as needed by your application.

Because messages travel between the Policy Accelerator and the host computer, which might use different techniques for representing multibyte information, you must package a message to preserve the byte order, and unpack it on the other side. For more information, see “Byte Order Issues” on page 10.

For more information about message passing, see Chapter 8, “Communication Within an Application,” in *Developing Applications Using the IX-API SDK*.

Asynchronous Calls and Callback Functions

Message passing operations, and certain other operations such as searching for strings in packets, are executed partly on the host and partly on the Policy Accelerator. These operations are asynchronous—that is, processing continues in the foreground while a separate process or thread executes the operation in the background. You can specify *callback functions* when you initiate these operations. The callback function that you specify is executed when the operation is complete.

Creating an IX-API SDK Application

The following sections show the basic steps for building an IX-API SDK application, starting with the order in which you create objects on the host side. For a more complete, step-by-step example, see Chapter 2, “Tutorial: Creating a Simple Application,” in *Developing Applications Using the IX-API SDK*.

Creating the Host Module

In the host module file named `YourAppName.cpp`:

1. Define a subclass of `NBApp1` (the main class for your IX-API SDK application). When constructing this subclass, you can have it initialize other objects such as ACE managers.
2. Define at least one subclass of an `AceManager` (the object on the host that communicates with the ACE on the Policy Accelerator). This subclass should contain state variables and methods needed to process ACE information on the host side; for example, to receive messages that the accelerator module side of the ACE sends to the host.

3. Define a subclass of an `AceGroup` for each group of ACE managers. You can group ACE managers by function or by their corresponding Policy Accelerator board.
4. Use the `load` method of the `AceManager` class to download the ACE's accelerator module files into the Policy Accelerator.
5. Use the `bind` method of the `NBApp1` class to define the packet flow through the application's ACEs.

Creating the Accelerator Module

The accelerator module consists of two files:

1. In `YourAppName.ncl` rules file:
 - a. Define the packet classification rules.
These rules describe what to look for in packets and specify what actions to perform on packets in the Policy Accelerator.
2. In `YourAppName.cpp` actions file:
 - a. Define a subclass of the `Ace` class that contains the state variables and methods needed to process packets in your application. The `Ace` constructor's arguments identify the host-side `AceManager` object with which it corresponds.
 - b. Write the initialization function for the ACE, which creates the Policy Accelerator side of the ACE in response to the download of that ACE's code files.
 - c. Write action code to implement the actions specified by the rules. The action functions are the entry points to the ACE's methods or any other utility functions you provide.

For More Information

This overview provides only a simple introduction to the IX-API SDK. For more detailed information, see *Developing Applications Using the IX-API SDK*.

For More Information

Chapter 2

System Types and Methods



This chapter describes the data types, classes, and methods that both the host and accelerator modules use.

This chapter contains the following sections:

- Overview
- Data Types
- Byte Order Issues
- Byte Order Classes
- Byte Order and Intermodule Communication
- The Data Type API

Overview

The IX-API SDK defines numeric data types and classes that you use in both the host and accelerator modules. Using the SDK data types ensures that numeric variables are of the precise type needed despite possible differences in how compilers handle standard data types.

The byte-order classes and conversion methods provide a mechanism for dealing with possible differences in how multibyte values are handled among the various processors that touch the data.

Include Files

To use these classes and data types, include the following header file in all of your code files, in both the host module and accelerator module:

```
#include <NBtypes.h>
```

Data Types

The C programming language does not guarantee any particular width for its basic data types. However, network programming requires that you use quantities of specific known lengths. The coding sequences required to guarantee the precise width of variables might differ among compilers, and different compilers might use signed or unsigned variants of basic numeric types.

The SDK provides unsigned and fixed-width numeric data types to ensure compiler-independent data integrity.

- The following type definitions are for specifically unsigned variants of the standard basic C types:

```
typedef ... u_char;
typedef ... u_short;
typedef ... u_int;
typedef ... u_long;
```

- The following type definitions are for fixed-width integers:

```
typedef ... int8;
typedef ... int16;
typedef ... int32;
typedef ... int64;
typedef ... uint8;
typedef ... uint16;
typedef ... uint32;
typedef ... uint64;
```

- The NT-specific type `DWORD` is defined by the Policy Accelerator system as `uint32`.
- The NT-specific type `PCHAR` is defined by the Policy Accelerator system as the standard C type `char *`.

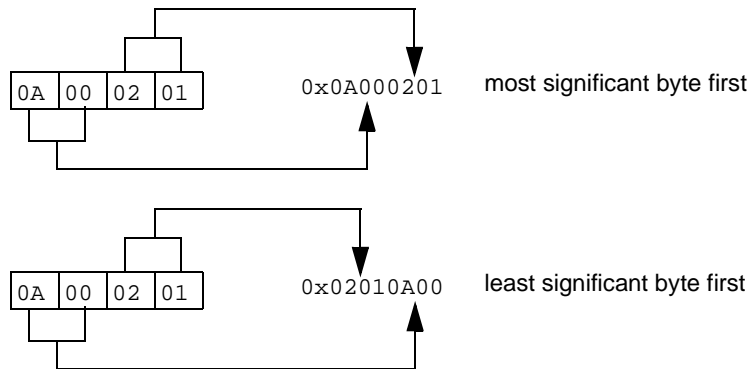
Byte Order Issues

There are two ways to handle storage of data items larger than one byte: with the least significant byte first, or with the most significant byte first. Compilers identify these styles as *little-endian* and *big-endian*. Most network data is organized most-significant byte first (big-endian), so this style is called *network byte order*. The Policy Accelerator generally uses network byte order.

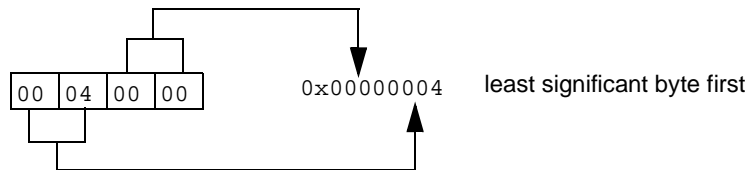
Different platforms use different byte orders. Big-endian processors, in which words are stored with the most significant byte at the lowest address, include IBM 370, SPARC, MIPS and ARM. Little-endian processors include x86 and

Pentium. In the context of IX-API SDK application programming, the byte order used by the host, or source of data, is called *host byte order*, regardless of whether it is the same as or different from network byte order.

When data is passed over a network, a difference in how the processors handle the byte order can lead to false interpretation of the data. For example, suppose a typical IP address of 10.0.2.1 is stored in the network packet as four bytes: 0x0A, 0x00, 0x02 and 0x01. When observed by a RISC processor as a four-byte integer, the hex value is 0x0A000201. However, when observed by a Pentium processor, the value is 0x0102000A—not what was intended.



Similarly, if a packet size of 1024 is stored in a network stream, it appears as the bytes 0x04, 0x00. An Intel CPU would observe this (incorrectly) as 0x0004.



Byte Order Classes

You are responsible for ensuring that the data you pass between the host and accelerator modules, as well as data that you receive from, process, and return to the network, is in a known byte order.

To ensure that data is transmitted accurately regardless of byte order, the SDK includes two classes that explicitly store 16-bit and 32-bit network-ordered data.

Class	Description
<code>nuint16</code> Class	Accesses 16-bit data that is stored in network byte order.
<code>nuint32</code> Class	Accesses 32-bit data that is stored in network byte order.

You can use these classes as if they were additional data types.

The byte-order classes represent data that is stored in network byte order, or that is in a CPU register after having been loaded from such a storage location without applying conditional byte swapping primitives.

The byte-order classes contain methods that convert values between host-ordered representation and network-ordered representation. The classes have four data transformation methods, with names that are familiar to networking programmers:

```
uint32  ntohl (uint32 n); // net to host (32 bit)
uint16  ntohs (uint16 n); // net to host (16 bit)
nuint32 htonl (uint32 h); // host to net (32 bit)
nuint16 htons (uint16 h); // host to net (16 bit)
```

The classes also have the following methods for explicitly switching the byte order of 16-bit or 32-bit values:

```
uint32 swapl (uint32 n);
uint16 swaps (uint16 n);
```

All six of these methods are real functions that have macro equivalents you can use when the result must be a compile-time constant:

```
#define NTOHL (x)...
#define NTOHS (x)...
#define HTONL (x)...
#define NTOHS (x)...
#define SWAPL (x)...
#define SWAPS (x)...
```

**Byte Order and
Intermodule
Communica-
tion**

In a IX-API SDK application, the Policy Accelerator uses network byte order, while the host might or might not do so. The host and accelerator modules might disagree on the byte order issue, both with each other and with the network.



NOTE: In the context of byte order, host means the sender of the data, and network means the transport mechanism. The host module of your IX-API SDK application might or might not be the sender of the data.

In communication between the host module and the accelerator module (upcalls and downcalls), as well as when interpreting network protocol fields, you must take into account of the possibility of different byte orders.

Upcalls, Downcalls, and Byte Order

To pass data between the host and accelerator modules you encapsulate it in a message and send the message in a downcall or upcall. To maintain the integrity of the data despite possible differences in byte order, use the byte-order operators to *marshal* the arguments that carry the data when you create the message.

To marshal arguments, you serialize and convert them to ensure that they are in a standard byte order before passing them to the message constructors.

In the following example a method creates a message and sends it in an upcall, first marshalling the arguments using the conversion function `htonl`:

```
void NBBasicAce::peekPacketUpcall (Buffer *buf) {
    buf = buf; /* prevent "buf not used" compiler warning */
    packetCounter++;
    if ((packetCounter % 100) == 0) {
        msg = htonl (packetCounter);
        MessageBlock b ((char *)&msg, sizeof (msg));
        Message m (b);
        peekPacketUpcallHandle.call (&m);
    }
}
```

The following upcall handler callback converts the arguments back to host order, using the conversion function `ntohl`:

```
void NBBasicAce::peekPacketUpcall (Message* m) {
    NB_ASSERT (m->getLen1 () == sizeof (nuint32));
    printf ("NoOfPackets: %05d\n",
            ntohl (* (nuint32 *) m->getBuffer1 ()));
    releaseMessage (m);
}
```

For more information on passing and receiving messages, see Chapter 8, “Communication Within an Application,” in *Developing Applications Using the IX-API SDK*.

Operators and Byte Order

In most cases, numeric and logical operators infer significance about bytes within words of their operands. You must convert the two network-ordered operands of such an operator to host order before applying the operation, and convert the result from host order back to network order.

In some cases, you can apply operators between two `nuint16` expressions or two `nuint32` expressions without converting to host order. This is true for operations where byte order is not important, such as determining equality or inequality.

The following table lists operations that you can apply without converting the operands to host order, or the result back to network order.

Operator type	Byte-order independent operations	Syntax
Unary	inversion	<code>nuint16 operator ~ () const;</code> <code>nuint32 operator ~ () const;</code>
	not	<code>bool operator ! () const;</code>
Comparison	equality	<code>bool operator == (nuint16 y) const;</code> <code>bool operator == (nuint32 y) const;</code>
	inequality	<code>bool operator != (nuint16 y) const;</code> <code>bool operator != (nuint32 y) const;</code>
Binary	AND	<code>nuint16 operator & (nuint16 y) const;</code> <code>nuint32 operator & (nuint32 y) const;</code>
	OR	<code>nuint16 operator ^ (nuint16 y) const;</code> <code>nuint32 operator ^ (nuint32 y) const;</code>
	XOR	<code>nuint16 operator (nuint16 y) const;</code> <code>nuint32 operator (nuint32 y) const;</code>
Assignment	AND	<code>nuint16 & operator &= (nuint16 y);</code> <code>nuint32 & operator &= (nuint32 y);</code>
	OR	<code>nuint16 & operator ^= (nuint16 y);</code> <code>nuint32 & operator ^= (nuint32 y);</code>
	XOR	<code>nuint16 & operator = (nuint16 y);</code> <code>nuint32 & operator = (nuint32 y);</code>

The Data Type API

This section provides a detailed description of the data type classes. Within each class, the constructor and destructor for that class are listed first, followed by the remaining methods in alphabetical order.

The Data Type API contains the following classes:

Class	Description
<code>nuint16</code> Class (page 16)	Access 16-bit data that is stored in network byte order.
<code>nuint32</code> Class (page 20)	Access 32-bit data that is stored in network byte order.

nuint16 Class

Use the `nuint16` class to access 16-bit data that is stored in network byte order. To provide type-checked access to the network data, cast pointers to 16-bit network data into pointers to `nuint16` objects.

The `nuint16` class contains the following methods:

Method	Description
<code>nuint16</code> Constructor	Initializes and constructs <code>nuint16</code> objects.
<code>htons</code> Method	Converts a value from a host-ordered representation to a network-ordered representation, returning an object of the <code>nuint16</code> class.
<code>ntohs</code> Method	Converts a value from a network-ordered representation (in an <code>nuint16</code> object) into a host-ordered representation, returning a value of type <code>uint16</code> .
<code>raw_</code> Member	An unsigned 16-bit integer that contains the data being held by the <code>nuint16</code> , stored in network byte order.
<code>swaps</code> Method	Switches the byte order of a <code>uint16</code> value.

The `nuint16` class is not derived from any other class.

Examples

All of the following examples result in the value with bytes 08 00.

```
const nuint16 ether_type_ip (0x0800); /* implicit conversion */

const nuint16 my_type (ether_type_ip); /* copy constructor */

nuint16 ether_type;
ether_type = htons (0x0800); /* explicit conversion */
printf ("ether type is %04X\n", ntohs (ether_type));

void
use_type (uint16 type) /* type really in network order */
{
    nuint16 my_type (type, true); /* init without swapping */
    printf ("my type is %04X\n", ntohl(type));
}
```


nuint16 Constructor

Initializes and constructs `nuint16` objects.

```
nuint16();  
  
nuint16 (const nuint16 &dup);  
  
nuint16 (uint16 val);  
  
nuint16 (uint16 image,  
         bool dummy);
```

Argument	Description
<code>dup</code>	Reference to existing object to duplicate.
<code>val</code>	16-bit value to store after swapping (if appropriate) to network order.
<code>image</code>	16-bit image in network order to store without swapping.
<code>dummy</code>	Distinguishes between the <code>val</code> and <code>image</code> versions of the constructor. Pass either 0 or 1. The presence of this argument is significant; its value is not.

Returns A reference to the newly created object.

Description The class has constructors that allow you to create objects with or without initial values.

To do this:	Use this form:
Construct an uninitialized <code>nuint16</code> object	No arguments
Copy another <code>nuint16</code> object	<code>dup</code> argument
Specify a standard 16-bit value as the initial value, swapping the byte order if necessary.	<code>val</code> argument
Initialize with a standard 16-bit value that you know to be in network byte order	<code>image</code> and <code>dummy</code> arguments

htons Method

Converts a value from a host-ordered representation to a network-ordered representation, returning an object of the `nuint16` class.

```
nuint16 htons(uint16 val);
```

Argument	Description
val	A value to be converted.

Returns An object of the `nuint16` class.



NOTE: In the context of byte order, host means the sender of the data, and network means the transport mechanism. The host module of your IX-API SDK application might or might not be the sender of the data.

ntohs Method

Converts a value from a network-ordered representation (in an `nuint16` object) into a host-ordered representation, returning a value of type `uint16`.

```
uint16 ntohs (nuint16 val);
```

Argument	Description
val	A value to be converted.

Returns A value of type `uint16`.



NOTE: In the context of byte order, host means the sender of the data, and network means the transport mechanism. The host module of your IX-API SDK application might or might not be the sender of the data.

raw_ Member

Contains the data held by the `nuint16` object.

```
uint16 raw_;
```

Description This data field is an unsigned 16-bit integer that contains the data being held by the `nuint16`, stored in network byte order.

swaps Method

Switches the bytes of a value of type `uint16`, returning a value of type `uint16` with the opposite byte order.

```
uint16 swaps (uint16 val);
```

Argument	Description
val	A value to be swapped.

Returns A value of type `uint16`.

nuint32 Class

Use the `nuint32` class to access 32-bit data that is stored in network byte order. To provide type-checked access to the network data, cast pointers to 32-bit network data into pointers to `nuint32` objects.

The `nuint32` class contains the following methods:

Method	Description
<code>nuint32</code> Constructor	Initializes and constructs <code>nuint32</code> objects.
<code>htonl</code> Method	Converts a value from a host-ordered representation to a network-ordered representation, returning an object of the <code>nuint32</code> class.
<code>ntohl</code> Method	Converts a value from a network-ordered representation (in an <code>nuint32</code> object) into a host-ordered representation, returning a value of type <code>uint32</code> .
<code>raw_</code> Member	An unsigned 32-bit integer that contains the data being held by the <code>nuint32</code> , stored in network byte order.
<code>swapl</code> Method	Switches the byte order of a <code>uint32</code> value.

The `nuint32` class is not derived from any other class.

Examples

All of the following examples result in the value with bytes 0A 00 02 41.

```
const nuint32 my_ip_addr (0x0A000241); /* implicit conversion */
const nuint32 an_addr (my_ip_addr); /* copy constructor */
nuint32 some_addr;
some_addr = htonl (0x0A000241); /* explicit conversion */

void use_addr (uint32 addr) /* addr really in network order */
{
    union {
        nuint32 addr;
        unsigned char oct[4];
    } u;
    u.addr = nuint32 (addr, true); /* init without swapping */
    printf ("ip addr is 0x%08X (%d.%d.%d.%d)\n",
        ntohl (u.addr),
        u.oct[0], u.oct[1], u.oct[2], u.oct[3]);
}
```

nuint32 Constructor

Initializes and constructs `nuint32` objects.

```
nuint32 ();

nuint32 (const nuint32 &dup);

nuint32 (uint16 ext);

nuint32 (uint32 val);

nuint32 (uint32 image,
         bool dummy);
```

Argument	Description
<code>dup</code>	Reference to existing object to duplicate.
<code>ext</code>	16-bit network data to convert to 32-bit network data.
<code>val</code>	32-bit value to store after swapping (if appropriate) to network order.
<code>image</code>	32-bit image in network order to store, without swapping.
<code>dummy</code>	Distinguishes between the <code>val</code> and <code>image</code> versions of the constructor. Pass either 0 or 1. The presence of this argument is significant; its value is not.

Returns A reference to the newly created object.

Description The class has constructors that allow you to create objects with or without initial values.

To do this:	Use this form:
Construct an uninitialized <code>nuint32</code> object	No arguments
Copy another <code>nuint32</code> object	<code>dup</code> argument
Specify a standard 32-bit value as the initial value, swapping the byte order if necessary.	<code>val</code> argument
Initialize with a standard 32-bit value that you know to be in network byte order	<code>image</code> and <code>dummy</code> arguments

Example

```

nuint16 fred (1024);
    /* fred is 1024, stored as 0x04 0x00 */
nuint32 dave (fred);
    /* dave is also 1024, stored as 0x00 0x00 0x04 0x00 */

```

htonl Method

Converts a value from a host-ordered representation to a network-ordered representation, returning an object of the `nuint32` class.

```
nuint32 htonl (uint32 val);
```

Argument	Description
val	A value to be converted.

Returns

An object of the `nuint32` class.



NOTE: In the context of byte order, host means the sender of the data, and network means the transport mechanism. The host module of your IX-API SDK application might or might not be the sender of the data.

ntohl Method

Converts a value from a network-ordered representation (in an `nuint32` object) into a host-ordered representation, returning a value of type `uint32`.

```
uint32 ntohl (nuint32 val);
```

Argument	Description
val	A value to be converted.

Returns

A value of type `uint32`.



NOTE: In the context of byte order, host means the sender of the data, and network means the transport mechanism. The host module of your IX-API SDK application might or might not be the sender of the data.

raw_Member

Contains the data held by the `nuint32` object.

```
uint32 raw_;
```

Description This data field is an unsigned 32-bit integer that contains the data being held by the `nuint32`, stored in network byte order.

swapl Method

Switches the bytes of a value of type `uint32`, returning a value of type `uint32` with the opposite byte order.

```
uint32 swapl (uint32 val);
```

Argument	Description
val	A value to be swapped.

Returns A value of type `uint32`.

Chapter 3

Host API



Overview

The host part of the application programming interface (API) is a set of C++ classes you use to create and configure *Action/Classification Engines* (ACEs), bind targets, and pass messages between the Policy Accelerator board and the host. Use this API to develop your host module, which is the part of your application that runs on the host.

The first part of this chapter introduces the classes by functional area and explains their relationship to classes in the *Action Services Library* (ASL):

- Host API Class Organization (page 26)
- Object Pairing on the Host and Policy Accelerator (page 27)
- Application and ACE Management Classes (page 28)
- Message Support Classes (page 29)
- Base Class (page 30)
- Error Handling in the Host API (page 30)

The second part of this chapter describes the classes and their methods in detail, in alphabetical order:

- Host API Reference (page 31)

Include Files

To use the host API classes, include the following header file in your code:

```
#include "nbapi\nbappl.h"
```

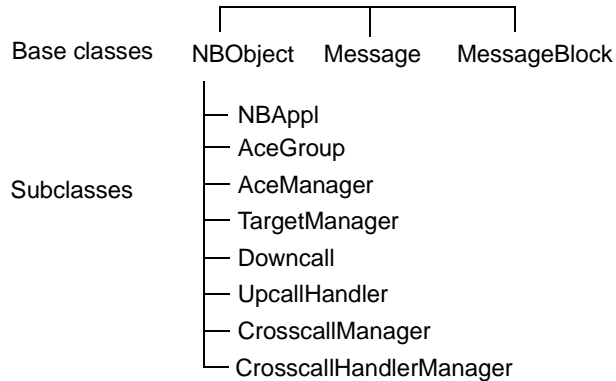


Host API Class Organization

The host API classes fall into the following functional categories:

Category	Classes
Application and ACE Management Classes	<ul style="list-style-type: none"> ■ NBSApp1 Class ■ AceGroup Class ■ AceManager Class ■ TargetManager Class
Message Support Classes	<ul style="list-style-type: none"> ■ CrosscallManager Class ■ CrosscallHandlerManager Class ■ Downcall Class ■ Message Class ■ MessageBlock Class ■ UpcallHandler Class
Base Class	<ul style="list-style-type: none"> ■ NBOBJECT Class

The following figure shows the inheritance tree for the host API classes.



Object Pairing on the Host and Policy Accelerator

Many logical entities in an IX application must use data resident in both the host and the Policy Accelerator. These are represented by *dual objects*. Each dual object resident in the Policy Accelerator is paired with a corresponding object in the host.

The following manager classes on the host are paired with managed object classes on the Policy Accelerator. For these pairs, the host object initializes and manages the structure, keeping track of links and control information:

Host Class	Policy Accelerator Class
AceManager Class	Ace Class
CrosscallManager Class	Crosscall Class
CrosscallHandlerManager Class	CrosscallHandler Class
TargetManager Class	Target Class

In addition, the following message passing classes are paired. In these cases, one side of the pair passes the message, and the other receives the message and directs it to a handler function:

Host Class	Policy Accelerator Class
Downcall Class	DowncallHandler Class
UpcallHandler Class	Upcall Class

For more information on the Policy Accelerator classes, see Chapter 4, “Action Services Library.”

Dictionary Names

The Resolver (a process that always runs in the background) keeps an object name dictionary to track the host and Policy Accelerator portions of IX applications. To associate paired objects with each other, assign them the same dictionary name. You specify the dictionary name in the *name* or *argName* argument when constructing the object. This name can be any string of valid characters. Valid characters are:

A-Z a-z _ [] | - () +

The shared dictionary name of a pair of objects must be unique within the containing entity:

- The name of an ACE group must be unique within the application.
- The shared name of an ACE and ACE manager object must be unique within the ACE group.
- The shared names of upcall, downcall, crosscall, and target objects, and of their handler and manager objects, must be unique within the ACE.

You use the dictionary names of application and ACE objects when binding targets (see “bind Method” on page 72) and when linking crosscalls with their handlers (see “link Method” on page 77). Your application does not otherwise refer to an object using the dictionary name. It generally uses the object handle that you assign to the object on creation.

For more information on naming, see Appendix C, “Policy Accelerator Name Space.”

Application and ACE Management Classes

An Action/Classification Engine (ACE), contains a set of packet classification criteria and associated actions, upcall and downcall entry points, and targets. Applications use ACEs to process packets. ACEs are local to an application and are not shared among applications.

The following classes represent an application and ACEs on the host side:

Class	Description
NBApp1 Class	The main object of a policy enforcement application. Provides services to manage ACE setup. Every application must contain exactly one application object.
AceGroup Class	Containers that hold one or more ACE managers on the host. Every application must contain at least one ACE group object.
AceManager Class	Manages and controls an ACE. Every application must contain at least one ACE manager object.
TargetManager Class	Manages targets in the Policy Accelerator.

Message Support Classes

Frequently, an application needs to pass configuration changes from the host to a Policy Accelerator or pass summary information back. The host API provides access to the asynchronous messaging system, in the form of `Downcall` and `UpcallHandler` objects, and the means to coordinate crosscalls between ACEs, in the form of crosscall manager objects. It also provides the `Message` and `MessageBlock` classes for constructing messages for downcalls from the host to the Policy Accelerator.

Because the messaging system is asynchronous, you supply a callback function in each message handler object, which is executed when the message is received.

The following host classes support message passing between the host and Policy Accelerators, or between ACEs:

Class	Description
<code>CrosscallManager</code> Class	Manages crosscall objects, which ACEs on the Policy Accelerator use to send calls to each other.
<code>CrosscallHandlerManager</code> Class	Manages crosscall handlers, which ACEs on the Policy Accelerator use to receive calls from each other.
<code>Downcall</code> Class	Sends messages from the host module to the accelerator module.
<code>Message</code> Class	Encapsulates data to send from the host module to the Policy Accelerator module using downcalls.
<code>MessageBlock</code> Class	Encapsulates data buffers for messages.
<code>UpcallHandler</code> Class	Receives messages sent in upcalls from the accelerator module to the host module.

Base Class

All of the classes except `Message` and `MessageBlock` are derived from a base class, which provides basic functionality. You do not normally use this class directly or create your own subclasses of it.

Class	Description
<code>NBObject</code> Class	Most of the classes in the host API are derived from this base class, and inherit basic methods.

Error Handling in the Host API

When host API functions are successful, they either return a reference to a newly created object (in the case of constructors), or return the constant value `NBSuccess`. On failure:

- Constructors throw an exception of type `NBError`.
- Other functions return `NULL` or an error code of type `NBError`.

The predefined error codes, defined in `NBError.h`, are listed and described in Appendix A, “IX-API SDK Host API Error Codes.”

Host API Reference

This section lists and describes all of the host API classes in alphabetical order. Within each class, the constructor for that class is listed first, followed by the remaining methods in alphabetical order.

Include Files To use these classes, include the following header file in your code:

```
#include "nbapi\nbappl.h"
```

API Classes The host API contains the following classes:

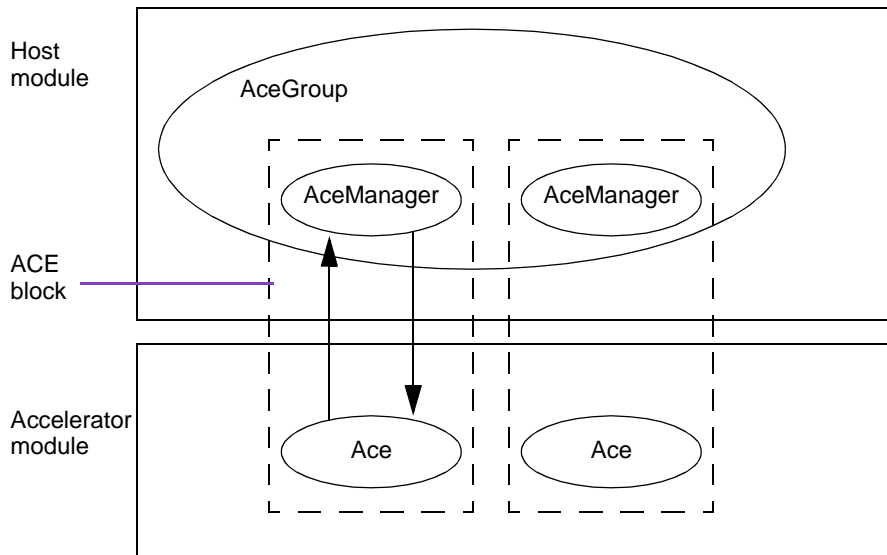
Class	Description
AceGroup Class (page 33)	Holds one or more ACE managers on the host. Every application must contain at least one ACE group object.
AceManager Class (page 37)	Manages and controls an ACE. Every application must contain at least one ACE manager object.
CrosscallHandlerManager Class (page 48)	Manages crosscall handlers that ACEs on the Policy Accelerator use to receive calls from one another.
CrosscallManager Class (page 52)	Manages crosscall objects that ACEs on the Policy Accelerator use to send calls to one another.
Downcall Class (page 56)	Sends messages to the accelerator module from the host module.
Message Class (page 62)	Encapsulates data to pass from the host module to the Policy Accelerator module using downcalls.
MessageBlock Class (page 66)	Encapsulates data in buffers to be used in messages.
NBApp1 Class (page 68)	Provides services to manage the setup of ACEs. Every application must contain exactly one application object. This is the main object of a policy enforcement application.
NBError Class (page 81)	Provides access to error codes from object constructors in the host module.
NBObject Class (page 83)	Provides basic methods. Most of the classes in the host API are derived from this base class and inherit basic methods.

Class	Description
TargetManager Class (page 85)	Manages targets in the Policy Accelerator.
UpcallHandler Class (page 88)	Receives messages sent in upcalls from the accelerator module to the host module.

AceGroup Class

Use the `AceGroup` class to associate ACEs with one another. ACE groups are containers that hold one or more ACE managers. Each ACE manager must be associated with an ACE group. Every application must have at least one ACE group object. You normally create a subclass of this base class for your application.

`AceManager` objects on the host are paired with `Ace` objects on the Policy Accelerator. The pair is known as an ACE block. The following figure shows how an ACE group can define a collection of ACE blocks by grouping the ACE managers.



You can use ACE groups to associate ACE managers (and thereby ACE blocks) according to functionality. That is, you can spread the implementation of a particular function over several ACEs, and use an ACE group to make them a functional entity. You can also group ACEs for resource management; for example, to keep track of which ACEs reside on which Policy Accelerators.

The `AceGroup` class contains the following method:

Method	Description
<code>AceGroup</code> Constructor	Instantiates an ACE group.

Class Derivation

The `AceGroup` class is derived from the `NBObject` class, inheriting all its public methods.

Example

This example is from the `basicApp` demo application. The subclass `NBBasicAceGroup` contains the handle for the ACE (`AceManager`) contained by this group.

```
class NBBasicAceGroup: public AceGroup {
public:
    NBBasicAceGroup(NBAppI* appl, NBFactory* nbFactory,
                    char* name);
    ~NBBasicAceGroup();
    NBBasicAce* basicAce;
};
```

The constructor instantiates the ACE manager contained by the group, and the destructor deletes the ACE manager object. Anything created with the constructor must be explicitly deleted with the destructor as shown here.

```
NBBasicAceGroup::NBBasicAceGroup(NBAppI* appl,
                                   NBFactory* nbFactory,
                                   char* name):
    AceGroup(appl, NULL, name) {
    // Create the only ACE needed for this group
    try {
        basicAce = new NBBasicAce(appl, this, "NBBasicAce");
    }
    catch (NError E) {
        throw NError(NB_ERROR(NBERROR_TESTAPP_ERRACE));
    }
}

NBBasicAceGroup::~NBBasicAceGroup()
{
    delete basicAce;
}
```

See Also

- [AceManager Class](#)
- “The Object Framework” on page 49 of *Developing Applications Using the IX-API SDK*

AceGroup Constructor

Creates an `AceGroup` object.

```
AceGroup (NBAppI* argAppI,
          NBFactory* argNBFactory,
          char * argName) throws NError;

AceGroup (NBAppI* argAppI,
          NBFactory* argNBFactory,
          char * argName
          NBStringList* list = NULL) throws NError;
```

Argument	Description
<code>argAppI</code>	The object name of the application object for this application.
<code>argFactory</code>	Not used. Pass the value <code>NULL</code> .
<code>argName</code>	The dictionary name of this ACE group. Must be unique within the application.
<code>list</code>	Not used. Pass the value <code>NULL</code> .

Returns

When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
<code>NBERROR_NBOBJ_NULLNAME</code>	The name of the ACE group (<code>argName</code>) is <code>NULL</code> .
<code>NBERROR_NBOBJ_NAMETOOLONG</code>	The name of the ACE group (<code>argName</code>) is too long. The maximum length of the object name is <code>OBJNAME_MAXLEN</code> , which is defined in <code>NBapi\nbparam.h</code> .
<code>NBERROR_NBOBJ_OUTOFMEMORY</code>	Cannot allocate memory to create the ACE group object.

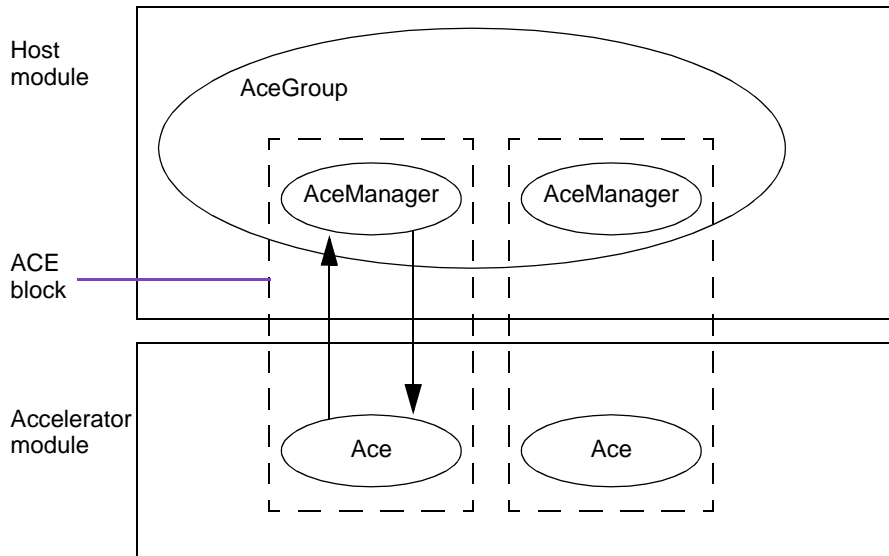
Return Codes	Description
NBERROR_NBOBJ_NULLAPPL	The name of the application (<code>argAppl</code>) is NULL.
NBERROR_NBOBJ_CANNOTREGISTER	Cannot register this object with the Resolver for one of the following reasons: <ul style="list-style-type: none">■ The Resolver is not running.■ The Policy Accelerator driver is not running.■ There is a problem communicating with the Policy Accelerator.■ The name has already been registered.

Description

ACE groups are containers that hold one or more ACE managers. ACE managers are objects on the host that communicate with ACEs on the Policy Accelerator. When you create an ACE manager, you specify the ACE group object to which it belongs.

AceManager Class

Use this class to create ACE managers, which represent the host side of an ACE block. Each ACE manager object has a corresponding ACE object on the Policy Accelerator. The paired objects are associated by having the same dictionary name. Every ACE manager must be part of an ACE group, and every application must have at least one ACE block.



The `load` method of the ACE manager initializes the ACE on the Policy Accelerator, identifying and loading the NCL rules and action code files for the accelerator module.

The ACE manager can get and set the state of an ACE. The state of an ACE includes:

- Its dictionary name
- The ACE group object to which it belongs
- Its bindings (pass and drop targets)
- Its links

You also use ACE managers to communicate with ACEs on the Policy Accelerator using upcalls and downcalls.

You create subclasses of this base class in which to define the methods that are used by your application. Typically, these are upcall handler callbacks and methods to send downcalls. For more information on upcall handlers and downcalls see “UpcallHandler Class” on page 88 and “Downcall Class” on page 56.

The `AceManager` class contains the following methods:

Method	Description
<code>AceManager</code> Constructor	Instantiates the class.
<code>getCompilerErrorMessages</code> Method	Retrieves error messages generated by the NCL compiler when loading an ACE.
<code>getDropTarget</code> Method	Identifies the drop target manager that is associated with the ACE manager.
<code>getPassTarget</code> Method	Identifies the pass target manager that is associated with the ACE manager.
<code>getTag</code> Method	Identifies the tag associated with the ACE.
<code>load</code> Method	Loads the NCL rules file containing the rules and the compiled action file containing the actions for the ACE manager.
<code>releaseCompilerErrorMessages</code> Method	Releases memory allocated by the <code>getCompilerErrorMessages</code> method.
<code>releaseMessage</code> Method	Release memory allocated by the system for a message sent in an upcall.

Class Derivation

The `AceManager` class is derived from the `NBObject` class, inheriting all its public methods.

Example

Use the `AceManager` class as a base to derive a subclass that represents the ACE or ACEs used by your application. This example is from the `basicApp` demo application.

Define the subclass, which in this case contains a handle for an upcall handler object and its callback method:

```
class NBBasicAce: public AceManager {
public:
    NBBasicAce(NBAppl* appl, AceGroup* acegroup, char* name);
    ~NBBasicAce();

    void peekPacketUpcall(Message* m);
};
```

```

UpcallHandler* peekPacketUpcallHandle;

ULONG getUpcallId(void) {
    return (ULONG)peekPacketUpcallHandle->getId();
}
};

```

Define the constructor and methods for the subclass. This one creates the upcall handler object:

```

NBBasicAce::NBBasicAce (NBAppI* appl, AceGroup* acegroup,
                        char* name):
    AceManager(appl, acegroup, name)
{
    // create upcall object
    try {
        peekPacketUpcallHandle = new UpcallHandler
                                (appl, acegroup, this,
                                 "peekPacketUpcall",
                                 (UpcallFp)peekPacketUpcall);
    }
    catch (NError E) {
        throw NError(NB_ERROR(NBERROR_TESTAPP_ERRUPCALLACEONE));
    }
    // init accelerator side of ACE: load rules, actions
    if (load ("basicAppRules", "basicAppActions")
        != NB_SUCCESS) {
        throw NError (NB_ERROR (NBERROR_TESTAPP_CANNOTLOADACEONE));
    }
}

```

Define the destructor to clean up any structures you have created:

```

NBBasicAce::~NBBasicAce ()
{
    delete peekPacketUpcallHandle;
}

```

Define the upcall handler callback as a method in the ACE manager subclass:

```

void NBBasicAce::peekPacketUpcall (Message* m)
{
    NB_ASSERT (m->getLen1 () == sizeof (nuint32));
    printf ("NoOfPackets: %05d\n",
            ntohl (* (nuint32 *) m->getBuffer1 ()));
    releaseMessage (m);
}

```

AceManager Constructor

Creates an `AceManager` object.

```
AceManager (NBAppl* argAppl,
            AceGroup* argAceGroup,
            char * argName,
            DWORD argAceMode,
            char * argPEName = NULL) throws NError;
```

Argument	Description
argAppl	The object name of the application object for this application.
argAceGroup	The object name of the ACE group containing the ACE manager.
argName	The dictionary name of this ACE manager, which must be the same as the dictionary name of the corresponding ACE object in the accelerator module. Must be unique among ACE managers in this ACE group.
argAceMode	<p>A set of one-bit flags that define the behavior of an ACE block with respect to whether it can modify packets and which hardware resources are used to execute it.</p> <p>Use a logical OR of the mode constants to combine them.</p> <p>Specify one of the following modification modes:</p> <ul style="list-style-type: none">■ ACE_WRITER (default)■ ACE_READER <p>Specify one of the following placement modes:</p> <ul style="list-style-type: none">■ ACE_PLMODE_ADVISORY (default)■ ACE_PLMODE_MANDATORY <p>Specify the following flag if the ACE will perform string searches:</p> <ul style="list-style-type: none">■ ACE_STRINGSEARCH
argPEName	The name of the Policy Accelerator where the ACE is to be executed.

Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NBError` with one of the following error codes, which you can access using the `NBError` method `GetErrorcode`.

Return Codes	Description
<code>NBERROR_NBOBJ_NULLNAME</code>	The name of the ACE manager (<code>argName</code>) is NULL.
<code>NBERROR_NBOBJ_NAME_TOO_LONG</code>	The name of the ACE manager (<code>argName</code>) is too long. The maximum length of the object name is <code>OBJNAME_MAXLEN</code> , which is defined in <code>NBapi\nbparam.h</code> .
<code>NBERROR_NBOBJ_OUTOFMEMORY</code>	Cannot allocate memory to create the ACE manager object.
<code>NBERROR_ACEMGR_NULLAPPL</code>	The name of the application (<code>argAppl</code>) that owns this ACE manager is NULL.
<code>NBERROR_ACEMGR_NULLACEGROUP</code>	The name of the ACE group (<code>ArgAceGroup</code>) is NULL.
<code>NBERROR_ACEMGR_INVACEMODE</code>	Invalid ACE mode specified.
<code>NBERROR_ACEMGR_CANNOTREGISTER</code>	Cannot register this ACE manager with the Resolver for one of the following reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
<code>NBERROR_ACEMGR_CANNOTDEVREG</code>	Cannot register this ACE manager with the IX-API SDK kernel device driver for any of the following reasons: <ul style="list-style-type: none"> ■ The driver is not running. ■ The name has already been registered. ■ There is a communication problem with the Policy Accelerator.
<code>NBERROR_ACEMGR_CANNOTCREATEPASS</code>	Cannot create default pass target.
<code>NBERROR_ACEMGR_CANNOTCREATEDROP</code>	Cannot create default drop target.

Description Use the `AceManager` constructor to create an ACE manager object. ACE managers initialize the ACE, get the ACE state, and manage communication between ACEs. You typically create a subclass and customize the constructor to create the communication objects and methods.

You specify the ACE's modification and placement mode in the constructor. These flags define the behavior of the ACE block with respect to whether it can modify packets and which hardware resources are used to execute it.

You must specify one modification flag, one placement flag, and, optionally, a string search flag. Use a logical OR to combine the flags.

- Specify one of the following modification modes:
 - `ACE_WRITER`: The ACE can modify the packet that it is processing. Therefore, the system cannot speculatively execute code belonging to ACEs downstream in the data flow chain.
 - `ACE_READER`: The ACE can never modify the packet that it is processing. Therefore, the system can speculatively execute code belonging to other ACEs downstream in the data flow chain.
- Specify one of the following placement modes:
 - `ACE_PLMODE_ADVISORY`: If `argPEName` is a valid Policy Accelerator name, the system places the ACE on the named Policy Accelerator, if it is not already fully allocated. If the named Policy Accelerator is fully allocated, the system chooses another hardware resource on which to place the ACE. If `argPEName` is `NULL`, the system chooses where to execute the ACE.
 - `ACE_PLMODE_MANDATORY`: If `argPEName` is a valid Policy Accelerator name, the system places the ACE on the named Policy Accelerator, if it is not already fully allocated or not available for some other reason. If the named Policy Accelerator is not available, the constructor throws an error and the ACE is not constructed. If `argPEName` is `NULL`, the constructor throws an error.
- Specify the `ACE_STRINGSEARCH` flag if the ACE will perform string searches in packet buffers. For more information on string searches, see “String Search Classes” in Chapter 4, “Action Services Library.”

To specify the Policy Accelerator on which the ACE is to run, use a Policy Accelerator name of the following form:

`nbhwpe n`

The value of n indicates the order in which the Policy Accelerator was installed in the system. The first Policy Accelerator installed is number 0, the next is 1, and so on; for example, `nbhwpe0` and `nbhwpe1`. For more information on Policy Accelerator naming, see Appendix C, “Policy Accelerator Name Space.”

See Also

“AceGroup Class” on page 33

getCompilerErrorMessages Method

Retrieves error messages generated by the NCL compiler when loading an ACE.

```
DWORD getCompilerErrorMessages (char *& errorBuffer);
```

Argument	Description
errorBuffer	A pointer to the error buffer, modified by the method.

Returns When successful, NB_SUCCESS. When not successful, returns one of the following codes:

Return Codes	Description
NBERROR_ACEMGR_CANNOTRECOVERMSG	Cannot retrieve NCL compiler error messages because communication with the Resolver was interrupted during this operation.
NBERROR_ACEMGR_INVALIDCMPLRERRMSG	No error messages are currently available.

Description The retrieved error messages are stored at the location pointed to by the errorBuffer argument. The library allocates memory for this buffer. When it is no longer needed, you are responsible for freeing it using the releaseCompilerErrorMessages method.

See Also releaseCompilerErrorMessages Method

getDropTarget Method

Identifies the drop target manager associated with this ACE manager.

```
TargetManager * getDropTarget (void);
```

Returns A pointer to the drop target.

Description Use this method to identify the `TargetManager` object associated with the drop target of the ACE. If you do not create and bind a drop target, one is created by default when you initialize the ACE.

All packets sent to the drop target are dropped. By default, all packets that the application does not explicitly dispose of are sent to the drop target.

You can develop your own targets, as described in “`TargetManager Class`” on page 85. You must bind a target to an ACE using the `bind` method in the `NBApp1` class. Packets passed to unbound targets are dropped.

See Also `getPassTarget Method`, `TargetManager Class`

getPassTarget Method

Identifies the pass target manager associated with this ACE manager.

```
TargetManager* getPassTarget (void);
```

Returns A pointer to the pass target.

Description Use this method to identify the `TargetManager` object associated with the pass target of the ACE. If you do not create and bind a pass target, one is created by default when you initialize the ACE.

All packets sent to the pass target are directed to the ACE bound to this target.

You can develop your own targets, as described in “`TargetManager Class`.” You must bind a target to an ACE using the `bind` method in the `NBApp1` class. Packets passed to unbound targets are dropped.

See Also `getDropTarget Method`, `TargetManager Class`

getTag Method

Retrieves the tag associated with the ACE.

```
uint16 getTag (void);
```

Returns The tag associated with the ACE.

Description Tags are used to identify the packets that are coming in through interfaces. A tag is a unique integer that identifies an ACE within the context of the Policy Accelerator. The Policy Accelerator generates the tag and uses it to set the `ifnum` field in the base protocol (`base.ifnum`).

See Also `NBAppl::getTag Method`, `Buffer::interfaceNum Method` in Chapter 4, “Action Services Library.”

Load Method

Loads the Policy Accelerator memory with the files containing the NCL rules and actions for the ACE block of which this ACE manager is a part.

```
DWORD load (char * rulesFilename,
            char * actionsFilename);
```

Argument	Description
<code>rulesFilename</code>	The name of the file containing the rules to load. Must be a valid filename with the extension <code>.ncl</code> (NCL classification rules file).
<code>actionsFilename</code>	The name of the compiled file containing the actions to load. Must be a valid filename with the extension <code>.nbo</code> (IX-API SDK object file).

Returns One of the following codes:

Return Codes	Description
<code>NB_SUCCESS</code>	The method succeeded.
<code>NBERROR_ACEMGR_NULLFILENAME</code>	One of the filename pointers is <code>NULL</code> .

Return Codes	Description
NBERROR_ACEMGR_FNTOOLONG	One of the filenames is too long, exceeding the maximum <code>MAX_FILENAME_LENGTH</code> , which is defined in <code>NBapi\nbparam.h</code> .
NBERROR_ACEMGR_OUTOFMEM	Cannot allocate memory to load code.
NBERROR_ACEMGR_CWDERR	Cannot access the current directory.
NBERROR_ACEMGR_CANNOTSENDFN	Cannot communicate with the Resolver to request that this code be loaded.
NBERROR_ACEMGR_CANNOTRECVFNACK	Communication with the Resolver was broken while this operation was in progress.

Description

Use the load method to do one of two things:

- To provide the NCL classification rules and action code for this ACE block when it is being initialized. Both filenames (`rulesFilename` and `actionsFilename`) specified must be valid.
- To load new NCL classification rules on the fly. To do this, specify a valid filename for the rules and `NULL` for the actions filename. When an application loads a new NCL rules file, the system compiles the new file and loads it into the hardware resource associated with this ACE block. All rules contained in the file are automatically enabled in the order in which they are defined.

When the application executes the ACE manager's `load` method, the host downloads the specified files to the Policy Accelerator, and the Policy Accelerator immediately calls the initialization function in the specified action code file. You define this function to construct the ACE object for the accelerator module. See "Initialization Function" on page 172 in Chapter 4, "Action Services Library."

The actions file must be precompiled using the `nbgcc` cross compiler. See "nbgcc Command" on page 419 in Chapter 7, "Command-Line Tools."

Example

```
load ("MyRules", "MyActions");
```

releaseCompilerErrorMessages Method

Releases memory allocated by `getCompilerErrorMessages`.

```
void releaseCompilerErrorMessages (void);
```

Returns Nothing.

See Also `getCompilerErrorMessages Method`

releaseMessage Method

Releases memory allocated for a message sent in an upcall.

```
static void releaseMessage (Message *pMessage);
```

Argument	Description
pMessage	A pointer to the message to be released.

Returns Nothing.

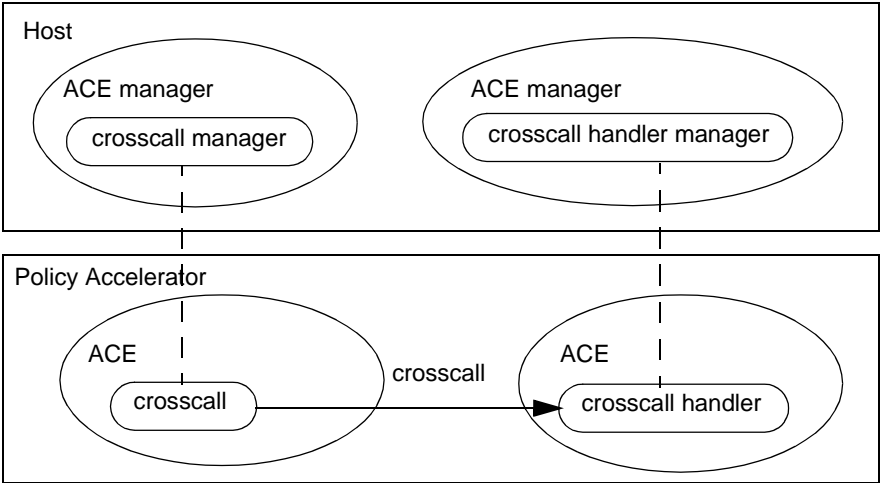
Description Use this method to release the memory allocated by the Policy Accelerator system for a message sent in an upcall from the Policy Accelerator. This is a static method that you can call from anywhere in the application using the scope specifier for the `AceManager` class.

You normally use this method in the callback for an upcall handler, to release the received message that was allocated by the system.

See Also `Message Class`, `UpcallHandler Class`

CrosscallHandlerManager Class

Use this class to define and create crosscall handler managers. Crosscall handler managers are host objects that mirror crosscall handler objects on the Policy Accelerator. The accelerator module object and its manager object are associated by having the same dictionary name. The manager object allows for management (such as linking and unlinking) of crosscall handlers.



Crosscall handlers reside on one ACE and accept messages from a crosscall in another ACE. For more information on crosscalls and crosscall handlers, see Chapter 4, “Action Services Library.”

The `CrosscallHandlerManager` class contains the following method:

Method	Description
<code>CrosscallHandlerManager</code> Constructor	Instantiates the class.

Class
Derivation

The `CrosscallHandlerManager` class is derived from the `NBObject` class, inheriting all its public methods.

Example

This example illustrates how to use crosscall handler managers. In this example, `AceOne` sends a crosscall and `AceTwo` receives it. Therefore `AceTwo` must contain a `CrosscallHandler` object in the accelerator module, and its manager in the host module. Here, the constructor for the `AceTwo` ACE manager creates the crosscall handler manager object.


```

NBACETwoMgr::NBACETwoMgr (NBAPpl* appl, AceGroup* AceGroup,
                          char * name):
    AceManager (appl, AceGroup, name)
{
    try {
        // create the Crosscall Handler Manager object
        crosscallHandlerMgr = new CrosscallHandlerManager (
                                appl, AceGroup, this,
                                "crosscallTestHandler");
    }
    catch (NBEError E) {
        throw;
    }
}

```

The ACE constructor in the accelerator module would create the corresponding crosscall handler object, using the same dictionary name, `crosscallTestHandler`.

Before you can send a message, you must link the crosscall in `AceOne` and its handler in `AceTwo` using the `link` method of the application object.

See Also

- “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*
- CrosscallHandler Class in Chapter 4, “Action Services Library.”
- NBAPpl::link Method

CrosscallHandlerManager Constructor

Creates a `CrosscallHandlerManager` object.

```
CrosscallHandlerManager (NBAppI* argAppl,
                        AceGroup * argAceGroup,
                        AceManager * argAceMgr,
                        char * argName) throws NError;
```

Argument	Description
argAppl	The object name of the application that created the ACE group specified in argAceGroup.
argAceGroup	The object name of the ACE group that contains the ACE manager specified in argAceMgr.
argAceMgr	The object name of the ACE manager that contains this crosscall handler manager.
argName	The dictionary name of this crosscall handler manager. This must be the same as the dictionary name of the associated <code>CrosscallHandler</code> object in the Policy Accelerator. Must be unique among objects in this ACE manager.

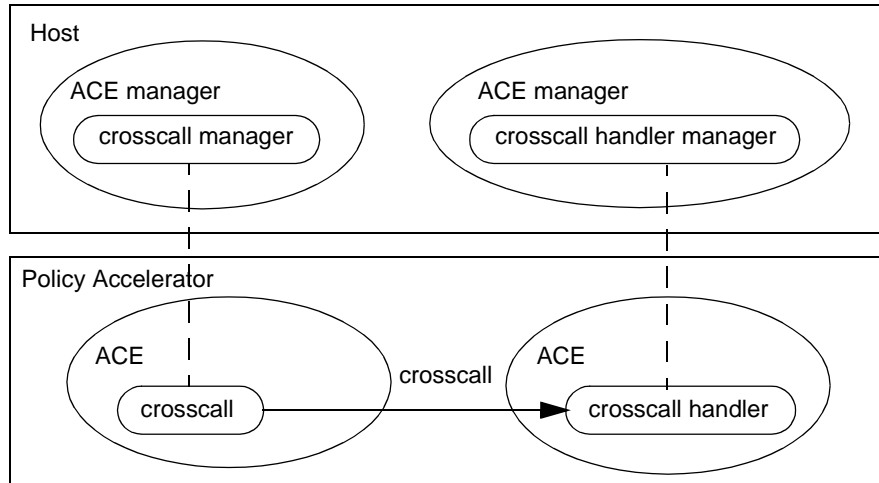
Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
NBERROR_NBOBJ_NULLNAME	The name of the crosscall handler manager (argName) is NULL.
NBERROR_NBOBJ_NAME_TOO_LONG	The name of the crosscall handler manager (argName) is too long. The maximum length of the object name is OBJNAME_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_NBOBJ_OUTOFMEMORY	Cannot allocate memory to create the crosscall handler manager.
NBERROR_CROSSCALLHANDLER_NULLAPPL	argAppl is NULL.
NBERROR_CROSSCALLHANDLER_NULLACEGROUP	argAceGroup is NULL.
NBERROR_CROSSCALLHANDLER_NULLACEMGR	argAceMgr is NULL.

Description	A crosscall handler manager manages the crosscall handler object on the Policy Accelerator, which enables one ACE to receive messages from another ACE.
See Also	CrosscallManager Class, ASL Crosscall Class, ASL CrosscallHandler Class

CrosscallManager Class

Use this class to define and create crosscall managers. Crosscall managers are host objects that mirror crosscall objects on the Policy Accelerator. The accelerator module object and its manager object are associated by having the same dictionary name. The manager object allows for management (such as linking and unlinking) of crosscalls.



Crosscalls allow one ACE to send messages to another ACE. For more information on crosscalls and crosscall handlers, see Chapter 4, “Action Services Library.”

The `CrosscallManager` class contains the following method:

Method	Description
<code>CrosscallManager</code> Constructor	Instantiates the class.

Class Derivation

The `CrosscallManager` class is derived from the `NBObject` class, inheriting all its public methods.

Example

This example illustrates how to use crosscall managers. In this example, `AceOne` sends a crosscall and `AceTwo` receives it. Therefore `AceOne` must contain a `Crosscall` object in the accelerator module, and its manager in the host module. Here, the constructor for the `AceOne` ACE manager creates the crosscall manager object.

```
class NBACEOneMgr : public AceManager {
public:
    NBACEOneMgr (NBAPpl* appl, AceGroup* AceGroup, char * name);
    ~NBACEOneMgr ();
    CrosscallManager* crosscallMgr;
};

NBACEOneMgr::NBACEOneMgr (NBAPpl* appl, AceGroup* AceGroup,
                          char * name):
    AceManager (appl, AceGroup, name)
{
    try {
        // Create CrosscallManager object
        crosscallMgr = new CrosscallManager (appl, AceGroup, this,
                                              "crosscallTest");
    }
    catch (NBEError E) {
        throw;
    }
}
```

The ACE constructor in the accelerator module would create the corresponding crosscall object, using the same dictionary name, `crosscallTest`. Before you can send a message, you must link the crosscall in `AceOne` and its handler in `AceTwo` using the `link` method of the application object.

See Also

- “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*
- Crosscall Class in Chapter 4, “Action Services Library.”
- `NBAPpl::link` Method

CrosscallManager Constructor

Creates a `CrosscallManager` object.

```
CrosscallManager (NBAPpl* argAppl,
                  AceGroup * argAceGroup,
                  AceManager * argAceMgr,
                  char * argName) throws NError;
```

Argument	Description
<code>argAppl</code>	The object name of the application that created the ACE group specified in <code>argAceGroup</code> .
<code>argAceGroup</code>	The object name of the ACE group that contains the ACE manager specified in <code>argAceMgr</code> .
<code>argAceMgr</code>	The object name of the ACE manager that contains this crosscall manager.
<code>argName</code>	The dictionary name of this crosscall manager. This must be the same as the name of the associated <code>Crosscall</code> object in the Policy Accelerator. Must be unique among objects in this ACE manager.

Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
<code>NBERROR_NBOBJ_NULLNAME</code>	The name of the crosscall manager (<code>argName</code>) is NULL.
<code>NBERROR_NBOBJ_NAMETOOLONG</code>	The name of the crosscall manager (<code>argName</code>) is too long. The maximum length of the object name is <code>OBJNAME_MAXLEN</code> , which is defined in <code>NBapi\nbparam.h</code> .
<code>NBERROR_NBOBJ_OUTOFMEMORY</code>	Cannot allocate memory to create the crosscall manager.
<code>NBERROR_NBOBJ_NULLAPPL</code>	The <code>argAppl</code> argument is NULL.
<code>NBERROR_NBOBJ_NULLACEGROUP</code>	The <code>argAceGroup</code> argument is NULL.
<code>NBERROR_UPCALL_NULLACEMGR</code>	The <code>argAceMgr</code> argument is NULL.

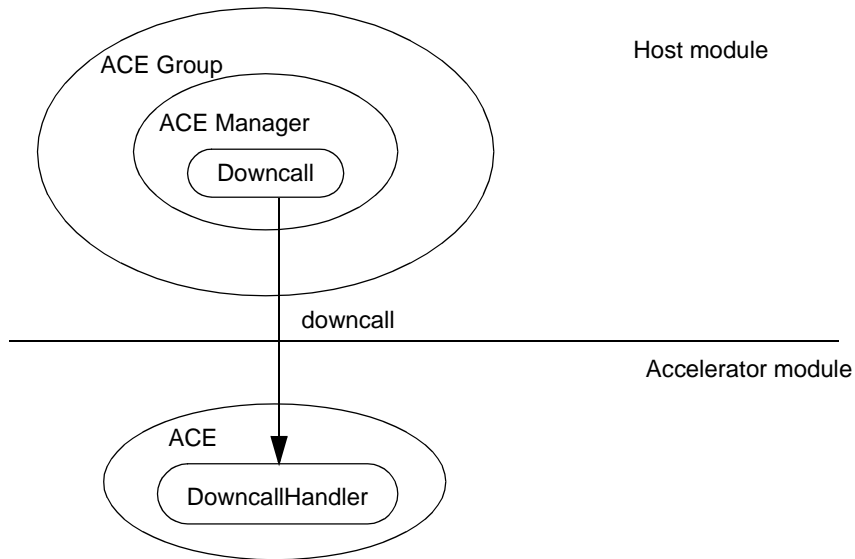
Description A crosscall manager manages a crosscall object on a Policy Accelerator, which enables one ACE to issue calls to another ACE.

See Also CrosscallHandlerManager Class, ASL Crosscall Class, ASL Cross-callHandler Class

Downcall Class

Use this class to create downcalls to send messages to the Policy Accelerator.

A downcall sends a message from the host application to the application running on the Policy Accelerator. A downcall object resides on the host and requires a corresponding downcall handler object on the Policy Accelerator. The host module and accelerator module objects are associated by having the same dictionary name. For more information on downcall handlers, see Chapter 4, “Action Services Library.”



Typically, you create a subclass of `AceManager` that contains a method to create a message and send it in a downcall, using the `call` method. The constructor for the ACE manager would also create the `Downcall` object.

Use downcalls to send small amounts of data, such as state notifications and counters. To make applications run faster, do most of your packet processing on the Policy Accelerator. If you need to transfer large amounts of data between the Policy Accelerator and the host, do so using the stack. See “Moving Packets between the Policy Accelerator and the Host” on page 114 in *Developing Applications Using the IX-API SDK*.

The Downcall class contains the following methods:

Method	Description
Downcall Constructor	Instantiates the class.
call Method	Sends the specified message from the host to the ACE on the Policy Accelerator.

Class Derivation

The Downcall class is derived from the NBOBJECT class, inheriting all its public methods.

Example

The following example of downcall usage is from the TwoAceApp demo application. The definition of the ACE manager subclass declares a downcall object and a method that creates and sends a message using the downcall:

```
class NBACEOneMgr : public AceManager {
public:
    NBACEOneMgr (NBAPPL* appl, AceGroup* aceGroup, char* name);
    ~NBACEOneMgr ();
    // Downcall
    void setReportPeriod (int reportPeriod);
    Downcall* setReportPeriodDowncallHandle;
};
```

The constructor and destructor for the ACE manager subclass create and delete the downcall object:

```
NBACEOneMgr::NBACEOneMgr (NBAPPL* appl, AceGroup* acegroup,
                           char* name):
    AceManager (appl, acegroup, name)
{
    // create downcall object
    try {
        setReportPeriodDowncallHandle =
            new Downcall (appl, acegroup, this,
                          "setReportPeriodDowncall");
    }
    catch (NError E) {
        throw;
    }
    // init accelerator side of ACE: load rules, actions
    if (load ("aceOneRules", "aceOneActions") != NB_SUCCESS)
    {
        throw NError (NB_ERROR (NBERROR_TWOACEAPP));
    }
}
```

```
NBAceOneMgr::~~NBAceOneMgr ()
{
    delete setReportPeriodDowncallHandle;
}
```

The method that creates and sends the downcall message is defined as follows:

```
void
NBAceOneMgr::setReportPeriod (int reportPeriod)
{
    Message* m = new Message ( (char*)&reportPeriod,
                               sizeof (int), NULL, 0);
    setReportPeriodDowncallHandle->call (m);
    delete m;
}
```

The ACE constructor in the accelerator module would create a downcall handler object to receive the message, using the same dictionary name, `setReportPeriodDowncall`.

See Also

- “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*
- DowncallHandler Class in Chapter 4, “Action Services Library.”

Downcall Constructor

Creates a Downcall object.

```
Downcall (NBAppl* argAppl,
          AceGroup * argAceGroup,
          AceManager * argAceMgr,
          char * argName) throws NError;
```

Argument	Description
argAppl	The object name of the application that created the ACE group specified in argAceGroup.
argAceGroup	The object name of the ACE group that contains the ACE manager specified in argAceMgr.
argAceMgr	The object name of the ACE manager to which a message is being sent.
argName	The dictionary name of this downcall. This must be the same as the dictionary name of the corresponding downcall handler object in the accelerator module. Must be unique among objects in this ACE manager.

Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
NBERROR_NBOBJ_NULLNAME	The name of this downcall (argName) is NULL.
NBERROR_NBOBJ_NAMETOOLONG	The name of this downcall (argName) is too long. The maximum length of the object name is OBJNAME_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_NBOBJ_OUTOFMEMORY	Cannot allocate memory to create the downcall.
NBERROR_DOWNCALL_NULLAPPL	The argAppl argument is NULL.
NBERROR_DOWNCALL_NULLACEGROUP	The argAceGroup argument is NULL.

Return Codes	Description
NBERROR_DOWNCALL_NULLACEMGR	The <code>argAceMgr</code> argument is NULL.
NBERROR_DOWNCALL_CANNOTREGISTER	Cannot register this downcall with the Resolver for one of the following reasons: <ul style="list-style-type: none">■ The Resolver is not running.■ The Policy Accelerator driver is not running.■ There is a problem communicating with the Policy Accelerator.■ The name has already been registered.

Description Downcalls enable the host module to asynchronously send messages to the Policy Accelerator module.

You typically create the downcall object as part of constructing the ACE manager object. The ACE manager object should also contain a method that constructs the message and sends it using this object's `call` method.

See Also [Message Class](#), [MessageBlock Class](#), [ASL DowncallHandler Class](#)

call Method

Send a message from the host to the Policy Accelerator.

```
DWORD call (Message * m);
```

Argument	Description
m	Contains the data being passed to the accelerator module.

Returns When successful, NB_SUCCESS. When not successful, one of the following codes:

Return Codes	Description
NBERROR_DOWNCALL_CANNOTOBTAINPECONTEXT	Cannot find downcall handler in the accelerator module. Every downcall must have a corresponding downcall handler. For more information on downcall handlers, see Chapter 4, “Action Services Library.”
NBERROR_DOWNCALL_CANNOTSENDDOWNCALL	The kernel driver refused to send this downcall.

Description Use this method to send the specified message from the host to the ACE on the Policy Accelerator. You are responsible for deleting the Message object after sending the call.

You typically call this method as part of a method in the ACE manager object, which constructs the message, sends it, and then deletes it.

Example

```
setReportPeriodDowncallHandle->call (m);  
delete m;
```

See Also Message Class, MessageBlock Class, ASL DowncallHandler Class

Message Class

Use the `Message` class in the host module to create messages to send in downcalls. To create messages to send in upcalls or crosscalls, see the ASL's `Message` Class in Chapter 4, "Action Services Library."

A `Message` object encapsulates the entire communication being transferred during a downcall within two buffers. You can specify the data buffers directly when you create the `Message` object, or you can encapsulate the data separately in up to two `MessageBlock` objects. For more information on message blocks, see "MessageBlock Class" on page 66.

Delete the `Message` object using the `delete` operator after the call has been successfully sent. There is a delay between the time the call is sent and the time it is completed.

This class is not derived from any other class. It contains the following methods:

Method	Description
Message Constructor	Instantiates the class.
<code>getBuffer1</code> Method	Gets the first buffer from which a message was made.
<code>getBuffer2</code> Method	Gets the second buffer from which a message was made.
<code>getLen1</code> Method	Gets the length of the first buffer.
<code>getLen2</code> Method	Gets the length of the second buffer.

Example

This example illustrates how to use the `Message` class with a downcall. It constructs a message with data to be passed (in this case, the period to report some statistics), and then sends the downcall. Finally, it frees the `Message` object using the `delete` operator.

```
NBAceOneMgr::setReportPeriod (int reportPeriod) {
    Message* m = new Message ((char *)&reportPeriod,
                               sizeof (int), NULL, 0);
    setReportPeriodDowncallHandle->call (m);
    delete m;
}
```

See Also

"Creating Messages and Message Blocks" on page 108 of *Developing Applications Using the IX-API SDK*

Message Constructor

Create a `Message` object from one or two message blocks or buffers.

```
Message (char * argBuffer1,
        DWORD argLen1,
        char * argBuffer2 = NULL,
        DWORD argLen2 = 0) throws NError;
```

```
Message (MessageBlock & b1,
        MessageBlock & b2) throws NError;
```

```
Message (MessageBlock & b1) throws NError;
```

Argument	Description
<code>argBuffer1</code>	A pointer to the first buffer.
<code>argLen1</code>	The number of bytes in the first buffer.
<code>argBuffer2</code>	A pointer to the second buffer. Optional.
<code>argLen2</code>	The number of bytes in the second buffer. Optional.
<code>b1</code>	A <code>MessageBlock</code> object for the first buffer of a message.
<code>b2</code>	A <code>MessageBlock</code> object for the second buffer of a message.

Returns

When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with the following error code, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
<code>NBERROR_NBOBJ_OUTOFMEMORY</code>	Cannot allocate memory to create this message.

Description

The three forms of the constructor allow you to specify the data buffers directly, or specify `MessageBlock` objects that encapsulate the data.

- The first form directly specifies the buffers that contain the message data. Because you specify the length, the buffers do not need to be `NULL`-terminated. If you specify only one of the buffers, the second buffer is empty.
- The second and third forms specify `MessageBlock` objects that encapsulate the message data. If you specify only one `MessageBlock`, the second buffer is empty.

The maximum size for message data is 3968 bytes; that is, one page (4096 bytes) minus some overhead (128 bytes) for metadata. When there are two blocks or buffers, the maximum is for the total size of both.

See Also `MessageBlock` Class

getBuffer1 Method

Retrieves the address of the first buffer of a message.

```
char * getBuffer1 (void);
```

Returns A pointer to the first buffer of a message, or to the data encapsulated by the first `MessageBlock`.

getBuffer2 Method

Retrieves the address of the second buffer of a message.

```
char * getBuffer2 (void);
```

Returns A pointer to the second buffer of a message, or to the data encapsulated by the second `MessageBlock`.

getLen1 Method

Retrieves the size of the first buffer of a message.

```
DWORD getLen1 (void);
```

Returns The length in bytes of the first buffer of a message, or of the data encapsulated by the first `MessageBlock`.

getLen2 Method

Retrieves the size of the second buffer of a message.

```
DWORD getLen2 (void);
```

Returns The length in bytes of the second buffer of a message, or of the data encapsulated by the second `MessageBlock`.

MessageBlock Class

Use this class to define and create message blocks. A message block is the building block of `Message` objects and is simply a pointer to a block of memory (buffer) and its length.

The maximum size for message data is 3968 bytes; that is, one page (4096 bytes) minus some overhead (128 bytes) for metadata. When you use two blocks to construct a message, the maximum is for the total size of both blocks.

You can create two kinds of message buffers:

- **NULL-terminated buffers**
Create NULL-terminated message buffers to hold any amount of data up to the first NULL byte; for example, when passing text in a message. To create a message from a NULL-terminated buffer, omit the buffer-length argument when creating the `MessageBlock` object.
- **Fixed-length buffers**
To create a message from a fixed-length buffer, specify the number of bytes for the buffer when creating the `MessageBlock` object. Use this form, for example, to pass a list of NULL-terminated strings.

This class is not derived from any other class. It contains the following method:

Method	Description
<code>MessageBlock</code> Constructor	Instantiates the class.

See Also “Creating Messages and Message Blocks” on page 108 of *Developing Applications Using the IX-API SDK*

MessageBlock Constructor

Creates a MessageBlock object.

```
MessageBlock (char * argBuffer);

MessageBlock (char * argBuffer,
              DWORD argLen);
```

Argument	Description
argBuffer	A pointer to the buffer.
argLen	The number of bytes in the buffer.

Returns

A reference to the newly created object.

Description

The first constructor creates a message block from a `NULL`-terminated buffer. The second constructor creates a message block from a buffer of a fixed length that is not terminated by `NULL`.

Example

The following example creates a message block from a string terminated by `NULL`:

```
char test [] = {"This is a test"};
MessageBlock (test); //NULL termination
```

The following example creates a message block from a buffer that can hold 30 bytes of data:

```
char test [3] [10] = {"This", "is", "test"}; //list of strings
MessageBlock (test, 30); //specify a length to copy all data
```

See Also

Message Class

NBApp1 Class

Use the NBApp1 class to create the main application object for the IX-API SDK host application. Use the methods in this class to bind targets to and unbind targets from ACEs, and to link and unlink crosscalls.

The NBApp1 class represents the Policy Accelerator portion of an application. For each IX-API SDK application that you create, you create a subclass of the NBApp1 class and an object of that subclass. Every application must contain exactly one application object.

NBApp1 subclasses contain ACE group objects. When you create a subclass, define the constructor for your subclass to create the ACEGroup objects.

The NBApp1 class contains the following methods:

Method	Description
NBApp1 Constructor	Instantiates the class.
bind Method	Binds a target to an ACE or to an interface represented by an ACE.
getTag Method	Identifies the tag value associated with a binding.
getStackDriverName Method	Retrieves the driver name for an interface.
link Method	Connects a crosscall in one ACE to a crosscall handler in another ACE.
unbind Method	Unbinds a target from an ACE.
unlink Method	Disconnects a crosscall from its crosscall handler.

Class Derivation

The NBApp1 class is derived from the NBOBJECT class, inheriting all its public methods.

Example

The following example is from the BasicApp demo application. In this example, the constructor for the application object creates the ACE group and the ACE manager objects, and uses its own bind method to set the bindings:

```
BasicApp::BasicApp (void):
    NBApp1 ("BasicApp", NULL, NULL)
{
    // Create ACE group
```

```

try {
    aceGroup = new AceGroup (this, NULL, "BasicAceGroup");
}
catch (NError E) {
    throw NError (NB_ERROR(NBERROR_BASICAPP_CANNOTCREATEGROUP));
}
// Create ACE manager
try {
    BasicAceManager =
        new BasicAceManager (this, aceGroup, "BasicAce");
}
catch (NError E) {
    throw NError (NB_ERROR(NBERROR_BASICAPP_CANNOTCREATEACE));
}
// Create bindings
NB_TRACE ("BINDING\n");
// incoming packets on the accelerator's
// FROM interface go to the ACE
unsigned long rval =
    bind("/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass",
        "/basicAppl/basicAceGroup/basicAce");
if (rval != NB_SUCCESS)
{
    NB_ABORT(rval);
}
// the ACE passes packets back to the accelerator's
// TO interface
rval = bind("/basicAppl/basicAceGroup/basicAce/pass",
    "/nbhwpe0/ToInterface:nbhwpe0B/Interface");
if (rval != NB_SUCCESS) {
    NB_ABORT(rval);
}
NB_TRACE ("BINDINGS ARE DONE\n");
}

```

See Also

- Chapter 3, “Elements of an Application,” in *Developing Applications Using the IX-API SDK*
- Chapter 5, “Controlling Packet Flow,” in *Developing Applications Using the IX-API SDK*

NBApp1 Constructor

Creates an NBApp1 object.

```
NBApp1 (char * argName,
        char * workingDirectory,
        char * cmdLine) throws NError;

NBApp1 (char * argName,
        char * cmdLine) throws NError;

NBApp1 (char * argName) throws NError;
```

Argument	Description
argName	The dictionary name of the new application object to create.
workingDirectory	Not used. Pass NULL.
cmdLine	Not used. Pass NULL.

Returns

When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Code	Description
NBERROR_NBOBJ_NULLNAME	The name of the application (argName) is NULL.
NBERROR_NBOBJ_NAMETOOLONG	The name of the application (argName) is too long. The maximum length of the object name is OBJNAME_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_NBOBJ_OUTOFMEMORY	Cannot allocate memory to create the application.
NBERROR_NBOBJ_CANNOTCREATEPIPE	The system cannot create a named pipe to communicate with the Resolver.
NBERROR_NBAPPL_ERRNBPIPE	The system cannot allocate memory to create an object used to manage the named pipe used to communicate with the Resolver.
NBERROR_NBAPPL_ERRNBPIPE	The system cannot allocate memory to create an object used to manage the pipe used by the Resolver to communicate with the application.

Return Code	Description
NBERROR_NBAPPL_CANNOTACCESSDEV	The application cannot connect with the Policy Accelerator kernel driver.
NBERROR_NBAPPL_ERRORDEVREG	The application cannot register with the Policy Accelerator kernel driver.
NBERROR_NBAPPL_CANNOTCREATEUPCALLTHREAD	The application cannot create a thread dedicated to handle upcalls from the accelerator module.
NBERROR_NBAPPL_CANNOTCREATESLVREQTHREAD	The application cannot create a thread dedicated to handle requests issued by the Resolver.
NBERROR_NBAPPL_CANNOTREGISTER	<p>Cannot register this application with the Resolver for one of the following reasons:</p> <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.

Description Use the `NBApp1` constructor to create the main object representing an IX-API SDK application. The `NBApp1` class provides services to manage the setup of ACEs. Applications use ACEs to process packets.

bind Method

Binds a target to an ACE or to an interface represented by an ACE.

```
DWORD bind (char * from,
            char * to);
```

Argument	Description
from	The input target to be bound to the ACE. Specify the path as <i>Appl\AceGroup\Ace\target</i> , using the dictionary names of the objects as described in Appendix C, “Policy Accelerator Name Space.”
to	The output ACE to be bound to the target. Specify the path as <i>Appl\AceGroup\Ace</i> as described in the Appendix above.

Returns When successful, NB_SUCCESS. When not successful, one of the following codes:

Return Codes	Description
NBERROR_NBAPPL_NULLNAMES	The pointer that should contain the object name is NULL. (That is, one of the arguments is a NULL pointer.)
NBERROR_NBAPPL_CANNOTSENDBINDREQ	The application cannot send a message to the Resolver to request a binding.
NBERROR_NBAPPL_CANNOTRECVBINDREQ	The application lost communication with the Resolver during execution of the bind request.
NBERROR_NBAPPL_ERRBINDREQ	Either the target or the destination ACE does not exist.

Description The bindings that you specify with this method determine how packets flow into and out of this ACE.

Targets represent possible sources and destinations for packets. Packets delivered to unbound targets are dropped. Every ACE has two system-defined targets named `pass` and `drop`. You do not normally bind the `drop` target. You can bind the `pass` target to another ACE that you have defined, or to a system-defined ACE.

To pass packets to and from the Policy Accelerator interfaces or host stack, use the system ACE and target names, as described in “System Names for Policy Accelerator Interfaces” on page 444.

If your site has customized the drivers for a standard network interface card (NIC) for communication with the Policy Accelerator using the ODX protocol, you can address the NIC connection directly as interface C. The syntax for using interface C is the same as that for the built-in interfaces A and B. For more information, see *Customizing a NIC Driver Using the ODX Protocol*.

Example

This example is part of the constructor for the application object of the BasicApp demo application. It creates bindings such that packets flow from the Policy Accelerator's A interface into the ACE, and out of the ACE to the Policy Accelerator's B interface.

```
// Create bindings
NB_TRACE ("BINDING\n");
// incoming packets on the Policy Accelerator's
// FROM interface A go to the ACE
unsigned long rval =
    bind("/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass",
        "/basicAppl/basicAceGroup/basicAce");
if (rval != NB_SUCCESS)
{
    NB_ABORT(rval);
}
// the ACE passes packets back to the Policy Accelerator's
// TO interface B
rval = bind("/basicAppl/basicAceGroup/basicAce/pass",
    "/nbhwpe0/ToInterface:nbhwpe0B/Interface");
if (rval != NB_SUCCESS) {
    NB_ABORT(rval);
}
NB_TRACE ("BINDINGS ARE DONE\n");
```

See Also

- [unbind Method](#)
- [Target Class in Chapter 4, “Action Services Library.”](#)
- [Chapter 5, “Controlling Packet Flow,” in *Developing Applications Using the IX-API SDK*](#)

getTag Method

Identifies the tag value associated with a binding.

```
uint16 getTag (char * argAceName);
```

Argument	Description
argAceName	The full path of the ACE associated with this tag, as described in Appendix C, "Policy Accelerator Name Space."

Returns The tag associated with the specified ACE block.

Description A tag is an integer that identifies an ACE binding within the context of the Policy Accelerator. The value of tags for system ACEs (such as the FROM and TO interfaces) varies from session to session.

The Policy Accelerator uses this tag to set the `ifnum` field in the `base` protocol during classification. NCL code can access the value in `base.ifnum` to find the tag associated with the ACE that represents the interface through which the packet came.

- See Also**
- `getTag` Method in `AceManager` Class
 - `Buffer::interfaceNum` Method in Chapter 4, "Action Services Library."
 - Appendix C, "Policy Accelerator Name Space."

Example When an ACE receives packets from different interfaces, it can use the tag to determine which interface the packet came through. NCL code can check `base.ifnum` and pass its value as an argument to an action, or action code can check this field directly. The following example code could be used in such an action function to retrieve the tags associated with ACEs for all interfaces and stacks in two boards.

```
OUTPUT:

TAGS PE0: FROMA=3, TOA=4, STACKFROMA=6, STACKTOA=5
TAGS PE0: FROMB=7, TOB=8, STACKFROMB=10, STACKTOB=9
TAGS PE1: FROMA=131, TOA=132, STACKFROMA=134, STACKTOA=133
TAGS PE1: FROMB=135, TOB=136, STACKFROMB=138, STACKTOB=137

CODE:
try {
    printf ("Getting tag for Interfaces & Stacks:\n");
    uint16 i1 = getTag
```

```

        ("\nbhwpe0\FromInterface:nbhwp0A\Interface");
uint16 i2 = getTag
        ("\nbhwpe0\ToInterface:nbhwp0A\Interface");
uint16 s1 = getTag ("\nbhwpe0\FromStack:nbhwp0A\Stack");
uint16 s2 = getTag ("\nbhwpe0\ToStack:nbhwp0A\Stack");
uint16 i3 = getTag
        ("\nbhwpe0\FromInterface:nbhwp0B\Interface");
uint16 i4 = getTag
        ("\nbhwpe0\ToInterface:nbhwp0B\Interface");
uint16 s3 = getTag ("\nbhwpe0\FromStack:nbhwp0B\Stack");
uint16 s4 = getTag ("\nbhwpe0\ToStack:nbhwp0B\Stack");
uint16 j1 = getTag
        ("\nbhwpe1\FromInterface:nbhwp1A\Interface");
uint16 j2 = getTag
        ("\nbhwpe1\ToInterface:nbhwp1A\Interface");
uint16 t1 = getTag ("\nbhwpe1\FromStack:nbhwp1A\Stack");
uint16 t2 = getTag ("\nbhwpe1\ToStack:nbhwp1A\Stack");
uint16 j3 = getTag
        ("\nbhwpe1\FromInterface:nbhwp1B\Interface");
uint16 j4 = getTag
        ("\nbhwpe1\ToInterface:nbhwp1B\Interface");
uint16 t3 = getTag ("\nbhwpe1\FromStack:nbhwp1B\Stack");
uint16 t4 = getTag ("\nbhwpe1\ToStack:nbhwp1B\Stack");
printf ("TAGS PE0: FROMA=%d, TOA=%d, STACKFROMA=%d,
        STACKTOA=%d\n", i1, i2, s1, s2);
printf ("TAGS PE0: FROMB=%d, TOB=%d, STACKFROMB=%d,
        STACKTOB=%d\n", i3, i4, s3, s4);
printf ("TAGS PE1: FROMA=%d, TOA=%d, STACKFROMA=%d,
        STACKTOA=%d\n", j1, j2, t1, t2);
printf ("TAGS PE1: FROMB=%d, TOB=%d, STACKFROMB=%d,
        STACKTOB=%d\n", j3, j4, t3, t4);
} catch (NBEError E) {
    ...
}

```

getStackDriverName Method

Retrieves the driver name for an interface.

```
DWORD getStackDriverName (char* argAceName,
                          char *argdriverName);
```

Argument	Description
argAceName	The full path to a system ACE for a stack interface, as described in Appendix C, "Policy Accelerator Name Space."
argdriverName	[OUT] On return, points to the driver name for the specified interface.

Returns When successful, NB_SUCCESS. When not successful, throws the NB_Error code NBERROR_NBAPPL_ERRPECONTEXT.

Description Use the driver name returned in the argdriverName argument to obtain interface configuration information, such as IP address or net mask, using your operating system's services.

Example

```
/* Get the stack's driver name in main function */
void main( void ) {
    try {
        MyAppl *appl = new MyAppl ();
    }
    catch (NError E){
        NB_ABORT (1);
    }
    char temp [32];
    try{
        appl->getStackDriverName
            ( "/nbhwpe0/ToInterface:nbhwpe0A/Interface", temp );
    }
    catch (NError E){
        NB_ABORT (1);
    }
    printf ("Name of driver for stack A on accelerator 0 is
    %s\n",
           temp);
    while (1) Sleep(99999);
}
```

link Method

Connects a crosscall to a crosscall handler.

```
DWORD link (char * from,
            char * to);
```

Argument	Description
from	The ACE and crosscall object issuing crosscalls. Specify the path as <i>Appl\AceGroup\Ace\crosscall</i> , as described in Appendix C, “Policy Accelerator Name Space.”
to	The ACE and crosscall handler object receiving crosscalls. Specify the path as <i>Appl\AceGroup\Ace\crosscallhandler</i> .

Returns When successful, `NB_SUCCESS`. When not successful, one of the following codes:

Return Codes	Description
<code>NBERROR_NBAPPL_NULLNAMES</code>	The pointer that should contain the object name is <code>NULL</code> . (That is, one of the arguments is a <code>NULL</code> pointer.)
<code>NBERROR_NBAPPL_CANNOTSENDLINKREQ</code>	The application cannot communicate with the Resolver.
<code>NBERROR_NBAPPL_CANNOTRECVLINKREQ</code>	The application lost communication with the Resolver during execution of the link request.
<code>NBERROR_NBAPPL_ERRLINKREQ</code>	Either the target or the destination ACE does not exist.

Description You use crosscall and crosscall handler objects to send messages from one ACE to another. Before you can send a message, you must use this method to associate the `Crosscall` object in the sending ACE with the `CrosscallHandler` object in the receiving ACE. The crosscall handler receives and acts on messages sent using a crosscall.

Any number of `Crosscall` objects can be linked to the same `CrosscallHandler` object.

`Crosscall` and crosscall handler classes are defined in the ASL. You create these objects in the action code for each ACE's accelerator module. Although each of these objects is paired with a manager object in the host module (`CrosscallManager` and `CrosscallHandlerManager` objects), it is the ASL objects, not the manager objects, that you pass to this method.

See Also

- `unlink` Method
- `Crosscall` Class and `CrosscallHandler` Class in Chapter 4, “Action Services Library.”
- “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*

unbind Method

Unbinds a target from an ACE.

```
DWORD unbind (char * from);
```

Argument	Description
from	The target to be unbound from the ACE. Specify the path as <i>Appl\AceGroup\Ace\target</i> , as described in Appendix C, “Policy Accelerator Name Space.”

Returns When successful, `NB_SUCCESS`. When not successful, one of the following codes:

Return Codes	Description
<code>NBERROR_NBAPPL_NULLNAMES</code>	The pointer that should contain the object name is <code>NULL</code> . (That is, the argument is a <code>NULL</code> pointer.)
<code>NBERROR_NBAPPL_CANNOTSENDBINDREQ</code>	The application cannot send a message to the Resolver to request a binding.
<code>NBERROR_NBAPPL_CANNOTRECVBINDREQ</code>	The application lost communication with the Resolver during execution of the unbind request.
<code>NBERROR_NBAPPL_ERRBINDREQ</code>	The target does not exist.

Description You must unbind a target from any currently bound ACE before you can bind it to a different ACE.

Targets represent possible destinations for packets. Packets delivered to unbound targets are dropped.

See Also

- `bind` Method
- Target Class in Chapter 4, “Action Services Library.”
- Chapter 5, “Controlling Packet Flow,” in *Developing Applications Using the IX-API SDK*

unlink Method

Disconnects a crosscall from a crosscall handler.

```
DWORD unlink (char * from);
```

Argument	Description
from	The crosscall to be disconnected. Specify the path as <i>Appl\AceGroup\Ace\crosscall</i> , as described in Appendix C, “Policy Accelerator Name Space.”

Returns When successful, NB_SUCCESS. When not successful, one of the following codes:

Return Codes	Description
NBERROR_NBAPPL_NULLNAMES	The pointer that should contain the object name is NULL. (That is, one of the arguments is a NULL pointer.)
NBERROR_NBAPPL_CANNOTSENDLINKREQ	The application cannot communicate with the Resolver.
NBERROR_NBAPPL_CANNOTRECVLINKREQ	The application lost communication with the Resolver during execution of the unlink request.
NBERROR_NBAPPL_ERRLINKREQ	The crosscall does not exist.

Description Use this method to disconnect the specified crosscall from any crosscall handler to which it was linked using the `link` method. Unlink a crosscall before linking it to a new crosscall handler.

When a crosscall is not linked to any crosscall handler, you cannot use it to send messages.

The `Crosscall` class is defined in the ASL. You create these objects in the action code for each ACE’s accelerator module. Although each of these objects is paired with a `CrosscallManager` object in the host module, it is the ASL object, not the manager object, that you pass to this method.

- See Also**
- `link` Method
 - `Crosscall` Class and `CrosscallHandler` Class in Chapter 4, “Action Services Library.”
 - “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*

NError Class

The `NError` class represents errors that can be generated by methods in the host API. Use the `getErrorCode` method to access the error code contained in an `NError` object.

Error objects of this type are returned by host API methods. Host API object constructors return a reference to the new object, and throw an error object, which you can access using a `catch` statement.

You can add your own uniquely numbered error codes to provide information about failed operations. Give your own error codes numbers greater than the constant `NERROR_USER_BASE`. For an example of how to define your own error codes, see “Preparing for Error Handling” on page 20 of *Developing Applications Using the IX-API SDK*.

This class is not derived from any other class. It contains the following method:

Method	Description
<code>getErrorCode</code> Method	Retrieves the error code in the error object returned by a host API method, or thrown by an object constructor.

See Also

Chapter 11, “Debugging and Troubleshooting,” in *Developing Applications Using the IX-API SDK*

getErrorcode Method

Retrieves the error code in the error object returned by a host API method, or thrown by an object constructor.

DWORD getErrorcode (void)

Returns The error code constant encapsulated by the error object. For a complete list and description of error codes, see Appendix A, “IX-API SDK Host API Error Codes.”

Example The following code fragment uses the `getErrorcode` method in a `catch` statement after a call to an object constructor to print out a debugging message:

```
try { crosscallMgr = new CrosscallManager (appl, AceGroup, this,
                                           "crosscallTest");
}
catch (NError E) {
    fprintf(stderr, "Demo app caught NError 0x%X\n",
           E.getErrorcode ());
    NB_ABORT (1);
}
```

NObject Class

All classes of the host API, with the exception of the `Message` and `Message-Block` classes, are derived from `NObject`. `NObject` provides derived classes with the basic ability to get certain object properties.

This class is not derived from any other class. You do not instantiate it or use it directly. It contains the following methods:

Method	Description
<code>getId</code> Method	Returns the identifier of the object.
<code>getType</code> Method	Retrieves the object's class.

getId Method

Returns the identifier of the object.

```
LONG getId (void);
```

Returns The identifier of the object.

Description An object's identifier is a global reference to that object. Because an object ID is independent of the process context, it is not context-sensitive.

getType Method

Retrieves the object's class.

```
NBOBJTYPE getType (void);
```

Returns The object type, which can be one of the following:

- `OBJECT_TYPE_NBAPPL`
- `OBJECT_TYPE_ACE`

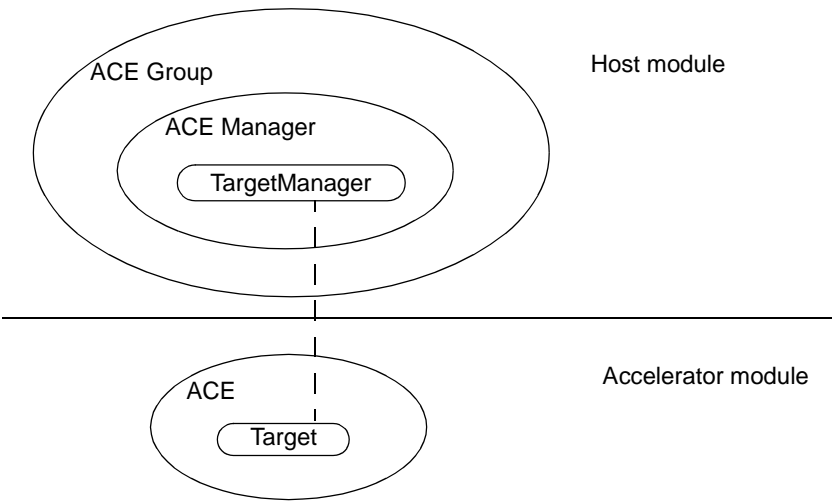
NBObject Class

- OBJECT_TYPE_TARGET
- OBJECT_TYPE_UPCALL
- OBJECT_TYPE_DOWNCALL
- OBJECT_TYPE_ACEGROUP
- OBJECT_TYPE_CCALLSEND
- OBJECT_TYPE_CCALLRECV

Description Use this method to determine the types of methods you can use on an object.

TargetManager Class

Use the `TargetManager` class to define and create target managers. Target managers are objects on the host that manage targets, which reside in ACEs on the Policy Accelerator. Each `TargetManager` object corresponds to exactly one `Target` object with the same dictionary name in the accelerator module.



The `TargetManager` class contains the following method:

Method	Description
<code>TargetManager</code> Constructor	Instantiates the class.

Class Derivation

The `TargetManager` class is derived from the `NBObject` class, inheriting all its public methods.

See Also

- Chapter 5, “Controlling Packet Flow,” of *Developing Applications Using the IX-API SDK*
- `Target` Class in Chapter 4, “Action Services Library.”

TargetManager Constructor

Creates a `TargetManager` object.

```
TargetManager (NBAppI* argAppl,
               AceGroup* argAceGroup,
               AceManager * argAceMgr,
               char * argName) throws NError;
```

Argument	Description
<code>argAppl</code>	The object name of the application that created the ACE group specified in <code>argAceGroup</code> .
<code>argAceGroup</code>	The object name of the ACE group that contains the ACE manager specified in <code>argAceMgr</code> .
<code>argAceMgr</code>	The object name of the ACE manager that contains this target manager.
<code>argName</code>	The dictionary name of this target manager. This must be the same as the dictionary name of the corresponding <code>Target</code> object in the accelerator module. Must be unique among objects in this ACE manager.

Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
<code>NBERROR_NBOBJ_NULLNAME</code>	The name of the target manager (<code>argName</code>) is NULL.
<code>NBERROR_NBOBJ_NAMETOOLONG</code>	The name of the target manager (<code>argName</code>) is too long. The maximum length of the object name is <code>OBJNAME_MAXLEN</code> , which is defined in <code>NBapi\nbparam.h</code> .
<code>NBERROR_NBOBJ_OUTOFMEMORY</code>	Cannot allocate memory to create target manager.
<code>NBERROR_TARGETMGR_NULLAPPL</code>	The <code>argAppl</code> argument is NULL.
<code>NBERROR_TARGETMGR_NULLACEGROUP</code>	The <code>argAceGroup</code> argument is NULL.

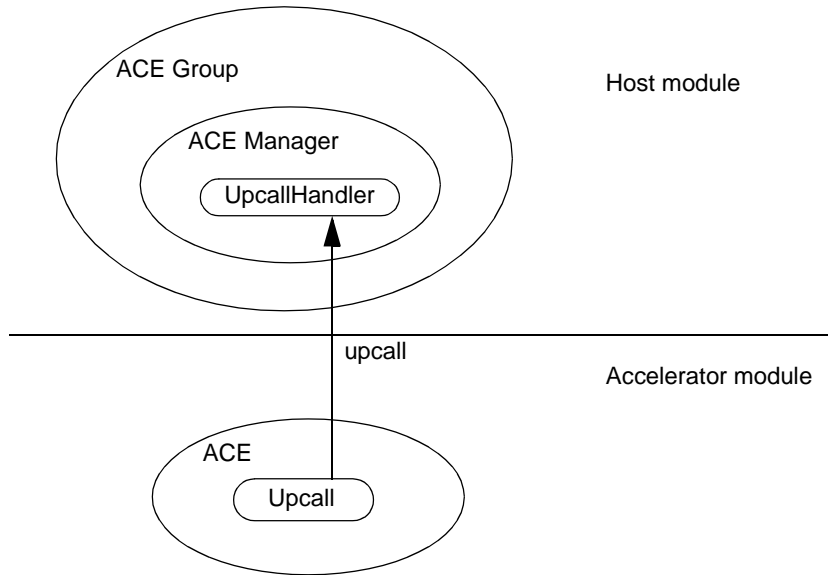
Return Codes	Description
NBERROR_TARGETMGR_NULLACEMGR	The argAceMgr argument is NULL.
NBERROR_TARGETMGR_CANNOTREGISTER	Cannot register this target manager with the Resolver for one of the following reasons: <ul style="list-style-type: none">■ The Resolver is not running.■ The Policy Accelerator driver is not running.■ There is a problem communicating with the Policy Accelerator.■ The name has already been registered.

Description Target managers manage targets in ACEs on the Policy Accelerator.

UpcallHandler Class

Use the `UpcallHandler` class to define and create upcall handlers. Upcall handlers are objects on the host that receive messages from upcall objects on the Policy Accelerator. For information on the `Upcall` Class, see Chapter 4, “Action Services Library.”

Each `UpcallHandler` object corresponds to exactly one `Upcall` object with the same dictionary name in the accelerator module.



To make applications run faster, do most packet processing on the Policy Accelerator. Use upcalls and downcalls for statistical and administrative functions. If you need to transfer large amounts of data between the Policy Accelerator and the host, do so using the stack.

The `UpcallHandler` class contains the following methods:

Method	Description
<code>UpcallHandler</code> Constructor	Instantiates the class.
<code>getUpcallFunction</code> Method	Retrieves the upcall callback method from the ACE manager.

Class Derivation The UpcallHandler class is derived from the NBOBJECT class, inheriting all its public methods.

Example The following example of upcall usage is from the TwoAceApp demo application. The definition of the ACE manager subclass declares a callback method and an upcall handler object:

```
class NBACEOneMgr : public AceManager {
public:
    NBACEOneMgr(NBAppI* appl, AceGroup* aceGroup, char* name);
    ~NBACEOneMgr();
    // Upcall
    void reportPacketCount(Message* m);
    UpcallHandler* reportPacketCountUpcallHandle;
};
```

The constructor and destructor for the ACE manager subclass create and delete the upcall handler object:

```
NBACEOneMgr::NBACEOneMgr (NBAppI* appl,
                          AceGroup* acegroup,
                          char* name):
    AceManager(appl, acegroup, name)
{
    try {
        // create upcall handler object
        reportPacketCountUpcallHandle =
            new UpcallHandler (appl, this,
                              "reportPacketCountUpcall",
                              (UpcallFp)reportPacketCount);
    }
    catch (NBEError E) {
        throw;
    }
    // init accelerator side of ACE: load rules, actions
    if (load ("aceOneRules", "aceOneActions") != NB_SUCCESS)
    {
        throw NBEError (NB_ERROR(NBERROR_TWOACEAPP));
    }
}

NBACEOneMgr::~NBACEOneMgr()
{
    delete reportPacketCountUpcallHandle;
}
```

The ACE constructor in the accelerator module must create a corresponding upcall object using the same dictionary name, reportPacketCountUpcall.

This method is an example of an upcall handler callback. The method decodes the message and prints out the data. Finally, it frees the host memory that was allocated for the message, using the `releaseMessage` method.

```
void NBaceOneMgr::reportPacketCount (Message* m)
{
    printf ("NoOfPackets: %05d\n", * (int*) m->getBuffer1 ());
    releaseMessage (m);
}
```

See Also

- “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*
- Upcall Class in Chapter 4, “Action Services Library.”

UpcallHandler Constructor

Creates an UpcallHandler object.

```
UpcallHandler (NBAppI* argAppI,
               AceGroup* argAceGroup,
               AceManager * argAceMgr,
               char * argName)
               UpcallFp argUpcallFunction) throws NError;
```

Argument	Description
argAppI	The object name of the application that created the ACE group specified in argAceGroup.
argAceGroup	The object name of the ACE group that contains the ACE manager specified in argAceMgr.
argAceMgr	The object name of the ACE manager that contains this upcall.
argName	The dictionary name of this upcall handler. This must be the same as the dictionary name of the corresponding Upcall object in the accelerator module. Must be unique among objects in this ACE manager.
argUpcallFunction	The routine to call when the Policy Accelerator invokes this upcall. This must be a method defined in the ACE manager object.

Returns When successful, a reference to the newly created object. When not successful, throws an exception of type `NError` with one of the following error codes, which you can access using the `NError` method `GetErrorcode`.

Return Codes	Description
NBERROR_NBOBJ_NULLNAME	The name of this upcall (argName) is NULL.
NBERROR_NBOBJ_NAMETOOLONG	The name of the upcall (argName) is too long. The maximum length of the object name is OBJNAME_MAXLEN, which is defined in <code>NBapi\nbparam.h</code> .
NBERROR_NBOBJ_OUTOFMEMORY	Cannot allocate memory to create the upcall.
NBERROR_NBOBJ_NULLAPPL	The argAppI argument is NULL.
NBERROR_NBOBJ_NULLACEGROUP	The argAceGroup argument is NULL.

Return Codes	Description
NBERROR_UPCALL_NULLACEMGR	The <code>argAceMgr</code> argument is NULL.
NBERROR_UPCALL_CANNOTREGISTER	Cannot register this upcall handler with the Resolver for one of the following reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.

Description UpcallHandler objects on the host receive messages from Upcall objects on the Policy Accelerator. Every Upcall object on the Policy Accelerator must have a corresponding UpcallHandler object with the same name on the host.

You are responsible for freeing the local memory associated with the received message when it is no longer required, using the `releaseMessage` method; see `AceManager` Class.

Upcall Handler Callbacks You must define the upcall handling callback as a method in a subclass of the `AceManager` class. It must conform to the following prototype:

```
class MyAceMgr : public Ace {
    void my_handler (Message *m); ...}
```

The following example is from the `basicApp` demo application. The upcall handler callback, `peekPacketUpcall`, is defined as a method in the applications `AceManager` subclass, `NBBasicAce`. The method decodes the message to restore the byte order, prints the data, then frees the memory that was allocated for it on the host.

```
void NBBasicAce::peekPacketUpcall (Message* m) {
    NB_ASSERT (m->getLen1 () == sizeof (nuint32));
    printf ("NoOfPackets: %05d\n",
        ntohs (* (nuint32 *) m->getBuffer1 ()));
    releaseMessage (m);
}
```

Specify this callback when you create the upcall handler object:

```
peekPacketUpcallHandle =
    new UpcallHandler (appl, acegroup, this,
        "peekPacketUpcall",
        (UpcallFp)peekPacketUpcall);
```

- See Also
- [AceManager Class](#)
 - [Upcall Class](#), [Message Class](#), [MessageBlock Class](#) in Chapter 4, “Action Services Library.”
 - “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*
 - “Byte Order and Intermodule Communication” on page 12 in Chapter 2, “System Types and Methods.”

getUpcallFunction Method

Identify the current upcall handling callback method.

```
UpcallFp getUpcallFunction (void);
```

Returns The upcall callback method associated with this upcall handler.

Description Use this method if your application dynamically changes upcall service function callbacks. These callbacks must be defined in the ACE manager object.

Chapter 4

Action Services Library



This chapter describes the *Action Services Library* (ASL). You use this set of C++ library functions in action code in the accelerator module portion of your application. Action code uses ACEs to perform packet processing and to direct packet flow.

This chapter contains two parts:

- An introduction to the classes and functions of the ASL API by functional area:
 - “Initialization” on page 97
 - “Action Functions” on page 97
 - “Packet Moving Classes” on page 97
 - “String Search Classes” on page 98
 - “Message Support Classes” on page 99
 - “Time Support Classes” on page 100
 - “Statistical Support Class” on page 101
 - “Set Management Classes” on page 101
 - “Memory Management Classes and Functions” on page 103
 - “Interface Management Classes” on page 105
 - “Base Classes” on page 105
- An alphabetical listing of the classes and functions with complete details:
 - “The Action Services Library (ASL) API” on page 107

Overview

The Action Services Library (ASL) provides support for developing network applications. The ASL:

- Provides a basic class framework for representing network data packets (the `Buffer` class), and for sending them to different destinations on the network (the `Target` class).

- Together with the Network Classification Language (NCL) and the host API, supports the ACE structure; message sending using upcalls, downcalls, and crosscalls; and the association of arbitrary data with packets using sets and searches.
- Provides support for some basic system services, such as timers, statistical counters, and memory management.

TCP/IP Support

Extensions to the ASL provide support for action code to handle many TCP/IP functions such as IP fragmentation and reassembly, network address translation (NAT), and TCP connection monitoring, including stream reconstruction.

- For information on the TCP/IP extensions, see Chapter 5, “ASL Extensions for TCP/IP.”

Environmental Restrictions

Action code is downloaded from the host into the Policy Accelerator, an environment in which some customary services of a full programming environment are not available. The following services are not available to the action code part of an application:

- Floating-point math
- File system access
- Multithreading

The IX-API SDK provides all portions of the ANSI Standard C and C++ libraries that do not conflict with these environmental restrictions. See the documents *Installing the IX-API SDK* and *IX-API SDK Release Notes* for specific information on which tools and compilers are supported.

Include Files

To use all ASL classes except the string search classes, include the following header file in your code:

```
#include <NBAction/NBAction.h>
```

To use the string search classes, include the following header file in your code:

```
#include <NBAction/NBStringSearch.h>
```

For more information, see “String Search Classes” on page 98.

Initialization

You must provide a top-level initialization function in your action code that creates an ACE object using the passed parameters. The Policy Accelerator calls this function immediately after receiving the NCL and action code from the host.

Function	Description
<code>init_actions</code> Function	Initializes the Policy Accelerator portion of the network application by constructing the specified ACE object.

For more information on the syntax and usage, see “Initialization Function” on page 172.

Action Functions

The ASL provides the following functions that you can call directly in the action part of a rule in NCL:

Function	Description
<code>action_pass</code> Function	Routes a packet to the ACE's pass target.
<code>action_drop</code> Function	Routes a packet to the ACE's drop target.

You can also define your own action functions, using a prescribed syntax; see “Action Functions” on page 115.

Packet Moving Classes

Most applications contains actions that specify packet disposition. Applications can move packets by simply passing or dropping them, or can take more complex actions, such as splitting incoming packets into one of several outgoing packet streams.

Use the following classes when moving packets:

Class	Description
<code>Ace Class</code>	Represents an ACE in the Policy Accelerator. Passes and drops packet buffers.
<code>Buffer Class</code>	Represents packet information in buffers.
<code>Target Class</code>	Represents packet destinations in ACEs on the Policy Accelerator.

String Search Classes

The ASL provides a high-performance string search facility that allows you to search for strings within one or more packet buffers. You can search for occurrences of a constant string, or for all strings that match a regular expression. You can search the buffer or buffers for matches to several search strings at once. You use a tag or identifier to determine which of several search strings matches a found string.

String Search Management

When you plan to use the string search facility in an ACE:

- You must specify the `ACE_STRINGSEARCH` flag in the `argAceMode` argument when you construct the `AceManager` object in the host module for that ACE.
- The ACE object in the accelerator module must contain a reference to the string search context and engine objects.

For more information on ACE managers, see “Application and ACE Management Classes” in Chapter 3, “Host API.”

Initiating and Continuing Searches

You initiate a search using the `SearchBuffer` method, passing the current packet buffer with a new or reset context object. Typically, an action function calls this method.

If the context object specifies that the search can span multiple buffers, and if a search is already in progress for that context, the `SearchBuffer` passes a new buffer to the ongoing search. You can maintain multiple searches simultaneously, as long as each search is associated with its own context object.

You provide callback functions to act on the results of the search, to be invoked each time a matching string is found (a per-match callback) or each time a search is completed for a buffer (a per-buffer callback). The per-buffer callback determines how to dispose of the buffer.

**Search
Operating
Modes**

String searches are performed asynchronously. You must disable the search mechanism while you specify search strings or set other search parameters. When the search engine cannot execute an action immediately, it notifies the application of completion by invoking a callback function, which you provide.

**String Search
Classes**

Use the following classes to search for strings in packet buffers:

Class	Description
NBStringSearchEngine Class	Specifies search strings and initiates searches.
NBSearchContext Class	Configures string searches and maintains a multiple-buffer search context.
NBStringMatchReport Class	Allows access to the results of a string search in a buffer.



NOTE: To use the string search classes, include the following header file in your code:

```
#include <NBAction/NBStringSearch.h>
```

**For More
Information**

See Chapter 10, “Finding Strings in Packets,” in *Developing Applications Using the IX-API SDK*

Message Support Classes

Applications frequently need to pass configuration changes from the host to a Policy Accelerator or pass summary information back. The ASL provides access to the asynchronous messaging system, in the form of `Upcall` and `DowncallHandler` objects. The ASL also provides the `Message` and `MessageBlock` classes for constructing messages.

Because the messaging system is asynchronous, you supply a callback function in each message handler object, which is executed when the message is received.

Use network-ordered storage classes when passing multibyte integers in messages; see “Byte Order and Intermodule Communication” in Chapter 2, “System Types and Methods.”

Use the following classes to create and send messages between the host and Policy Accelerator or between two ACEs in the same or different Policy Accelerators:

Class	Description
Crosscall Class	Sends messages from this ACE to another ACE in the same or another Policy Accelerator.
CrosscallHandler Class	Directs crosscall messages from another ACE to a service function in this ACE.
DowncallHandler Class	Directs downcall messages from the host to a service function in the Policy Accelerator.
Message Class	Encapsulates data as messages to send in upcalls and crosscalls.
MessageBlock Class	Encapsulates a storage area within the Policy Accelerator memory for future call messages.
Upcall Class	Sends messages from the Policy Accelerator to the host.

For More Information

See Chapter 8, “Communication Within an Application,” in *Developing Applications Using the IX-API SDK*

Time Support Classes

The time support classes provide for the observation of time, scheduling of events, and correlation of arbitrary counts with time to provide useful reports for both packet rates and data rates.

Use the following classes to schedule and track events:

Class	Description
Event Class	Schedules and cancels events as necessary.
Rate Class	Tracks event rates and bandwidths so you can watch for rates that exceed desired values.
Time Class	Handles time values.

Statistical Support Class

The Policy Accelerator maintains counters of a variety of network traffic events on the MAC interfaces of the Policy Accelerator, using RMON block counters. You can use the counters to construct RMON groups and MIBs. You access these counters and control the query rate using the `NBRmon` class:

Class	Description
NBRmon Class	Collects statistics on network traffic events.

Set Management Classes

The ASL and Network Classification Language (NCL) together support data tables called *sets*, which are associated with named *searches*. Sets and searches are initially created according to definitions in NCL, as described in “Sets and Named Searches” on page 403. You generate action code directly from these definitions using the NCL compiler, as described in “Synchronizing NCL with Action Code” on page 411. The compiler generates a header file with class definitions that correspond to the NCL definitions. For each set *setname*, the NCL compiler creates a subclass of the `Set` class named `Set_setname`, and a subclass of the `Element` class named `Elt_setname`.

Declaring Sets

To use the sets and searches that you define in NCL, you must do the following things in your action code:

- Include the generated header file containing the set, element, and search class definitions.
- Extend those class definitions in further subclasses to add the functionality you want. It is best to do this in a file other than the generated header file, as the header file can be overwritten if you regenerate it.
- Create an object of the customized set subclass as part of the ACE object. Add a set declaration for each set to your subclass of the ACE class, with the same name that you declared for that set in the NCL file. See “`Set_setname` Class” on page 256 for an example.

Searches on Sets

When you generate the set header file from the NCL file, the NCL compiler creates an instance of the `Search` class for each defined search, to contain the search result. Both the search and its ASL result object are named in the form *setname.searchname*.

For each incoming packet, the Policy Accelerator tries every search defined in the NCL file. If the `requires` clause succeeds, the Policy Accelerator executes the search and stores the result in the `Search` object. The execution of a search in NCL determines whether the incoming packet has a matching set element.

A rule can refer to the search name on either the predicate side or the action side.

- On the predicate side, the search name acts as a Boolean expression that is `TRUE` when the search was executed and succeeded, and is `FALSE` if the search was not executed or failed.
- When a search name appears on the action side, the action function can access the corresponding search object to find the search result:
 - For a successful search, the search object provides a pointer to the element that was located.
 - On failure, the search object provides a pointer to a location at which an element can be inserted in the set.

See “Sets and Named Searches” on page 403 for more information about searching sets.

Set Elements

For each named set, the NCL compiler constructs a subclass of the `Element` class, named `Elt_setname`, whose `new` operator creates a member with the proper keys. Keys are stored as network-byte-order 32-bit unsigned integers. (See “Search Key Format” on page 102.)

To define the data associated with the element, you extend this class, overloading the constructor to initialize custom data fields. The elements stored in each set are members of your subclass. You populate a set through actions that create new elements or manipulate existing elements.

It is very important that the memory allocated for set elements be strictly aligned and offset from cache boundaries. For this reason, you always use the `new` operator to create set elements, rather than using the constructor directly. Similarly, you use the `delete` operator to remove elements from the set, rather than the destructor method, since `delete` cleans up internal references to the element.

Search Key Format

The search keys in a set element are compared to field values from network packets, as specified by the search definition in NCL. Keys are stored as network-byte-ordered 32-bit integers. When you use a shorter field as a key for set elements, it is zero-extended to 32 bits before being converted from host to network order.

Because the protocol field accessors generated from NCL code are defined to return results in network byte order, you can set or compare key values directly to protocol field values.

In the following example, four keys of different sizes are defined for elements of the set `myset`:

```
uint8    proto = 16;
uint16   ident = 1234;
nuint16  code = 0xC0DE; // 0xC0, 0xDE
uint32   here = 0x0A000203; // 10.0.2.3

Elt_myset * elt = new Elt_myset (here, code, ident, proto);
```

Set, Search, and Element Classes

Use the following classes to create and manage sets and their elements:

Class	Description
Element Class	The base class for set element objects.
Elt_setname Class	The NCL compiler creates an <code>Element</code> subclass <code>Elt_setname</code> for each named set to represent elements of that set. You create further subclasses and use them to create and manipulate elements of the named set.
Search Class	Allows access to the results of a search. Points to a found element, or to a location to insert a new element.
Set Class	The base class for set objects.
Set_setname Class	The NCL compiler creates a <code>Set</code> subclass <code>Set_setname</code> to represent each named set. Use to search for elements in a set.

For More Information

See Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*

Memory Management Classes and Functions

The ASL provides some facilities for controlling and monitoring memory usage in the Policy Accelerator.

**Controlling
Memory Usage**

The memory for the Policy Accelerator is divided into a stack (for system usage), a heap (used for data sets), and packet buffer memory. You can, to some extent, control how available memory is partitioned between the heap and packet buffer memory.

The pool of memory allocated to packet buffers starts out at a value large enough for the system to function, and grows when appropriate. You can limit the total number of network packet buffers in the system by editing the configuration file *SDKinstallpath\hpex\hpex.cfg* to insert the following directive:

```
maxbuf number
```

This sets a soft limit to the number of network packet buffers for the Policy Accelerator. Because the Policy Accelerator allocates a fixed minimum and always increases it by a fixed amount, the actual number of packet buffers is not round and the system may exceed this limit.

If the configuration variable is not set, the amount of memory being used for packet buffers will grow on demand up to half of all memory, or until it meets the system heap growing in the other direction.

**Monitoring
Memory Usage**

A set of methods in the `Buffer` class allow you to monitor the number of free buffers available for packet flow. See “Buffer Class” on page 123. In addition, you can use the following class to monitor buffer memory usage:

Class	Description
Backlog Class	Monitors packet backlogs in buffer memory.

Use the following global functions to monitor heap size and activity at run time:

Function	Description
getmemstatvalues Function	Retrieves the current total used and free memory on the heap.
mstats Function	Displays memory usage statistics for the heap.

Interface Management Classes

The ASL defines the following classes that help you monitor the MAC interfaces of a Policy Accelerator:

Class	Description
NBInterfaceProp Class	Manages the specific properties of a MAC interface, such as its MAC address, speed, and duplex capability.
NBLinkwatch Class	Monitors the network connection for a MAC interface.

Base Classes

The ASL defines base classes from which the user-level classes are derived. These base classes implement the architectural, structural, and referential features needed by most classes.

Memory Allocation

The customary memory allocation primitives are available from the standard libraries (that is, C programmers can use `malloc`, `calloc`, `realloc`, and `free`; C++ programmers can use the `new` and `delete` operators).

To accelerate memory allocations when many objects of a fixed size are being rapidly allocated and freed, the ASL keeps a private list of objects to be recycled. This list is implemented using the `Pool` and `Tagged` classes. The `Dynamic` class is a base class that overloads the `new` and `delete` operators to make use of this memory management scheme. Most of the ASL classes are descended from the `Dynamic` class.

The following base classes are used in allocating memory for objects in the Policy Accelerator:

Class	Description
<code>Dynamic</code> Class	Provides fast pool allocations for your objects.
<code>Pool</code> Class	Used by the <code>Tagged</code> class to quickly allocate objects of fixed sizes at specified offsets from specified power-of-two alignments.
<code>Tagged</code> Class	Used by the <code>Dynamic</code> class to free tagged objects into the appropriate memory pool.

Name Space

Additional base classes implement the namespace that associates objects in the accelerator module with their counterparts and managers in the host module.

- Some objects in the Policy Accelerator require managers or other related objects on the host. The `Dualobj` base class implements the connection between these paired objects. Objects that are paired in the host module and accelerator module are associated with each other by having the same dictionary name. For more information, see Appendix C, “Policy Accelerator Name Space.”
- The `Name` and `Named` base classes implement an internal name space.
- Objects descended from the `Linked` base class can be ordered in a circular linked list, in which you can access a next and previous object.

Many ASL classes are derived from the following base classes:

Class	Description
<code>Dualobj</code> Class	Determines which objects in the Policy Accelerator correspond with which objects in the host.
<code>Name</code> Class	Maintains an internal name database for objects.
<code>Named</code> Class	Assigns internal names to objects.
<code>Linked</code> Class	Links objects with each other to form a ring.

The Action Services Library (ASL) API

- This section provides a detailed description of each ASL class and function.
- Classes are listed in alphabetical order. Within each class, the constructor and destructor for that class are listed first, followed by the remaining methods in alphabetical order.
 - Global functions are grouped by usage, and are described in the sections “Initialization Function,” “Action Functions,” and “Memory Management Functions.”

Include Files To use these classes, include the following header file in your code:

```
#include <NBaction/NBaction.h>
```

Classes and Functions The ASL API contains the following classes and functions:

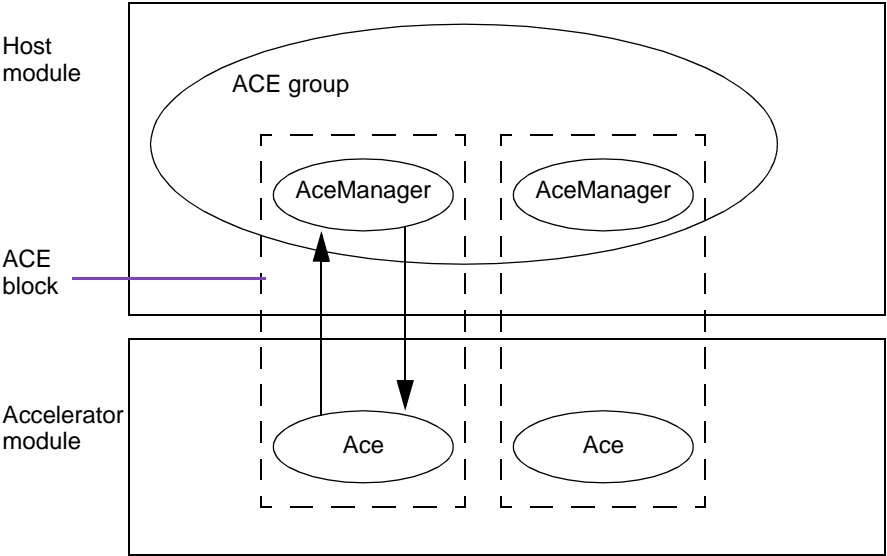
Class/Function group	Description
Ace Class (page 110)	Represents an ACE in the Policy Accelerator. Passes and drops packet buffers.
Action Functions (page 115)	action_drop Function: Routes a packet to the ACE's drop target. action_pass Function: Routes a packet to the ACE's pass target.
Backlog Class (page 119)	Monitors packet backlogs in buffer memory.
Buffer Class (page 123)	Represents and manipulates packet information in buffers.
Crosscall Class (page 139)	Sends messages from this ACE to another ACE in the same or another Policy Accelerator.
CrosscallHandler Class (page 144)	Directs crosscall messages from another ACE to a service function in this ACE.
DowncallHandler Class (page 150)	Directs downcall messages from the host to a service function in the Policy Accelerator.
Dualobj Class (page 155)	Keeps track of which objects in the Policy Accelerator correspond with which objects in the host. A base class.

Class/Function group	Description
Dynamic Class (page 157)	Provides fast memory pool allocations for objects. A base class.
Element Class (page 159)	Represents set elements. A base class.
Elt_setname Class (page 160)	Represents set elements. The NCL compiler creates an <code>Element</code> subclass <code>Elt_setname</code> for each named set to represent elements of that set. You create further subclasses and use them to create and manipulate elements of the named set.
Event Class (page 166)	Schedules and cancels events.
Initialization Function (page 172)	<code>init_actions</code> Function: Initializes the Policy Accelerator portion of the network application by constructing the specified ACE object.
Linked Class (page 174)	Links objects with each other to form a ring. A base class.
Memory Management Functions (page 178)	<code>getmemstatvalues</code> Function: Retrieves the current total used and free memory on the heap. <code>mstats</code> Function: Displays memory usage statistics for the heap.
Message Class (page 180)	Encapsulates data to send in upcalls and crosscalls.
MessageBlock Class (page 184)	Encapsulates a storage area within the Policy Accelerator memory for future call messages.
Name Class (page 191)	Maintains an internal name database for objects. A base class.
Named Class (page 194)	Assigns internal database names to objects. A base class.
NBInterfaceProp Class (page 197)	Manages the specific properties of a MAC interface, such as its MAC address, speed, and duplex capability.
NBLinkwatch Class (page 205)	Monitors the network connection for a MAC interface.
NBRmon Class (page 208)	Collects statistics on network traffic events.
NBStringMatchReport Class (page 216)	Accesses the results of a string search in a buffer.

Class/Function group	Description
NBSearchContext Class (page 222)	Configures string searches and maintains a multiple-buffer search context.
NBStringSearchEngine Class (page 233)	Specifies search strings and initiates searches.
Pool Class (page 245)	Quickly allocates objects of fixed sizes at specified offsets from specified power-of-two alignments. A base class, used by the Tagged class.
Rate Class (page 248)	Tracks event rates and bandwidths.
Search Class (page 251)	Gets results of a search. Points to a found element, or to a location to insert a new element.
Set Class (page 255)	Represents sets. A base class.
Set_setname Class (page 256)	Represents sets. The NCL compiler creates a Set subclass Set_setname to represent each named set. Use to search for elements in a set.
Tagged Class (page 262)	Frees tagged objects into the appropriate memory pool. A base class, used by the Dynamic class.
Target Class (page 265)	Represents packet destinations in ACEs on the Policy Accelerator.
Time Class (page 268)	Represents and handles time values.
Upcall Class (page 273)	Sends messages from the Policy Accelerator to the host.

Ace Class

An *Action/Classification Engine* (ACE) is the primary object for processing packets through an IX-API SDK application. It is a dual object, represented on the host side by the `AceManager` class, and on the Policy Accelerator side by the `Ace` class. `Ace` objects in the Policy Accelerator contain the state information for each ACE.



To simplify common operations, the base `Ace` class provides basic `pass` and `drop` methods that you can call from any action function. The ASL also defines functions that you can call directly from the action part of an NCL rule, `action_pass` function and `action_drop` function, that serve the same purpose. Both the methods and the functions send packets through to the ACE's pass or drop targets.

You normally define subclasses of the `Ace` class to extend the functionality.

System ACEs The following ACEs are defined by the system, and are always available:

From packet interface 1	/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass
To packet interface 1	/nbhwpe0/ToInterface:nbhwpe0A/Interface

To host stack bound to interface 1	/nbhwpe0/ToStack:nbhwpe0A/Stack
From host stack bound to interface 1	/nbhwpe0/FromStack:nbhwpe0A/Stack/pass
From packet interface 2	/nbhwpe0/FromInterface:nbhwpe0B/Interface/pass
To packet interface 2	/nbhwpe0/ToInterface:nbhwpe0B/Interface
To host stack bound to interface 2	/nbhwpe0/ToStack:nbhwpe0A/Stack
From host stack bound to interface 2	/nbhwpe0/FromStack:nbhwpe0B/Stack/pass

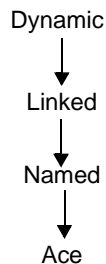
For more information on system ACEs, see Appendix C, “Policy Accelerator Name Space.”

The `Ace` class contains the following methods:

Method	Description
Ace Constructor	Instantiates the class.
drop Method	Marks the buffer to send to the ACE's drop target.
pass Method	Marks the buffer to send to the ACE's pass target.

Class Derivation

The `Ace` class is derived from the `Named` class.



From this class	The <code>Ace</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

From this class	The Ace class inherits
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.

Example

The following example is taken from the `BasicApp` demo application. The action file defines an ACE subclass, `NBBasicAce`, which contains an upcall object, and declares a method in that class that will create and send a message in an upcall:

```
class NBBasicAce : public Ace {
public:
    NBBasicAce (ModuleId id, char* name, Image* obj);
    void peekPacketUpcall (Buffer *buf);
    int packetCounter;
    nuint32 msg;
protected:
    Upcall peekPacketUpcallHandle;
};
```

The constructor for the ACE subclass creates the upcall object as well as the ACE object:

```
NBBasicAce::NBBasicAce (ModuleId id, char* iname, Image* iobj):
    Ace (id, iname, iobj),
    peekPacketUpcallHandle (id, this, "peekPacketUpcall")
{
    packetCounter = 0;
}
```

The initialization function calls the constructor to create the ACE object:

```
INITF init_actions(void* id, char* name, Image* obj)
{
    return new NBBasicAce(id, name, obj);
}
```

See Also

- [AceManager Class in Chapter 3, “Host API.”](#)
- [“The Object Framework” on page 49 of *Developing Applications Using the IX-API SDK*](#)

Ace Constructor

Creates an ACE object and registers it with the Policy Accelerator.

```
Ace (ModuleId id,
     char *   name,
     Image *  obj);
```

Argument	Description
id	The module identification number, assigned by the Resolver.
name	The dictionary name of the ACE. This must be the same as the dictionary name of the associated ACE manager object in the host module.
obj	The object containing the NCL classification rules and action code for this ACE.

Returns A reference to the newly created object.

Description You customize this constructor for your subclasses, then use the `new` operator in the body of the action code's initialization function to create an `Ace` object for newly downloaded NCL rules and action code. See "Initialization Function" on page 172.

drop Method

Marks the buffer to send to the ACE's drop target.

```
int drop (Buffer * b);
```

Argument	Description
b	The pointer to the buffer containing the packet to send.

Returns The constant `RULE_TOOK` (see "Custom Action Functions" on page 117).

Description	<p>Sending the packet to the ACE's drop target frees the <code>Buffer</code> object, unless the buffer's reference count is greater than 0, or unless the drop target has been bound to an ACE.</p> <p>By default, all packets that the application does not explicitly dispose of are sent to the drop target.</p>
See Also	<code>pass</code> Method, <code>action_drop</code> Function, <code>Buffer</code> Class

pass Method

Marks the buffer to send to the ACE's pass target.

```
int pass (Buffer * b);
```

Argument	Description
<code>b</code>	A pointer to the buffer containing the packet to send.

Returns	The constant <code>RULE_TOOK</code> (see Custom Action Functions on page 117).
Description	Sending the packet to the ACE's pass target results in the packet being either processed further by another ACE or transmitted to a network or host stack.
See Also	<code>drop</code> Method, <code>action_pass</code> Function

Action Functions

The ASL provides two functions that you can call directly in the action portion of a rule in NCL. You can also define your own action functions using the prescribed syntax.

Function	Description
action_drop Function	Routes a packet to the ACE's drop target.
action_pass Function	Routes a packet to the ACE's pass target.
Custom Action Functions	Define your own action functions to perform any action you want.

Example The following example is taken from the BasicApp demo application. The action function `action_all` calls a method in the ACE subclass that creates a message from the packet buffer and sends it to the host in an upcall:

```
ACTNF action_all (Buffer* buf, NBBasicAce* ace)
{
    ace->peekPacketUpcall (buf);
    return RULE_CONT;
}
```

See Also Chapter 7, “Acting on Packets in Your Action Code,” in *Developing Applications Using the IX-API SDK*

action_drop Function

Routes a packet to the ACE's drop target.

NCL Usage `rule rule-name { bool-expr } { action_drop () }`

Returns The constant `RULE_TOOK` (see “Custom Action Functions” on page 117).

Description The Policy Accelerator makes this action available to all ACEs for use when a packet should be explicitly routed to the ACE's drop target. You can name this action directly within the ACE's NCL source code in a rule association with a Boolean predicate.

See Also `drop` Method in `Ace` Class

action_pass Function

Routes a packet to the ACE's pass target.

NCL Usage `rule rule-name { bool-expr } { action_pass () }`

Returns The constant `RULE_TOOK` (see “Custom Action Functions” on page 117).

Description The Policy Accelerator makes this action available to all ACEs for use when a packet should be explicitly routed to the ACE's pass target. You can name this action directly within the ACE's NCL source code in a rule association with a Boolean predicate.

See Also `pass` Method in `Ace` Class

Custom Action Functions

You define additional action functions to be executed when a rule predicate is satisfied. The format for an action function is the same as that of the provided functions, `action_drop` and `action_pass`.

An action function that you define must conform to the following prototype:

```
(ACTNF) action_name (Buffer * buf,
                      Ace * ace,
                      args);
```



Argument	Description
buf	A pointer to the packet buffer to be processed. (Supplied automatically, not passed by the NCL rule.)
ace	A pointer to the ACE that is passing the buffer. (Supplied automatically, not passed by the NCL rule.)
args	<p>Define additional arguments as needed. The NCL rule must pass values of the correct type for all additional arguments defined for the action function it specifies.</p> <p>An action function can have a maximum of 31 arguments, and the call stack is limited to 87 arguments.</p> <p>NOTE: Due to limitations in the <code>gcc</code> compiler, it is recommended that you avoid the use of type <code>bool</code> arguments in action functions. Use <code>unsigned int</code> instead.</p>

Returns

An action function that you define must return one of the following constant values:

Return Code	Description
RULE_DONE	Return <code>RULE_DONE</code> to terminate processing of rules and actions within the context of the current ACE, for example, when a buffer has been sent to a target or stored for later processing. This value does not indicate whether the action has modified the packet.
RULE_TOOK	Return <code>RULE_TOOK</code> to terminate processing of this packet within this ACE if the action has not modified the packet starting location, size or contents, but has designated a target for the packet to flow through.

Return Code	Description
RULE_CONT	Return RULE_CONT if the action has only observed the buffer, and additional rules and actions within the context of the current ACE remain to be processed.
RULE_DEFER	Return RULE_DEFER if you want to modify a packet within a buffer but the buffer notes that the packet is currently busy elsewhere.



CAUTION: If your action function does not return one of these codes, you get a compiler warning. If you ignore this warning, your application is corrupted when the action function returns, and the corruption may not be detectable.

NCL Usage

To call your action function from a rule in NCL, use the following syntax:

```
rule rule-name { bool-expr } { action_name ( args... ) }
```

The `buf` and `ace` arguments are supplied automatically; you do not pass them in the rule. The rule must pass values of the correct types for any additional arguments you have defined for the action function.

See Also

- `action_drop` Function, `action_pass` Function
- “Rules and Actions” on page 406 in Chapter 6, “Network Classification Language.”
- Chapter 7, “Acting on Packets in Your Action Code,” in *Developing Applications Using the IX-API SDK*

Backlog Class

Use this class to monitor the various backlogs in packet flow for different ACEs. Backlogs occur when packets have finished one part of a processing cycle and are waiting for another part. For each ACE, there can be backlogs:

- between the classification of a packet and the execution of the required action on that packet
- between the time when a packet is transmitted and when its buffer memory is recovered

Use the `size` method to find out how many backlog counters are available, then the `names` method to find out which backlogs are counted. You can get either the estimated backlogs, using the `est` method, or the actual backlogs at the moment of making the `now` method call. It takes somewhat longer for the system to calculate the actual backlogs.

Because all of the retrieval methods are static, you do not need to instantiate this class.

The `Backlog` class contains the following methods:

Method	Description
<code>est</code> Method	Retrieves estimates of backlog counts.
<code>names</code> Method	Retrieves names of available backlog counts.
<code>now</code> Method	Retrieves current backlog counts.
<code>size</code> Method	Returns the number of available backlog counts.

The `Backlog` class is not derived from any other class.

est Method

Retrieves estimates of backlog counts.

```
static int est (int * ra,  
               int size);
```

Argument	Description
ra	[OUT] A pointer to an array of integers that, on return, contains the backlog count estimates.
size	The maximum number of pointers to write.

Returns The number of estimates written to the array.

Description This method makes estimates of the backlog counts, based on recent values. It places the estimated counts, up to the specified number, in the specified array. It returns the number of estimates actually written to the array.

The estimated values are the values most recently calculated in the normal processing path at the time when the object checked with the hardware to get index values. The first counter is an overall total.

This call is likely to be faster than the `now` method.

names Method

Retrieves names of available backlog counts.

```
static int names (char ** ra,  
                 int size);
```

Argument	Description
ra	A pointer to an array of strings that, on return, contains the backlog names.
size	The maximum number of pointers to write.

Returns The number of names written to the array.

Description This method places pointers to the names of the backlogs that can be counted, up to the specified number, in the specified array. It returns the number of pointers actually written to the array.

 The name describes which ACE the backlog is in and which process it belongs to (classification/action or disposition/recovery). The first counter is an overall total.

now Method

Retrieves the current backlog counts.

```
static int now (int * ra,  
               int size);
```

Argument	Description
ra	A pointer that an array of integers that, on return, contains to the current backlog counts.
size	The maximum number of counts to write.

Returns The number of pointers written to the array.

Description This method places the exact current backlog counts, up to the specified number, in the specified array. It returns the number of counts actually written to the array.

 The exact values are calculated during the call. The first counter is an overall total. This call is likely to take longer than the `est` method, but provides precise information.

size Method

Retrieves the number of available backlog counts.

```
static int names ();
```

Returns The number of available backlog counters, including the total.

Description This method returns the number of backlog counters that can be obtained, including the total counter, which is always first.

Buffer Class

Use this class to create and manipulate information in buffers. The network packet buffer is the basic unit of network data in the IX API. All data received from the network is received in buffers. All data transmitted on the network must be properly formatted into buffers. You allocate the buffer and declare a pointer to a new `Buffer` object using the `new` operator.

You can manually construct packets using this class. After creating a `Buffer` object using the `new` operator, you allocate space for packet data within the buffer using the `prepend` and `append` methods. When you first create the buffer object, the data offset points to the area where the IP datagram would start.

- To add data for any protocol header encapsulating the IP datagram (such as an ether header), use the `prepend` method.
- To add data for IP or any protocol encapsulated by IP, use the `append` method.

The `Buffer` class is not derived from any other class. It contains the following methods:

Method	Description
<code>append</code> Method	Adds space to the end of the packet, after the existing data.
<code>busy</code> Method	Indicates whether another action is using the buffer.
<code>decref</code> Method	Decrements the reference counter for the buffer by one; the counter keeps track of whether an action is using the buffer.
<code>headerBase</code> Method	Identifies the byte address of the first network header in the packet.
<code>headerType</code> Method	Gets a reference to the type of packet header.
<code>incref</code> Method	Increments the reference counter for the buffer by one; the counter keeps track of whether an action is using the buffer.
<code>interfaceNum</code> Method	Finds the source of the buffer.
<code>interfaceType</code> Method	Finds the type of the source of the buffer.
<code>new</code> Operator	Creates a new buffer.

Method	Description
<code>next</code> Method	Gets a reference to a field in the network packet buffer which applications can use to chain buffers together.
<code>packetPadHeadSize</code> Method	Gets the number of bytes of buffer space available at the beginning of the packet.
<code>packetPadTailSize</code> Method	Gets the number of bytes of buffer space available at the end of the packet.
<code>packetSize</code> Method	Gets the number of bytes in the network packet.
<code>prepend</code> Method	Adds space to the front of the packet, before the existing data.
<code>rxTime</code> Method	Identifies a <code>Time</code> object that corresponds to the timestamp taken when the packet was received from the network.
<code>takable</code> Method	Finds the current number of free network packet buffers.
<code>takable_clr</code> Method	Resets free buffer peak detector to the current number of free buffers.
<code>takable_max</code> Method	Finds the maximum number of free buffers since the peak value was reset.
<code>takable_min</code> Method	Finds the minimum number of free buffers since the peak value was reset.
<code>takable_set</code> Method	Updates the minimum and maximum values for the free buffer list.
<code>trim_head</code> Method	Discards data from the front of the packet.
<code>trim_tail</code> Method	Discards data from the end of the packet.
<code>txTime</code> Method	Gets a reference to a <code>Time</code> object that corresponds to the timestamp taken when the packet was transmitted to the network.

Example

This example shows how to manually construct packet buffers on the Policy Accelerator using data sent from the host in a downcall, as in the Loopback sample application.

The example copies packet data (11 bytes long, including both the MAC header and the IP datagram) that the accelerator module has received from a host downcall. The data is expected to be an IP datagram, so the function first allo-

cates space for the MAC header by prepending 14 bytes (the length of the header) to the buffer. It allocates space for the rest of the packet data (the IP datagram) using `append`. The function uses `memcpy` to copy the packet data into the appropriate location in the buffer.

```
void NBloopAce::packetDowncallHandler (Message *m)
{
    // get length of packet in bytes
    int ll = m->len1 ();
    // get packet data
    char *ml = m->msg1 ();

    if (ml && ll > 14) { // valid message contents
        Buffer *buf = new Buffer;
        if (buf) { // got a buffer
            // allocate space in buffer for ethernet
            // MAC header(14 bytes) and IP datagram
            // The packet data is ll bytes long,
            // including the header and the datagram
            char *ps = buf->prepend (14);
            char *pe = buf->append (ll - 14);
            if (ps && pe) { // prepend and append went OK
                // copy packet data into Buffer
                memcpy (ps, ml, ll);
                pass (buf);
                buf->decref ();
            }
            else { // prepend or append problem - dump buffer
                printf ("[ERROR] Got downcall(1)\n");
                buf->decref ();
                buf = 0;
            }
        }
        else
            printf ("[ERROR] Got downcall(2)\n");
    }
    else
        printf ("[ERROR] Got downcall(3) ml = %p, ll = %d\n",
            ml, ll);
    // done with the message
    m->done();
} //packetDowncallHandler
```

append Method

Adds space to the end of the packet, after the existing data.

```
char * append (size_t bytes);
```

Argument	Description
bytes	Number of bytes to append to the packet.

Returns	Returns a pointer to the newly appended storage. If there is insufficient padding space at the end of the packet, returns <code>NULL</code> and does not add any of the data.
Description	Prepares a packet to receive additional data after the existing data. For example, if you want to add an IP datagram payload to a packet containing only an IP header, you can append it to the packet. Bookkeeping values are updated to extend the packet space appropriately.
See Also	<code>prepend Method</code>

busy Method

Indicates whether a previous action is using the buffer.

```
int busy ();
```

Returns	<code>TRUE</code> if the action should not change the contents of the buffer. <code>FALSE</code> if applied to a newly allocated buffer.
Description	<p>Examines the reference counter, returning <code>TRUE</code> if the reference count indicates that the currently executing action function should not change the contents of the buffer. If <code>busy</code> is applied to a newly allocated buffer, it returns <code>FALSE</code>. After <code>incref</code> has been applied to such a packet, <code>busy</code> returns <code>TRUE</code>.</p> <p>Use this method in an action to determine whether a previous action has (directly or indirectly) applied the <code>incref</code> method to the buffer. This is useful when one action needs to maintain use of a packet temporarily (to send a copy</p>

to the host using an upcall, for example), and another needs to modify the packet; the second action can check if the buffer is busy, and if it is, return `RULE_RERUN`. (See “Custom Action Functions” on page 117)

See Also `decref Method`, `incref Method`

decref Method

Decrements the reference counter for the buffer by one.

```
void decref ();
```

Returns Nothing.

Description The reference counter keeps track of whether an action is using the buffer. The `decref` call undoes the effect of a corresponding `incref` call. When all `incref` calls have been undone, the next call to `decref` allows the buffer to move on to its next processing stage. Deferred actions or transfers can be executed. If no deferred actions are waiting, the memory containing the buffer is freed. Because the buffer can be freed immediately, never use a buffer pointer after `decref` has returned.

If `decref` is applied to a newly allocated buffer, the buffer is freed.

Example The following code fragment retains a packet buffer until another arrives. It then operates on both buffers in an unspecified way and sends the older buffer onward while retaining the newer buffer for later use.

The call to `incref` prevents the new packet buffer from being turned over to the normal modification routines and freed. This routine stores a pointer to the buffer into a private data object. The call to `decref` means that this code no longer requires the data in the previously saved packet buffer.

```
Buffer *old = ace->saved;
if (old != NULL) {
    /* act on "old" and "buf" */
    old->decref (); /* recycle "old" */
}
buf->incref (); /* prevent recycling "buf" */
ace->saved = buf; /* save it for later */
```

See Also `busy Method`, `incref Method`

headerBase Method

Identifies the byte address of the first network header in the packet.

```
char * headerBase ();
```

Returns Returns a byte address.

See Also headerType Method, packetSize Method

headerType Method

Gets a reference to the type of packet header at the header offset location.

```
uint32 & headerType ();
```

Returns A reference to an integer encoding the type of packet header.

Description Only one header type code, zero, is currently defined. A header type of zero specifies that the packet is an Ethernet packet, with decorations similar to those that the Policy Accelerator MACs (media access controllers) attached to the packet buffer. The decorations (receive status, receive timestamp, transmit status, and transmit timestamp) are placed in the buffer along with the packet.

When using the `base.proto` field to construct `headerType`, you must specify `0x0001` for Ethernet packets and `0x0021` for IP datagrams.

See Also headerBase Method

incred Method

Increments the reference counter for the buffer by one.

```
void incref ();
```

Returns Nothing.

Description The reference counter keeps track of whether an action is using the buffer. This method increments the reference counter for the buffer to prevent it from being deleted, transferred to another ACE, transmitted to a network, moved to a host stack, or modified by any action code that checks whether the buffer is busy.

A call to `incred` prevents a packet buffer from being turned over to the normal modification routines and freed until a corresponding `decref` call is made. Use this facility with care to make sure the buffers are properly freed when you are finished with them.

See Also `busy Method`, `decref Method`

interfaceNum Method

Finds the source of the buffer.

```
uint16& interfaceNum ();
```

Returns A reference to the interface or ACE from which the buffer was received.

Description This method returns the enumerated value, or ACE tag, of the source of the buffer, which can vary from session to session. Use the `NBApp1` object's `getTag` method in the host module to find the value corresponding to a binding in your application. For example:

```
uint16 i1 = getTag
    ("\\nbhwpe0\\FromInterface:nbhwpe0A\\Interface");
```

The host module must transfer these values to the accelerator module in a downcall. In this case, you can compare the value returned by the `interfaceNum` method to the `i1` value transferred from the host module to find out if the buffer was received from interface A on `nbhwpe0`, the first installed Policy Accelerator.

See Also

- `interfaceType` Method
- `NBApp1::getTag` Method and `AceManager::getTag` Method in Chapter 3, “Host API.”
- Appendix C, “Policy Accelerator Name Space.”

interfaceType Method

Finds the source type of the buffer.

```
uint16& interfaceType ();
```

Returns A reference to the interface or ACE from which the buffer was received.

Description This method returns the interface type of the source of the buffer. Interface type numbers are maintained by the Internet Assigned Numbers Authority (IANA). For the latest values, refer to the following Web site:

```
ftp://ftp.isi.edu/mib/ianaiftype.mib
```

See Also `interfaceNum Method`

new Operator

Creates a new buffer.

```
Buffer * b = new buffer ();
```

```
Buffer * b = new buffer (*b);
```

Returns A reference to the newly created buffer instance. If no buffers are available, returns a `NULL` pointer.

Description Use the standard C++ `new` operator to dynamically allocate buffers from the pool of appropriately aligned and addressable memory.

- If you pass no argument, the allocated packet buffer contains a zero-length packet starting at a word-aligned location with a moderately-sized head pad area.
- If you pass a buffer pointer, the new packet buffer is allocated and initialized to contain a copy of the packet from that buffer.

When the buffer is no longer required, use the method `Buffer::decref()` to delete it. Do not use the `delete` operator.



NOTE: If you allocate a buffer with the `new` operator, fill it in and send it on (using a `Target::take` call, for example, or the `Ace::pass` or `Ace::drop` methods), you must call the `decref()` method, or the packet you constructed will never go anywhere.

Example

```
class CloneAce : public Ace {
public:
    Target clone_target;
    CloneAce (ModuleId id, char *name, int tag)
        : Ace (id, name, tag)
        , clone_target (this, "clones")
    };
};
```

This action function copies the buffer and sends the duplicate to the clone target. The `decref` tells the support code to start processing the newly created clone. If all network packet buffers are in use, `new` returns `NULL`. Be sure to check for this.

```
action_clone (Buffer *buf, CloneAce *ace)
{
    Buffer *clone = new Buffer (*buf);
    if (clone) { //make sure buffer was allocated
        ace->clone_target.take (clone);
        clone->decref ();
    }
    return ace->pass (buf);
}
```

See Also

`decref Method`, `incref Method`

next Method

Locates a field in the network packet buffer that applications can use to chain buffers together.

```
Buffer &* next ();
```

Returns

A reference to a pointer to another buffer.

Description

The Policy Accelerator does not modify this field as buffers are passed among the ACEs in a Policy Accelerator. An application can store a pointer to another buffer in this field in order to maintain a sequence of buffers.

packetPadHeadSize Method

Gets the number of bytes of buffer space available for expansion at the beginning of the packet.

```
size_t packetPadHeadSize ();
```

Returns A number of bytes.

See Also [headerBase Method](#), [headerType Method](#), [packetPadTailSize Method](#), [packetSize Method](#), [prepend Method](#), [trim_head Method](#), [trim_tail Method](#)

packetPadTailSize Method

Gets the number of bytes of buffer space available for expansion at the end of the packet.

```
size_t packetPadTailSize ();
```

Returns A number of bytes.

See Also [append Method](#), [packetPadHeadSize Method](#), [packetSize Method](#), [trim_head Method](#), [trim_tail Method](#)

packetSize Method

Gets the number of bytes in the network packet.

```
uint32 & packetSize ();
```

Returns A number of bytes.

See Also [headerBase Method](#), [headerType Method](#), [packetPadHeadSize Method](#), [packetPadTailSize Method](#), [trim_head Method](#), [trim_tail Method](#)

prepend Method

Adds space to the beginning of the packet, before the existing data.

```
char * prepend (size_t bytes);
```

Argument	Description
bytes	Number of bytes to prepend to the packet.

Returns	A pointer to the newly prepended storage. If there is insufficient padding space at the head of the packet, returns <code>NULL</code> and does not add any of the data.
Description	Prepares a packet to receive additional data before the existing data. For example, use this method to add a new Ethernet header to an IP fragment. Bookkeeping values are updated to extend the packet space appropriately.
See Also	<code>append Method</code> , <code>headerBase Method</code> , <code>headerType Method</code> , <code>headerType Method</code> , <code>packetPadHeadSize Method</code> , <code>packetPadTailSize Method</code> , <code>packetSize Method</code>

rxTime Method

Identifies a `Time` object that corresponds to the timestamp taken when the packet was **received** from the network.

```
Time & rxTime ();
```

Returns	A reference to a <code>Time</code> object.
Description	The contents of this field are undefined if the packet was not received from a network interface.
See Also	<code>txTime Method</code>

takable Method

Finds the current number of free network packet buffers.

```
static int takable ()
```

Returns A number of free packet buffers.

Description Use this method to monitor the packet buffer memory usage. You can use this with the `Backlog` methods to determine whether enough memory is allocated for packet buffers on the Policy Accelerator.

See Also `Backlog Class`

takable_clr Method

Resets free buffer peak detector to the current number of free buffers.

```
static void takable_clr ()
```

Returns Nothing.

Description Use this method to monitor the packet buffer memory usage. Use this with the `takable_max` and `takable_min` methods to determine whether enough memory is allocated for packet buffers on the Policy Accelerator.

See Also `takable Method`, `takable_max Method`, `takable_min Method`

takable_max Method

Finds the maximum number of free buffers since the peak value was reset.

```
static int takable_max ()
```

Returns A number of free buffers.

Description Use this method to monitor the packet buffer memory usage. You can use this with the `takable_clr` method to determine whether enough memory is allocated for packet buffers on the Policy Accelerator.

See Also `takable Method`, `takable_clr Method`, `takable_min Method`

takable_min Method

Finds the minimum number of free buffers since the peak value was reset.

```
static int takable_min ()
```

Returns A number of free buffers.

Description Use this method to monitor the packet buffer memory usage. You can use this with the `takable_clr` method to determine whether enough memory is allocated for packet buffers on the Policy Accelerator.

See Also `takable Method`, `takable_clr Method`, `takable_max Method`

takable_set Method

Updates the minimum and maximum values for the free buffer list.

```
static void takable_set ()
```

Returns Nothing.

Description If you manipulate the size of the free packet buffer list directly by adjusting the list pointer, you must use this method to update the minimum and maximum values before you can continue to monitor free buffers.

See Also takable Method, takable_clr Method, takable_max Method, takable_max Method

trim_head Method

Discards data from the beginning of the packet.

```
char * trim_head (size_t bytes);
```

Argument	Description
bytes	Number of bytes to delete.

Returns The starting address of the trimmed packet, or NULL if no data is left.

Description If the packet is large enough to be trimmed by the specified amount, the book-keeping values are updated to reflect the shorter packet starting in a new place, and the method returns the address of the start of the remaining packet. If the existing packet is not large enough, the packet is not trimmed, and the method returns NULL.

See Also packetSize Method, trim_tail Method

trim_tail Method

Discards data from the end of the packet.

```
char * trim_tail (size_t bytes);
```

Argument	Description
bytes	Number of bytes to delete.

Returns	The starting address of the trimmed packet, or <code>NULL</code> if no data is left.
Description	If the packet is large enough to be trimmed by the specified amount, the book-keeping values are updated to reflect the shorter packet, and the method returns the address of the start of the remaining packet. If the existing packet is not large enough, the packet is not trimmed, and the method returns <code>NULL</code> .
See Also	<code>packetSize Method</code> , <code>trim_head Method</code>

txTime Method

Gets a reference to a `Time` object corresponding to the timestamp taken when the packet was most recently **transmitted** to the network.

```
Time & txTime ();
```

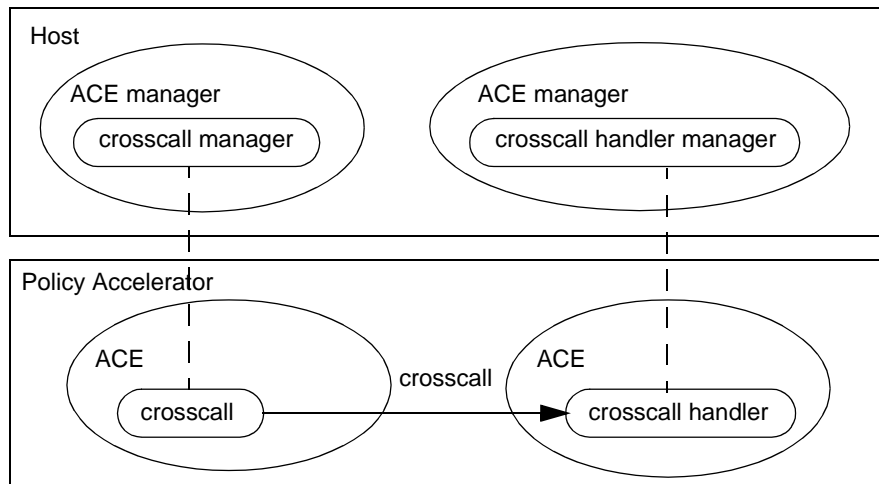
Returns	A reference to a <code>Time</code> object.
Description	If the packet has not been presented to a network interface for transmission, the content of this time value is undefined. If a TX Timestamp is not generated, the TX timestamp is zero.
See Also	<code>rxTime Method</code>

Crosscall Class

Use this class to send messages from one ACE to another in the same or another Policy Accelerator.

The `Crosscall` class contains the information that the Policy Accelerator requires to deliver crosscalls from one ACE in a Policy Accelerator to the proper service function in another ACE in any Policy Accelerator.

Each `Crosscall` object is associated with exactly one `CrosscallManager` object on the host. After creating a `Crosscall` object, you must create its associated manager object on the host. The paired objects are associated by having the same dictionary name. The manager objects keep track of the association between a call and its handler; see “`CrosscallManager` Class” on page 52.



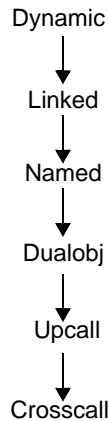
In response to the `Crosscall` object's `call` method, the host executes the service function specified in the associated `CrosscallHandler` object, passing it the specified message. To associate a `Crosscall` object with a `CrosscallHandler` object in another ACE, use the `link` method of the application's `NBappl` object. See “`NBappl` Class” on page 68. Any number of `Crosscall` objects can be linked to the same `CrosscallHandler` object.

The `Crosscall` class contains the following methods:

Method	Description
Crosscall Constructor	Instantiates the class.
call Method	Sends a message from one ACE in the Policy Accelerator to another ACE in the same or another Policy Accelerator.

Class Derivation

The `Crosscall` class is derived from the `Upcall` class.



From this class	The <code>Crosscall</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The <code>ace</code> method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerator.
Upcall	Nearly all of its abilities.

Example

In the following example, the ACE subclass, `MyAce`, is defined to contain both a crosscall object and a crosscall handler object, as well as methods to send and to receive a message using a crosscall:

```
class MyAce : public Ace {
public:
    MyAce (ModuleId id, char* name, Image* obj);
    ~MyAce ();
    void ToOtherAce ( ); // Crosscall sending method
protected:
    // CrosscallHandler callback
    void FromOtherAce (Message *p);
    // Define counter and snapshot for message
    int packetCounter;
    uint32 countSnapshot;
    Crosscall MyCrosscall; // Crosscall handle
    CrosscallHandler MyXcallHandler; // CrosscallHandler handle
};
```

Before you can send or receive a message, you must create two ACEs of this type, and link the crosscall object in one with the crosscall handler object in the other by calling the `link` method of the ACE manager object in the host module.

The constructor for the ACE object creates the crosscall and crosscall handler objects:

```
MyAce::MyAce (ModuleId id, char* name, Image* obj)
    : Ace (id, name, obj)
    , packetCounter = 0 // Init counter
    // create Crosscall & CrosscallHandler objects
    , MyCrosscall (id, this, "MyCrosscall")
    , MyXcallHandler (id, this, "MyCrosscallHandler",
                     (DcallMFp)&FromOtherAce)
    {}
```

An ACE method creates a message (preserving the byte order of numeric data) and sends it in the crosscall.

```
void MyAce::ToOtherAce (void) {
    // create message with known byte order
    countSnapshot = htonl(packetCounter); //take snapshot
    MessageBlock mb ((char *) &countSnapshot,
                     sizeof (countSnapshot));
    Message m (mb); //create message
    MyCrosscall->call (&m);
}
```

- See Also
- [CrosscallHandler Class](#), [Message Class](#), [MessageBlock Class](#)
 - [CrosscallManager Class](#) and `NBApp1::link` Method in Chapter 3, “Host API.”
 - “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*

Crosscall Constructor

Creates a `Crosscall` object for an ACE.

```
Crosscall (ModuleId id,
           Ace * ace,
           char * name);
```

Argument	Description
<code>id</code>	ACE identifier assigned by the Resolver.
<code>ace</code>	Pointer to the ACE object in the Policy Accelerator to which this crosscall belongs.
<code>name</code>	The crosscall's dictionary name. This must be the same as the dictionary name of the associated <code>CrosscallManager</code> object on the host.

Returns A reference to the newly created object.

Description The host module must create a corresponding `CrosscallManager` object using the same dictionary name. You cannot use the crosscall to send a message until the host calls the application object's `link` method to associate this crosscall with a crosscall handler.

call Method

Sends a message from this ACE to another ACE in the same Policy Accelerator or in a different Policy Accelerator attached to the same host.

```
int call (Message * m);
```

Argument	Description
m	Pointer to the message to be sent.

Returns 0 on success. If the crosscall is not yet initialized by the Resolver or if the cross-call channels are clogged, returns a negative error code.

Description For information on creating the message argument, see “Message Class” on page 180 and “MessageBlock Class” on page 184.



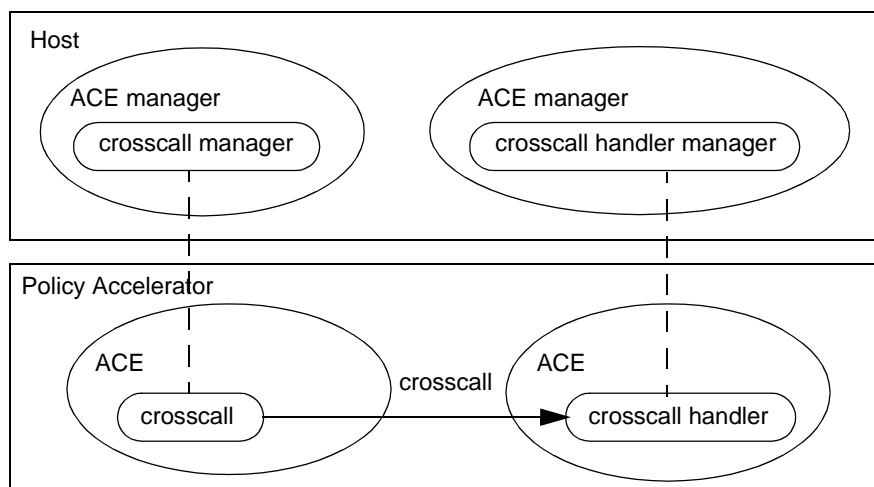
NOTE: Do not delete the message pointer after sending the call. Because the call is queued for asynchronous handling, the message could be deleted before the call is processed. By default, messages are automatically deleted when the call is complete. See “MessageBlock Class” on page 184 for information on alternative handling.

CrosscallHandler Class

Use this class to direct messages from another ACE to a service function.

The `CrosscallHandler` class contains the information that the Policy Accelerator requires to direct incoming crosscalls from another ACE in the same or another Policy Accelerator to the proper service function in the owning ACE.

Each `CrosscallHandler` object is associated with exactly one `CrosscallHandlerManager` object on the host. After creating a `CrosscallHandler` object, you must create its associated manager object on the host. The paired objects are associated by having the same dictionary name. The manager objects keep track of the association between a call and its handler; see “`CrosscallHandlerManager` Class” on page 48.



In response to a `Crosscall` object's `call` method, the host executes the service function specified in the associated `CrosscallHandler` object, passing it the specified message. To associate a `Crosscall` object with a `CrosscallHandler` object in another ACE, use the `link` method of the application's `NBappl` object. See “`NBappl` Class” on page 68. Any number of `Crosscall` objects can be linked to the same `CrosscallHandler` object.

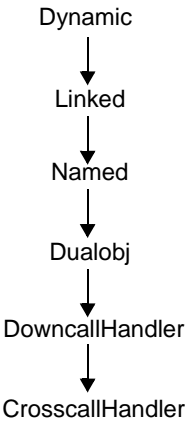
You are responsible for deallocating the local memory associated with the received message when it is no longer used, using the `delete` operator.

The CrosscallHandler class contains the following methods:

Method	Description
CrosscallHandler Constructor	Instantiates the class.
direct Method	Specifies the service function to use for handling crosscall messages.

Class
Derivation

The CrosscallHandler class is derived from the DowncallHandler class.



From this class	The CrosscallHandler class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The ace method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerator.
DowncallHandler	Nearly all of its abilities.

Example

In the following example, the ACE subclass, `MyAce`, is defined to contain both a crosscall object and a crosscall handler object, as well as methods to send and to receive a message using a crosscall:

```
class MyAce : public Ace {
public:
    MyAce (ModuleId id, char* name, Image* obj);
    ~MyAce ();
    void ToOtherAce ( ); // Crosscall sending method
protected:
    void FromOtherAce (Message *p); // CrosscallHandler callback
    int packetCounter; // Define counter and snap for msg
    uint32 countSnapshot;
    Crosscall MyCrosscall; // Crosscall handle
    CrosscallHandler MyXcallHandler; // CrosscallHandler handle
};
```

Before you can send or receive a message, you must create two ACEs of this type, and link the crosscall object in one with the crosscall handler object in the other by calling the `link` method of the ACE manager object in the host module.

The constructor for the ACE object creates the crosscall and crosscall handler objects:

```
MyAce::MyAce (ModuleId id, char* name, Image* obj)
    : Ace (id, name, obj)
    , packetCounter = 0 // Init counter
// create Crosscall & CrosscallHandler objects
    , MyCrosscall (id, this, "MyCrosscall")
    , MyXcallHandler (id, this, "MyCrosscallHandler",
                      (DcallMFp)&FromOtherAce)
{}

```

The callback method that handles the crosscall is defined in the ACE class. It accesses the message content, restores the byte order of numeric data, and acts on the message.

```
void MyAce::FromOtherAce (Message *p)
{
    // unpack message, restoring byte order
    nuint32* p = (nuint32*) (m->msg1 ());
    currcount = ntohl (*p);
    // act on message
    ...
}
```

See Also

■ [Crosscall Class](#), [Message Class](#), [MessageBlock Class](#)

- CrosscallHandlerManager Class and NBApp1::link Method in Chapter 3, “Host API.”
- “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*

CrosscallHandler Constructor

Creates a CrosscallHandler object, and optionally binds it to a service function callback.

```
CrosscallHandler (ModuleId id,
                  Ace * ace,
                  char * name);

CrosscallHandler (ModuleId id,
                  Ace * ace,
                  char * name,
                  DcallMFp func);
```

Argument	Description
id	ACE identifier assigned by the Resolver.
ace	Pointer to the Ace object in the Policy Accelerator to which this crosscall handler belongs.
name	The crosscall handler’s dictionary name. This must be the same as the dictionary name of the associated CrosscallHandlerManager object on the host.
func	Service function to be executed on the received message. (Optional)

Returns

A reference to the newly created object.

Description

You can specify the service function callback when you create the object, or later using the `direct` method. The service function must unpack the message, restoring the byte order for numeric data.

See Also

- `direct` Method
- “Communication among ACEs” on page 106 of *Developing Applications Using the IX-API SDK*

- “Byte Order and Intermodule Communication” on page 12 in Chapter 2, “System Types and Methods.”

direct Method

Specifies the service function to use to handle incoming messages.

```
void direct (DcallMFp func);
```

Argument	Description
func	Service function to be executed by the crosscall.

Returns Nothing.

Description The service function callback you specify must be a member function of a subclass of the `Ace` class.

Crosscall Callbacks You must supply a callback that conforms to the following prototype:

```
class MyAce : public Ace {
    void my_handler(Message *m);
    ...}
```

When the callback has finished processing the message, it is responsible for releasing any message data block memory that was allocated locally. For a crosscall, use the `Message` object's `done` method to do this. Do not delete the message pointer itself: the Policy Accelerator system software is responsible for it.

Example The following example defines and sets a callback using the `direct` method:

```
class CustomAce : public Ace {
public:
    void my_crosscall_handler(Message *m)
    {
        // do something with the content of "m"
        m->done (); // when finished with message's data block(s).
    }

    CrosscallHandler my_crosscall_handler;
```

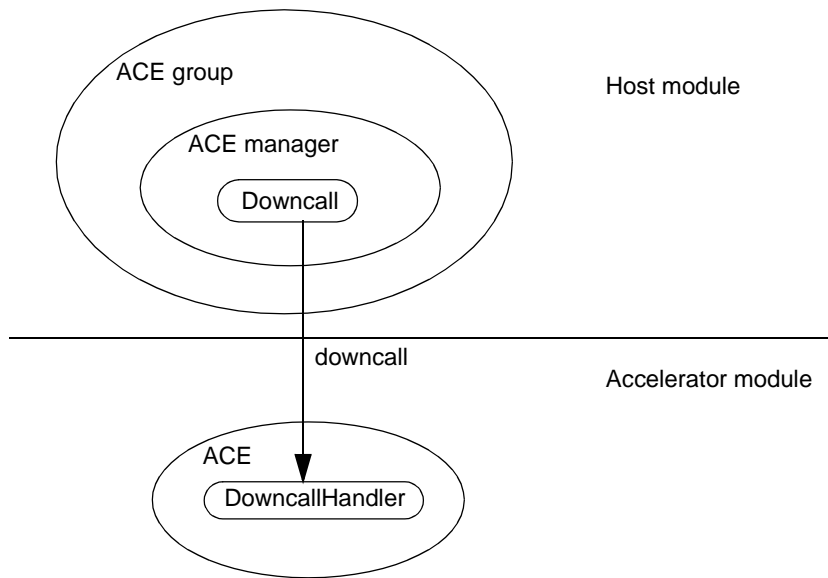
```
CustomAce(ModuleId id, char *name, int tag)
    : Ace(id, name, tag)
    , my_crosscall_handler(id, this, "my_crosscall")
{
    my_crosscall_handler.direct (DCALLMFP(my_crosscall_handler));
}
```

DowncallHandler Class

Use this class to direct downcalls from the host to the Policy Accelerator.

The `DowncallHandler` class contains the information that the Policy Accelerator requires to direct incoming downcalls from the host to the proper service function in the Policy Accelerator.

Each `DowncallHandler` object is associated with one `Downcall` object with the same dictionary name in the host module. In response to the `Downcall` object's `call` method, the Policy Accelerator executes the service function specified in the associated `DowncallHandler` object, passing it the specified message.



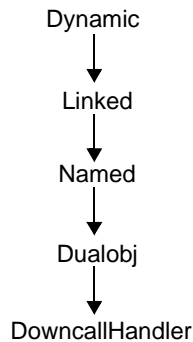
When a message is passed to the Policy Accelerator by a downcall, memory for the message is allocated on the Policy Accelerator. The Policy Accelerator is responsible for allocating and deallocating this memory. Do not attempt to free or delete the message pointer in the action code, or to store it for later handling.

The `DowncallHandler` class contains the following methods:

Method	Description
<code>DowncallHandler</code> Constructor	Instantiates the class.
<code>direct</code> Method	Specifies the service function to use for handling downcall messages.

Class Derivation

The `DowncallHandler` class is derived from the `Dualobj` class.



From this class	The <code>DowncallHandler</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The <code>ace</code> method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerators.

Example

The following example is taken from the `TwoAceApp` demo application. The ACE subclass `NBAceOne` contains a downcall handler object and a callback method:

```

class NBAceOne : public Ace {
public:

```

```

    NBaseOne (ModuleId id, char* name, Image* obj);
// DowncallHandler
    void setReportPeriod (Message* m);
    DowncallHandler* setReportPeriodDowncallHandle;
    int packetCount;
    int reportPeriod;
};

```

The constructor and destructor for the ACE subclass create and delete the downcall handler object:

```

NBaseOne::NBaseOne (ModuleId id, char* name, Image* obj):
    Ace (id, name, obj)
{
    setReportPeriodDowncallHandle =
        new DowncallHandler (id, this,
                              "setReportPeriodDowncall",
                              DCALLMFP (setReportPeriod));

    packetCount = 0;
    reportPeriod = 1000;
}

NBaseOneMgr::~~NBaseOneMgr ()
{
    delete setReportPeriodDowncallHandle;
}

```

The downcall handler callback unpacks the message (restoring the byte order) and uses the value to set a variable:

```

NBaseOne::setReportPeriod (Message* m)
{
    uint32* p = (uint32*) (m->msg1 ());
    reportPeriod = ntohl (*p);
    m->done();
}

```

See Also

- Downcall Class, Message Class, MessageBlock Class in Chapter 3, “Host API.”
- “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*
- “Byte Order and Intermodule Communication” on page 12 in Chapter 2, “System Types and Methods.”

DowncallHandler Constructor

Creates a `DowncallHandler` object and optionally binds it to a service function.

```
DowncallHandler (ModuleId id,
                 Ace * ace,
                 char * name);

DowncallHandler (ModuleId id,
                 Ace * ace,
                 char * name,
                 DcallMFp func);
```

Argument	Description
id	ACE identifier assigned by the Resolver.
ace	Pointer to the ACE object in the Policy Accelerator to which this downcall handler belongs.
name	The downcall handler's dictionary name. This must be the same as the dictionary name of the associated <code>Downcall</code> object in the host module.
func	Service function callback to be executed by the associated downcall. (Optional)

ReturnsA reference to the newly created object.

DescriptionYou can specify the service function callback when you create the object, or later using the `direct` method. The service function must unpack the message, restoring the byte order for numeric data.

See Also

- `direct` Method
- `Downcall` Class, `Message` Class, `MessageBlock` Class in Chapter 3, “Host API.”
- “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*
- “Byte Order and Intermodule Communication” on page 12 in Chapter 2, “System Types and Methods.”

direct Method

Specifies the service function for the downcall.

```
void direct (DcallMFp func);
```

Argument	Description
func	Service function to be executed by the downcall.

Returns Nothing.

Description The service function callback you specify must be a member function of a class derived from the `Ace` class.

Downcall Callbacks You must supply a callback that conforms to the following prototype:

```
class MyAce : public Ace {
    void my_handler (Message *m);
    ...}
```

When the callback has finished processing the message, it is responsible for releasing any message data block memory that was allocated locally. For a downcall, use the `Message` object's `done` method to do this. Do not delete the message pointer itself, as the Policy Accelerator system software is responsible for it.

Example The following example defines and sets a callback using the `direct` method:

```
class CustomAce : public Ace {
public:
    void my_downcall_handler (Message *m)
    {
        // do something with the content of "m"
        m->done (); // when finished with message's data block(s)
    }
private:
    DowncallHandler my_downcall_handler;
}

CustomAce (ModuleId id, char *name, int tag)
: Ace (id, name, tag)
, my_downcall_handler (id, this, "my_downcall")
{
    my_downcall_handler.direct (DCALLMFp(my_downcall_handler));
}
```

Dualobj Class

The system uses this class to determine which objects in the Policy Accelerator correspond with which objects in the host.

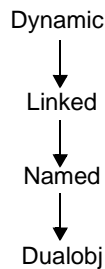
All paired classes in the Policy Accelerator are derived from the `Dualobj` (dual object) class. These are classes that use data resident in both the host and the Policy Accelerator. A dual object resident in the Policy Accelerator is connected to a corresponding object in the host that has the same dictionary name. The `Dualobj` class contains the information necessary for the Policy Accelerator to identify which particular object in the Policy Accelerator corresponds with which particular object in the host.

The `Dualobj` class contains the following methods:

Method	Description
<code>Dualobj</code> Constructor	Instantiates the class.
<code>ace</code> Method	Finds the ACE that owns this dual object.

Class Derivation

The `Dualobj` class is derived from the `Named` class:



From this class	The <code>Dualobj</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.

See Also “Object Pairing on the Host and Policy Accelerator” on page 27 in Chapter 3, “Host API.”

Dualobj Constructor

Creates a dual object resident in the Policy Accelerator which will be connected to its corresponding object in the host.

```
Dualobj (ModuleId id,
        Ace * ace,
        char * name);
```

Argument	Description
id	ACE Identifier assigned by the Resolver.
ace	Pointer to the <code>Ace</code> object in the Policy Accelerator.
name	The object's dictionary name. This must be the same as the dictionary name of the corresponding object in the host module.

Returns A reference to the newly created object.

Description Instantiates the Policy Accelerator-resident `Dualobj` object, noting the information about the object that enables a connection to be established with the corresponding object resident in the host.

ace Method

Finds the ACE that owns this dual object.

```
Ace * ace () const;
```

Returns A pointer to the `Ace` object in the Policy Accelerator for the ACE that owns this dual object.

Dynamic Class

Use this class as a base class to provide fast pool allocations for your objects. The `Dynamic` class has no storage requirements and no virtual functions. Including `Dynamic` in your object class hierarchy does not change the size or layout of your objects, only how they are allocated.

This class is responsible for overloading the `new` and `delete` operators, redirecting the memory allocation to use a number of `Tagged` pools managed by the Policy Accelerator. All descendants of `Dynamic` share the same set of `Tagged` pools. Each pool handles a specific range of object sizes, and objects of similar sizes share the same `Tagged` pool.

You can base your object hierarchy directly on the `Dynamic` class or one of the other classes in the Policy Accelerator that is derived from `Dynamic`, inheriting the allocation mechanism used inside the library for its own objects.

The `Dynamic` class is not derived from any other class.

The `Dynamic` class contains the following operators:

Operator	Description
<code>delete</code> Operator	Releases and redirects released objects into the appropriate <code>Tagged</code> pool.
<code>new</code> Operator	Acquires memory from an appropriate <code>Tagged</code> pool, based on the object's size.

delete Operator

Releases and redirects released objects into the appropriate Tagged pool.

```
delete dp;
```

Returns Nothing.

Description The `delete` operator handles releasing all objects of all classes derived from the `Dynamic` class, as long as they do not redefine the `delete` operator themselves. The operator redirects the released object into the appropriate Tagged pool.

See Also Tagged Class

new Operator

Redirects allocation requests into the appropriate Tagged pool.

```
Dynamic * dp = new Dynamic;
```

Returns A reference to the newly created object.

Description The `new` operator handles allocation of all objects of all classes derived from the `Dynamic` class, as long as they do not redefine the `new` operator themselves. The operator redirects the allocation request into the appropriate Tagged pool.

See Also Tagged Class

Element Class

This class is the base class from which set element subclasses (`Elt_setname`) are derived.

You do not use this class directly. Instead, you use the methods defined in this class to create and manipulate elements that are descended from the `Elt_setname` classes. For detailed descriptions of the methods, see “`Elt_setname` Class” on page 160.

The `Element` class is not derived from any other class.

See Also

- “Set Elements” on page 102
- `Elt_setname` Class, `Set_setname` Class
- Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*

Elt_setname Class

For each named set defined in Network Classification Language (NCL), the NCL compiler produces an adjusted Element subclass called `Elt_setname`, using the name of the set. You normally create at least one further subclass to define the data portion of the element.

Elements contain an expiration time, after which the Policy Accelerator executes a callback function that you supply. Basic methods allow you to set and cancel the expiration time and specify the expiration callback.



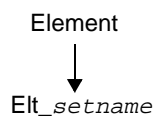
NOTE: Use the `Event` class to schedule events that are not associated with data. This is much more efficient than creating a set element for the sole purpose of using its `expire` method.

The `Elt_setname` class contains the following methods:

Method	Description
<code>Elt_setname</code> Constructor	Instantiates the class. Used by the <code>new</code> operator.
<code>Elt_setname</code> Destructor	Destroys an <code>Elt_setname</code> object. Used by the <code>delete</code> operator.
<code>cancel</code> Method	Cancels an element's expiration time.
<code>delete</code> Operator	Recycles the memory being used to store a set element, and cleans up references to it.
<code>expire</code> Method	Sets an element's expiration time and callback function.
<code>new</code> Operator	Allocates an appropriately aligned <code>Elt_setname</code> object using the constructor method, adding space for the correct number of key values.

Class Derivation

The `Elt_setname` class is derived from the `Element` class, inheriting all public methods.



From this class	The Elt_setname class inherits
Element	All public methods

Example

The following example is taken from the IPPairs demo application. The header file that defines the element subclass, Elt_pair, is automatically generated from the NCL file. It can be modified to add application-specific details. The skeleton definition for the subclass is as follows:

```
class Elt_pair : public Element {
public:
    inline Elt_pair (nuint32 k1, nuint32 k2,
                    nuint32 k3, nuint32 k4) {

        key_[0] = k1;
        key_[1] = k2;
        key_[2] = k3;
        key_[3] = k4;
    }
    nuint32 key_[4];
};
```

See Also

- “Set Elements” on page 102
- Set_setname Class, Search Class
- Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*

Elt_setname Constructor

Creates an `Elt_setname` object.

```
Elt_setname (nuint32 k1,...nuint32 kn);
```

Argument	Description
k1	The key values to match. You must pass the number of key values specified by <code>nkeys</code> in the NCL set definition.

Returns A reference to the newly created object.

Description This constructor sets up the key information for the set element.

Note that this method does not have a variable number of arguments. You must pass the number of arguments defined for the specific set by the `nkeys` argument in the NCL `set` statement that declared and defined it.

Key values are network-ordered words. The compiler converts from host to network order when setting key values during initialization. See “Byte Order Issues” in Chapter 2, “System Types and Methods.”

See Also `Set_setname` Class, `Search` Class, “Set Elements” on page 102

Elt_setname Destructor

Destroys the `Elt_setname` object.

```
~ Elt_setname ();
```

Description This method is called by the `delete` operator. If an expiration time for the element has been set, the expiration event is cancelled and the associated expiration object is deleted.

You do not call this method directly; instead, you use the `delete` operator defined for your subclass. In your own subclasses, you must add functionality in the destructor to clean up any additional references that you make to the element.

See Also `delete` Operator, “Set Elements” on page 102

cancel Method

Cancels the expiration event for the element.

```
int cancel ();
```

Returns Zero when successful, a negative number when unsuccessful.

Description Cancels the expiration event for the element but does not delete the event marker. If no expiration has been set, returns successfully with no effect.

See Also `expire Method`

delete Operator

Recycles the memory being used to store a set element.

```
delete (MyElement * ) sr.toElement();
```

This method calls the destructor function to remove the element from the set and free the memory associated with it, and also cleans up references to the element.

In your own subclasses, you must add functionality in the destructor to clean up any additional references that you make to the element.

Before deleting a set, you should delete all elements in that set.

expire Method

Establishes or reschedules the expiration date of a set element, and optionally specifies a callback function to execute on expiration.

```
int expire (Time dt,
            ExpireMFp fp);

int expire (Time dt);
```

Argument	Description
dt	The time for the element to expire, expressed as an amount of time from when this method executes.
fp	The callback function to execute at the expiration time.

Returns Zero when successful, a negative number when unsuccessful.

Description This overloaded method establishes or reschedules the expiration of a set element. The time argument (dt) expresses how far into the future the expiration should occur. (See “Time Class” on page 268.)

✓ **NOTE:** Use the `Event` class to schedule events that are not associated with data. This is much more efficient than creating a set element for the sole purpose of using its `expire` method.

The callback argument (fp), if present, specifies a callback method to be executed at that time. If no callback is specified, the expiration triggers the previously-active callback method. If no callback method is active, the `expire` method returns an error.

Expiration You must define expiration callback methods in the same `Element` subclass as
Callback Type the `expire` method that uses them. An expiration callback method must conform to the following prototype:

```
typedef void (Element::* ExpireMFp) (void);
```

You must explicitly cast an expiration callback method to (`ExpireMFp*`) when you pass it to the `expire` method. You can use the `EXPIREMFP()` macro to do this.

new Operator

Allocates an appropriately aligned `Elt_setname` object.

```
class MyElement : public Element ...;  
MyElement * myElt = new MyElement...;
```

Returns A reference to the newly created object, or `NULL` if sufficient free storage is not available.

See Also Set Class, Search Class, “Set Elements” on page 102

Event Class

Use this class to schedule, reschedule, and cancel events.

The `Event` class provides for execution of functions at arbitrary times in the future, with efficient rescheduling of the event and the ability to cancel an event without deleting the event marker itself.



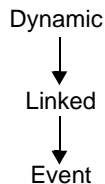
NOTE: Use this class to schedule events that are not associated with data. It is much more efficient than, for example, creating a set element for the sole purpose of using its `expire` method.

The `Event` class contains the following methods:

Method	Description
Event Constructor	Instantiates the class.
Event Destructor	Destroys an <code>Event</code> object.
<code>cancel</code> Method	Cancels the event but does not delete the <code>Event</code> object.
<code>curr</code> Method	Gets the current execution time for the event, assuming an event handler is currently executing.
<code>direct</code> Method	Designates the function to execute at the scheduled time of the event.
<code>schedule</code> Method	Specifies how far into the future the event should trigger.

Class Derivation

The `Event` class is derived from the `Linked` class.



From this class	The Event class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.

Event Callback Functions

You must define an event callback method as a member of the same `Event` subclass that uses it.

All event callbacks must be cast to the type `EventMFp*` when sent as arguments to methods of the `Event` class. For this cast to be successful, such methods must be member methods in a class that is derived from the `Event` class. This type has the following form:

```
typedef void (Event::* EventMFp) (void);
```

You can use the macro `EVENTMFp` to cast callbacks, as in the following example:

```
class MyOneHzTicker : public Event {
public:
    void ticktock (void)
    {
        schedule (Time::secs(1));
        // do something! now! for example,
        printf ("pling!\n");
    }
    MyOneHzTicker () : Event (EVENTMFp (ticktock), Time::secs
(1))
    { }
};
```

It is possible to change the service function between events, as in the following example:

```
class MyOneHzTicker : public Event {
public:
    void tick (void)
    {
        direct (EVENTMFp (tock));
        schedule (Time::msec (400));
        // do something! now! for example,
        printf ("tick\n");
    }
    void tock (void)
```

```

    {
        direct (EVENTMFP (tick));
        schedule (Time::msec (600));
        // do something else! for example,
        printf ("... tock\n");
    }
    MyOneHzTicker () : Event (EVENTMFP (tick), Time::secs (1))
    { }
};

```

Event Constructor

Creates an `Event` object.

```

Event ();

Event (EventMFP fp);

Event (EventMFP fp,
      Time dt);

```

Argument	Description
fp	Function pointer to the function to execute.
dt	Time object containing the time for the event to execute.

Returns A reference to the newly created object.

Description When constructing `Event` objects, you can specify two optional arguments:

- The callback function, which must be a member function of a class derived from `Event`
- An initial scheduled time (how long in the future, expressed as a `Time` object). To specify this, you must also specify the callback function.

When both arguments are specified, the event's service function is set and the event is scheduled. If the delay time argument is not specified, the event's service function is still set but the event is not scheduled. If you do not specify the service function on creation, you must use the `direct` method to do so before you can schedule the event.

Event Destructor

Deletes the `Event` object.

```
~ Event ();
```

Description When an event is deleted, it is implicitly unscheduled before the object is recycled.

cancel Method

Cancels the event.

```
int cancel ();
```

Returns 0 if the event was previously scheduled, or -1 if the event was not on the calendar.

Description This method removes the event from the event queue if it is currently scheduled, but does not delete the event object. If the event is not currently scheduled, returns negative.

See Also `schedule Method`

curr Method

Gets the current execution time for the event.

```
static Time curr ();
```

Returns A `Time` object.

Description This method returns a `Time` object corresponding to the real time at which the currently executing event was scheduled to occur. This method assumes that an event handler is currently executing. If an event handler is not active when the method is called, the return value is undefined.

direct Method

Designates the function to be executed at the scheduled time of the event.

```
int direct (EventMFp fp);
```

Argument	Description
fp	A pointer to the function that is to be executed at the scheduled time.

Returns Zero when successful, a negative number when unsuccessful.

Description The function to be executed must be a member function of the same private subclass of `Event`. This subclass defines the functions that the event could execute, and can retain any private state associated with the event.

You can specify the callback function when you create the object, and use this method to change it, or you can leave it unspecified on creation and use this method to specify it later.

schedule Method

Specifies how far into the future the event should be executed.

```
int schedule (Time dt);
```

Argument	Description
dt	The time after this call that the event is to be executed.

Returns Zero when successful, a negative number when unsuccessful.

Description When the specified amount of time has elapsed, the event's associated function is executed.

When called from within an event function, the new time is relative to the scheduled time of the executing event. This provides drift-free scheduling of periodic events and reasonable handling of events scheduled to higher precision than the system clock.

You can specify both the callback function and delay time when you create the object, and use this method to change the delay time, or you can leave the delay time unspecified on creation and use this method to specify the delay time later.

See Also `direct Method`, `cancel Method`

Initialization Function

The ASL provides a top-level function that you use to initialize the Policy Accelerator portion of the network application. When the application executes the ACE manager's `load` method, the host downloads the NCL rules and actions files to the Policy Accelerator, and the Policy Accelerator immediately calls this function in the downloaded actions file. You define this function to construct the ACE object (an instance of the `Ace` class), and do any other initialization that your application requires.

For an example, see “Loading and Initializing the Policy Accelerator” on page 28 of *Developing Applications Using the IX-API SDK*.

Function	Description
<code>init_actions</code> Function	Initializes the Policy Accelerator portion of the network application and constructs the specified ACE object.

`init_actions` Function

Initializes the Policy Accelerator portion of the network application and constructs the specified ACE object.

```
Ace * init_actions (ModuleID id,
                  char * name,
                  Image* obj)
```

Argument	Description
<code>id</code>	The module identification number, assigned by the Resolver.
<code>name</code>	The dictionary name of the ACE. This is the dictionary name of both the accelerator module's <code>Ace</code> object and the corresponding <code>AceManager</code> object on the host.
<code>obj</code>	The system object that contains the NCL classification rules and action code for the new ACE.

Returns A pointer to an ACE, or a `NULL` pointer when unsuccessful.

Description Use this method to initialize the Policy Accelerator portion of the network application. It is the primary entry point into the application's action code (like the `C main()` function).

Your initialization function must, at least, construct and return an ACE object (an instance of a subclass of the `Ace` class) using the passed parameters, as in the following example:

```
INITF init_actions (ModuleID id, char * name, Image* obj)
{
    return new MyAce(id, name, obj);
}
```

The value passed for the *name* argument is the dictionary name that you assigned to the `AceManager` object on the host. An ID number is assigned automatically, as is the location of the ACE code.

In addition to creating the ACE object, you should create any objects that you will need, and populate any sets or other data structures.

Linked Class

Use this class to link objects to each other in the form of a ring.

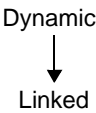
The `Linked` class is a common base for many classes in Policy Accelerator. This class provides object collection services by implementing a double-linked ring. Linked objects not currently placed in a collection of objects compose a single element ring on their own; orphan objects can be linked into any ring.

The `Linked` class contains the following methods:

Method	Description
Linked Constructor	Instantiates the class.
Linked Destructor	Destroys a <code>Linked</code> object.
link Method	Connects the object to the specified ring.
next Method	Locates the next object within the ring containing the current object.
orphan Method	Indicates whether the object is an orphan ring.
prev Method	Locates the previous object within the ring containing the current object.
unlink Method	Modifies the ring to exclude the current object.

Class Derivation

The `Linked` class is derived from the `Dynamic` class.



From this class	The <code>Linked</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

Linked Constructor

Creates a `Linked` object and initializes the linked list fields to make it into an orphan ring.

```
Linked ();
```

Returns A reference to the newly created object.

Linked Destructor

Removes the `Linked` object from its current ring before the object is freed.

```
~Linked ();
```

link Method

Connects the object to the specified ring.

```
void link (Linked * before);
```

Argument	Description
before	Pointer to a <code>Linked</code> object in the ring. The new object is inserted before this object.

Returns Nothing.

Description Links the object into the specified ring, before the argument object. If the argument object is a dummy header for a storage ring that is traversed along “next” pointers, this places the object at the end of the traversal order. To place the object at the beginning, link the new object to the first object in the ring, not the header object.

next Method

Locates the next object within the ring containing the current object.

```
Linked * next ();
```

Returns	A pointer to the located object.
Description	Returns the current object if the object is an orphan. Following <code>next()</code> links on any ring eventually visits all objects in the ring and then returns to the starting object.

orphan Method

Indicates whether the object is an orphan ring.

```
bool orphan ();
```

Returns	<code>TRUE</code> if the object represents an orphan ring, or if the object is a dummy header and the storage ring is empty.
Description	An orphan ring contains only the specified object (no dummy headers). The <code>orphan</code> method indicates whether the object is an orphan ring or a dummy header. If the storage ring is empty, the object is a dummy header.

prev Method

Locates the previous object within the ring containing the current object.

```
Linked * prev ();
```

Returns	A pointer to the located object.
Description	Returns the current object if the object is an orphan. Following <code>prev()</code> links on any ring eventually visits all objects in the ring and then returns to the starting object.

unlink Method

Modifies the ring to exclude the current object.

```
void unlink ();
```

Returns Nothing.

Description If the object is on a ring, this method modifies the ring to exclude the current object and modifies the object to become an orphan.

Memory Management Functions

The ASL provides two functions that you can use to monitor memory usage in the heap at run time.

Function	Description
<code>getmemstatvalues</code> Function	Retrieves the used and free memory totals for the Policy Accelerator's heap.
<code>mstats</code> Function	Displays memory usage statistics for the Policy Accelerator's heap.

The Policy Accelerator allocates memory on the heap for data sets that you define. The Policy Accelerator's memory is partitioned between the heap and packet buffer memory. For information on managing packet buffer memory, see the following:

- “Memory Management Classes and Functions” on page 103
- “Backlog Class” on page 119
- “Buffer Class” on page 123.

`getmemstatvalues` Function

Retrieves the used and free memory totals for the Policy Accelerator's heap.

```
int getmemstatvalues (int *pusedmem,
                    int *pfreemem)
```

Argument	Description
<code>pusedmem</code>	Pointer to a variable in which to store the used memory total.
<code>pfreemem</code>	Pointer to a variable in which to store the free memory total.

Returns One when successful, zero when unsuccessful.

Description This function stores the current used and free memory totals in the specified variables.

mstats Function

Displays memory usage statistics for the Policy Accelerator's heap.

```
void mstats (char *display_name)
```

Argument	Description
display_name	An identifying string for the report.

Returns Nothing.

Description This function prints a report on memory allocation statistics, using the specified display name. The report indicates currently free and used blocks of memory. For example, if you specify the display name "in Myfunc" the function would print something like the following:

```
Memory allocation statistics... in Myfunc
free: 0 60 2465 1477 145 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
used: 0 836 7135 1819 143 23 8 8 12 8 0 0 3 4 2 4 0 0 0 0 4 0 0
Total in use: 35677536, total free: 217760
```

Message Class

Use the `Message` class to create messages to send in upcalls and crosscalls. (Create message to be sent in downcalls on the host, using the host API's `Message` class.)

The `Message` class encapsulates the data to be transferred during an upcall or a crosscall. You can construct `Message` objects with up to two blocks of storage that you specify with a pair of `MessageBlock` objects. The maximum size for message data is 3968 bytes; that is, one page (4096 bytes) less some overhead (128 bytes) for metadata. When there are two blocks, the maximum is for the total size of both blocks.

By default, the `Message` object is automatically deleted when the call has been completed. (Note that there is a delay between the time the call is successfully sent and when it is completed.) As an alternative to this default behavior, you can specify a function to be executed when the call containing the message has been sent. You specify this function when you construct the `MessageBlock` from which the `Message` is constructed.

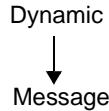
Although the `Message` object itself is handled by the system, the message block data might have locally-allocated memory that you must free after the call is received and you have finished processing the data. To free this memory, regardless of how it has been allocated during the call process, use the `Message` object's `done` method in the call handler callback. If you have specified a completion callback when constructing the `MessageBlock`, the `done` method calls it when needed.

The `Message` class contains the following methods:

Method	Description
Message Constructor	Instantiates the class.
Message Access Methods	Find the base address and length of each block in the message.
Message Completion Methods	Trigger the completion callback for the first block, the second block, or both blocks of the message.

Class Derivation

The `Message` class is derived from the `Dynamic` class.



From this class	The Message class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

Example

The following example is taken from the `BasicApp` demo application. The following method in the `ACE` subclass creates a message and sends it in an upcall:

```

void NBBasicAce::peekPacketUpcall (Buffer *buf)
{
    buf = buf; /* prevent "buf not used" compiler warning */
    packetCounter++;
    if ((packetCounter % 100) == 0) {
        msg = htonl (packetCounter);
        MessageBlock b ((char *)&msg, sizeof (msg));
        Message m (b);
        peekPacketUpcallHandle.call(&m);
    }
}
  
```

See Also

- [MessageBlock Class](#), [Crosscall Class](#), [Upcall Class](#)
- “Creating Messages and Message Blocks” on page 108 of *Developing Applications Using the IX-API SDK*

Message Constructor

Creates a `Message` object containing zero, one, or two blocks of data.

```
Message (MessageBlock &b1,
        MessageBlock &b2);

Message (MessageBlock &b1);

Message ();
```

Argument	Description
b1	Specifies where to get data for the first block of the message.
b2	Specifies where to get data for the second block of the message.

Returns A reference to the newly created object.

Description Messages contain zero, one, or two blocks of message data, which you create using the `MessageBlock` constructor.

- When you pass no message blocks to the constructor, the message tells the upcall or crosscall handler to execute its callback function, without passing it any data.
- To pass information, you can use one or two message blocks depending on your preferences. The API assigns no particular meaning to either block; you can use them arbitrarily as your application needs them.
For example, you could specify two message blocks to combine two different kinds of information in one message, such as to pass a summary of information about a packet, followed by the packet itself.

You use each `MessageBlock` object in only one `Message` object. Storage for the `MessageBlocks` used in message creation is freed by the system. You do not need to explicitly free this storage.

See Also `MessageBlock` Class

Message Access Methods

These methods return the base addresses of each block in the message and references to the lengths of the data blocks.

```
char * msg1 ();
int & len1 ();
char * msg2 ();
int & len2 ();
```

Returns	The base address or a reference to the length of the specified block.
Description	You can decrease the length of a block. Do not increase the length, as this can corrupt your data. See “MessageBlock Constructor” on page 186 for an example of how to specify the length of a block.

Message Completion Methods

These methods trigger the completion callback for the first block, the second block, or both blocks of the message.

```
void clr1 ();
void clr2 ();
void done ();
```

Returns	Nothing.
Description	<p>Although the <code>Message</code> object itself is handled by the system, the message block data may have locally-allocated memory that needs to be freed after the call is received and you have finished processing the data. To free this memory, regardless of how it has been allocated during the call process, use the <code>Message</code> object's <code>done</code> method in the call handler callback. If you have specified a completion callback when constructing the <code>MessageBlock</code>, the <code>done</code> method calls it as necessary.</p> <p>The <code>done</code> method calls the <code>clr1</code> and <code>clr2</code> methods as needed, for the first and second message blocks. You do not normally call these methods directly.</p>
See Also	CrosscallHandler Class, DowncallHandler Class, MessageBlock Class

MessageBlock Class

Use the `MessageBlock` class to create a storage area within the Policy Accelerator memory for future upcall or crosscall messages.

A message block is a place for storing chunks of a message. You can use one or two message blocks when building the final message object. You can discard the message block any time after constructing the message object.

The maximum size for message data is 3968 bytes; that is, one page (4096 bytes) less some overhead (128 bytes) for metadata. When you use two blocks to construct a message, the maximum is for the total size of both blocks.

By default, a `Message` object is automatically deleted when the call has been completed. (Note that there is a delay between the time the call is sent and when it is completed.) As an alternative to the default behavior, you can specify a function to be executed when the call containing the message has been completed. You specify this function when constructing the `MessageBlock`.

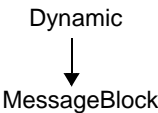
The message block data may have locally-allocated memory (separate from the `Message` object) that needs to be freed after the call is received and you have finished processing the data. To free this memory, regardless of how it has been allocated, use the `Message` object's `done` method in the call handler callback.

The `MessageBlock` class contains the following method:

Method	Description
<code>MessageBlock</code> Constructor	Instantiates the class

Class
Derivation

The `MessageBlock` class is derived from the `Dynamic` class.



From this class	The <code>MessageBlock</code> class inherits
<code>Dynamic</code>	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

Example

The following example is taken from the BasicApp demo application. The following method in the ACE subclass creates a message and sends it in an upcall:

```
void NBBasicAce::peekPacketUpcall (Buffer *buf) {
    packetCounter++;
    if ((packetCounter % 100) == 0) {
        msg = htonl (packetCounter);
        MessageBlock b ( (char *)&msg, sizeof (msg));
        Message m (b);
        peekPacketUpcallHandle.call(&m);
    }
}
```

See Also

- Message Class, Upcall Class, Crosscall Class
- “Creating Messages and Message Blocks” on page 108 of *Developing Applications Using the IX-API SDK*

MessageBlock Constructor

Creates a MessageBlock object.

```
MessageBlock b (char * msg);

MessageBlock b (char * msg,
               int len);

MessageBlock b (char * msg,
               int len,
               DoneFp done);

MessageBlock b (int len);

MessageBlock b (int len,
               int off);

MessageBlock b (Buffer * buf);
```

Argument	Description
msg	Pointer to data to be sent in the message.
len	Size of data block to be sent.
done	Function to be called after the message has been copied out of the source area.
off	Requested relative byte alignment for an allocated data area.
buf	Buffer containing the packet to be sent as the data block.

Returns A reference to the newly created object.

Description The constructor always encapsulates an area within the Policy Accelerator memory for a future upcall or crosscall message. Each form of the constructor specifies the message memory in a different way.

- MessageBlock b (char * msg);
Use this constructor to build a message block that contains a C string (a sequence of non-NULL characters followed by a NULL character). The string is not copied, so *msg* must point to storage that will not be freed before the call completes. This could be a static array, or an array that is contained in an ACE object, or any other retained storage. The length of the block is `strlen(msg)`.

Consider the following example:

```
{
    MessageBlock b1 ("reached checkpoint charlie");
    Message m (b1);
    my_upcall.send (&m);
}
```

■ MessageBlock b(char * msg, int len);

Use this constructor to build a message block that represents a fixed range of memory. In the following example, several counters are contained in a structure, then the structure is sent to the host:

```
struct MyAce_Counters {
    int count1;
    int count2;
    int count3;
};

class MyAce : public Ace {
public:
    MyAce_Counters data;
    Upcall report;
};

ACTNF example_action (Buffer *buf, MyAce *ace) {
    ace->data.count2 ++;
    return RULE_CONT;
}

send_report (MyAce *ace) {
    MessageBlock b ((char *)&ace->data, sizeof ace->data);
    Message m (b);
    ace->report.send (&m);
}
```

■ MessageBlock b (char * msg, int len, DoneFp done);

Use this constructor to notify action code when the message transfer has completed. The Message object's done method calls this function if necessary.

For instance, in the example above, if it were important that none of the counts change while the message is being sent, you could guarantee this with a busy flag and a callback function as follows:

```
struct MyAce_Counters {
    int count1;
    int count2;
    int count3;
```

```
// keep "busy" the last word, so it can be trimmed off
// before sending the message up. */
    int busy;
};

class MyAce : public Ace {
public:
    MyAce_Counters data;
    Upcall report;
};

ACTNFB example_action (Buffer *buf, MyAce *ace) {
    if (!ace->data.busy) {
        ace->data.count2 ++;
    }
    return RULE_CONT;
}

void send_done (size_t &len, char *&base){
    if (len && base) {
        MyAce_Counters * cp = (MyAce_Counters *)base;
        cp->done = 0;
    }
    len = 0;
    base = 0;
}

void send_report(MyAce *ace) {
    if (!ace->data.busy) {
        MessageBlock b ((char *)&ace->data, sizeof ace->data,
                        DONEFP (send_done));
        Message m (b);
        ace->data.busy = 1; // mark counter block as "busy"
// no need to send the "busy" word
        m.len1 () -= sizeof (ace->data.busy);
        ace->report.send (&m);
    }
}
```

■ MessageBlock b (int len);

Use this constructor to dynamically allocate a message area of a specified size, which will be automatically released to the free pool after the message is sent.

After the Message object is constructed, that object's message block accessors, `m.msg1()` and `m.msg2()`, get the address where you build the data block to be sent.

You can reduce the length of a message (as in the following example), but you cannot safely increase the length of a message.

```
{
    MessageBlock b (512); // allocate 512-byte data area
    Message m (b);
    // real code would verify "m.msg1 () != NULL"
    sprintf (m.msg1(), "%s, line %d: string is '%s'\n",
        __FILE__, __LINE__, str);
    // reduce size of message block to avoid sending a lot of
    // extra padding. Send up to and including the first
    // '\0' char.
    m.len1 () = strlen (m.msg1 ()) + 1;
    my_upcall.call (&m);
    // when we drop out of this block, storage for the Message
    // and MessageBlock objects is returned (it was on
    // the stack). The storage for the message remains
    // allocated until it is automatically freed when the
    // message has been sent to the host.
}
```

■ `MessageBlock b(int len, int off);`

Use this constructor to dynamically allocate storage of a specified size (like the previous constructor) but request that the message start at a small byte offset from the normally word-aligned allocated storage area. One use for this, shown in the following example, is to send a copy of an Ethernet packet to the host without keeping the original buffer busy. Using the offset increases the efficiency, since data copies are faster when the source and destination have the same alignment.

```
ACTNF send_copy (Buffer *buf, MyAce *ace) {
    char * pkt = buf->headerBase ();
    int len = buf->packetSize ();
    MessageBlock b (len, 3 & (unsigned)pkt);
    Message m (b);
    // real code verifies "m.msg1 () != NULL"
    // Duplicate the packet data. Takes some time, but when
    // the copy is done the original packet can continue on its
    // rounds immediately.
    bcopy (pkt, m.msg1 (), len);
    ace->packetdata.call (&m);
    return RULE_CONT;
}
```

■ `MessageBlock b(Buffer * buf);`

Use this constructor to hold a network packet buffer temporarily while a copy of the packet is sent through the messaging system. When the message has been successfully sent, processing of the buffer resumes.

When you construct a `MessageBlock` from a buffer, the method automatically increments the reference count on the `Buffer` object, and decrements the count when the `Message Completion` method is triggered in the `Message` object. Use the `busy` method of the `Buffer` class in any subsequent actions that modify the buffer, so that the modifications can be delayed until the original data has been sent, as in the following example:

```
ACTNF send_copy (Buffer *buf, MyAce *ace) {
    MessageBlock b (buf);
    Message m (b);
    ace->packetdata.call (&m);
    // Reference count of buffer has now been incremented,
    // so other code can check to see if buffer is "busy".
    return RULE_CONT;
}

ACTNF change_packet (Buffer *buf, MyAce *ace) {
    // OK to send busy buffers to targets. (In real code, may
    // be better to test for "last TTL" in NCL code instead)
    if (ip->ip_ttl < 1)
        return ace->drop (buf);
    // If sending copy of buffer to the application, and
    // send is not yet finished, hold off this rule.
    // Start over at this rule after transfer is done
    if (buf->busy ())
        return RULE_DEFER;
    // If got here, send (if any) done reading buffer,
    // OK to change it.
    make_some_changes(buf);
    return RULE_DONE;
}
```

See Also `Message Class`, `Crosscall Class`, `Upcall Class`

Name Class

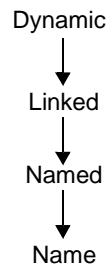
The `Name` class is used by the system to maintain an internal database of named objects, which are arbitrary pointers in the memory address space. The class is used internally.

The `Name` class contains the following methods:

Method	Description
Name Constructor	Instantiates the class
find Method	Locates an object by name in an internal database
here Method	Gets a reference to the <code>Name</code> object

Class Derivation

The `Name` class is derived from the `Named` class.



From this class	The <code>Name</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.

Name Constructor

Creates a `Name` object with a reference in the internal name database.

```
Name (Ptree * tree,
      char * name,
      void * here);
```

Argument	Description
tree	A pointer to the database in which the name should be placed.
name	The database name.
here	A pointer to the object to which the name should refer.

Returns A reference to the newly created object.

find Method

Locates an object by name in the internal name database.

```
static Name * find (Ptree * tree,
                   char * name);
```

Argument	Description
tree	A pointer to the database in which to search.
name	The database name of the object to locate.

Returns A pointer to a `Name` object, or a `NULL` pointer if no matching object is found.

Description Looks in the specified internal database for the specified name. If a match is found, returns a pointer to the object containing the name. If no match is found, returns a `NULL` pointer.

here Method

Creates a reference to the `Name` object.

```
void * & here ();
```

Returns A reference to the object to which the `Name` is applied.

Description Use this method to change the object to which a `Name` refers.

Named Class

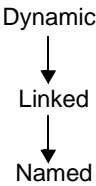
The `Named` class is used by the system to maintain an internal database of named objects, which are arbitrary pointers in the memory address space. The class is used internally.

The `Named` class contains the following methods:

Method	Description
Named Constructor	Instantiates the class.
Named Destructor	Deletes a <code>Named</code> object.
find Method	Finds a <code>Named</code> object in the internal database by name.
name Method	Gets a pointer to the internal database name of this <code>Named</code> object.

Class Derivation

The `Named` class is derived from the `Linked` class.



From this class	The <code>Named</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.

Named Constructor

Creates a `Named` object with a reference in the internal name database.

```
Named (Ptree * tree,  
       char * name);
```

Argument	Description
tree	A pointer to the dictionary into which the object should be placed.
name	The dictionary name of the object.

Returns A reference to the newly created object.

Named Destructor

Deletes a `Named` object.

```
~Named ( )
```

Description Deleting a `Named` object removes the name from the internal name database.

find Method

Locates an object in the internal name database by name.

```
static Named * find (Ptree * tree,  
                    char * name);
```

Argument	Description
tree	A pointer to the database in which to search.
name	The dictionary name of the object to locate.

Returns A pointer to a `Named` object, or a `NULL` pointer if no matching object is found.

Description Searches the specified database for the specified name. If a match is found, returns a pointer to the object containing the specified name. If no match is found, returns a `NULL` pointer.

name Method

Retrieves the database name of a `Named` object.

```
char * name ();
```

Returns The database name of the `Named` object.

NBInterfaceProp Class

The `NBInterfaceProp` class allows you to manage the properties of a Policy Accelerator's media access control (MAC) interface, such as its MAC address, speed, and duplex capability. You create an object of this type to set or read the properties of a specific interface. The properties are defined by data structures, which are described in this section.

MAC Interface Properties

The interface property feature is extensible. In the future, different properties will be defined for different types of interfaces. The properties of a MAC interface currently include the following:

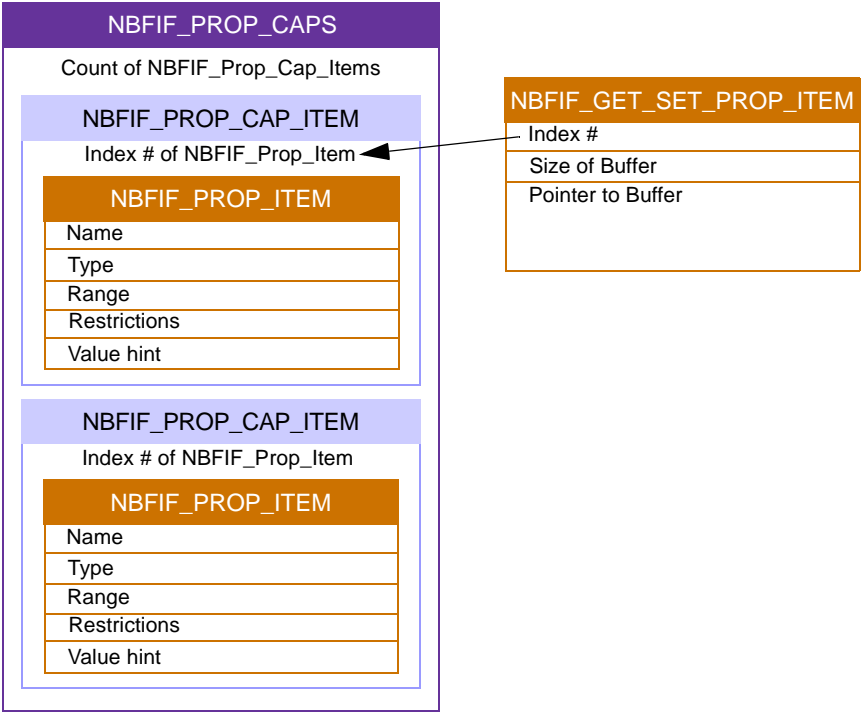
Property name	Type	Read values	Write values
Speed	Integer	0, 10, 100 megabits per second (Mbps)	10, 100 (Mbps)
Duplex	String list	Half, Full, LinkDown	Auto, Half, Full
MAC Address	Ethernet address	Ethernet address	(none)

The `Speed` and `Duplex` values reflect the current connection state of the interface, which you can monitor using the `NBLinkwatch` class. When the link is down, `Speed = 0` and `Duplex = LinkDown`. Set the value of the `Duplex` property to `Auto` to initiate automatic negotiation mode.

Interface Data Structures

You define and access properties for interfaces using the following structures:

Structure	Description
<code>NBFIF_GET_SET_PROP_ITEM</code> Structure	Holds the value of a property.
<code>NBFIF_PROP_ITEM</code> Structure	Describes a property.
<code>NBFIF_PROP_CAP_ITEM</code> Structure	Associates a property description with an index number when included in the <code>NBFIF_PROP_CAPS</code> structure.
<code>NBFIF_PROP_CAPS</code> Structure	Contains a list of property descriptions contained in <code>NBFIF_PROP_CAP_ITEMS</code> .



Methods in this Class

The NBInterfaceProp class is not derived from any other class. It contains the following methods:

Method	Description
NBInterfaceProp Constructor	Instantiates the class.
GetProperty Method	Retrieves the current value of an interface property.
GetPropertyList Method	Retrieves a list of all properties that are maintained for an interface.
SetProperty Method	Sets the value of an interface property.

Example

The following code demonstrates how to obtain the MAC address of a Policy Accelerator's MAC interface A using the NBInterfaceProp class:

```
typedef char EnetAddr[6]; // Ethernet Address - 6 bytes.  
EnetAddr m_MacAddr; // MAC/Ethernet address of an interface.
```

```

// Want to get IP address of MAC interface A
NBInterfaceProp interface_ ("A");
NBFIF_GET_SET_PROP_ITEM item;

// Get MAC address.
item.propIndx = NBFIF_PROP_ETHER_ADDR;
item.bufSizeInBytes = sizeof (m_MacAddr);
item.pValueBuf = m_MacAddr;
interface_.GetProperty (&item);

printf("MAC Addr: %02X:%02X:%02X:%02X:%02X:%02X\n",
       m_MacAddr[0], m_MacAddr[1], m_MacAddr[2],
       m_MacAddr[3], m_MacAddr[4], m_MacAddr[5] );

```

See Also

- “NBLinkwatch Class” on page 205
- “NBRmon Class” on page 208

NBInterfaceProp Constructor

Creates an NBInterfaceProp object.

```
NBInterfaceProp (char * interfacename);
```

Argument	Description
interfacename	Which Policy Accelerator MAC interface this object will manage. The value can be A or B.

Returns	A reference to the newly created object.
Description	An object of this type allows you to get and set the properties of the specified interface.

GetProperty Method

Retrieves the current value of an interface property.

```
void GetProperty (NBFIF_GET_SET_PROP_ITEM *propItem);
```

Argument	Description
propItem	A pointer to a property value structure in which you have: <ul style="list-style-type: none">■ Specified the desired property's index within an NBFIF_PROP_ITEM list■ Allocated a buffer of the proper size and type for the property's value.

Returns	<p>Nothing. Places the requested property value into propItem's buffer.</p> <p>The values that are retrieved for the Speed and Duplex properties depend on the connection state of the interface, which you can monitor using the NBLink-watch class:</p> <ul style="list-style-type: none">■ When the link is down, Speed is 0 and Duplex is LinkDown.■ When the link is active, properties have their current values. Speed can be 10 or 100 (Mbps), and Duplex can be Half or Full. In automatic negotiation mode, these are the negotiated values.
---------	---

Description	<p>This method retrieves the value of the property specified by the <code>propIndex</code> field of <code>propItem</code>, and places it into the buffer that you allocate and pass in the <code>pValueBuf</code> field of <code>propItem</code>.</p> <p>When retrieving the MAC address you must allocate a 6-byte buffer and pass it in the <code>pValueBuf</code> field of <code>propItem</code>.</p>
See Also	“NBLinkwatch Class” on page 205

GetPropertyList Method

Retrieves a list of all properties that are maintained for the interface.

```
PNBFIF_PROP_CAPS GetPropertyList ();
```

Returns	A pointer to the <code>NBFIF_PROP_CAPS</code> structure containing the array of properties supported by the interface.
---------	--

NBFIF_GET_SET_PROP_ITEM Structure

The `NBFIF_GET_SET_PROP_ITEM` structure holds the value of a property. You pass a structure of this type to the `GetProperty` and `SetProperty` methods to get and set the value of a property. It contains the following fields:

Field name	Type	Description
<code>propIndx</code>	<code>uint32</code>	The index number associated with the property.
<code>bufSizeInBytes</code>	<code>uint32</code>	The size of the value buffer in bytes.
<code>pValueBuf</code>	<code>void*</code>	A pointer to a value buffer that holds the value for the property.

NBFIF_PROP_CAP_ITEM Structure

The NBFIF_PROP_CAP_ITEM structure associates a property with an index number in the NBFIF_PROP_CAP structure. It contains the following fields:

Field name	Type	Description
propIndx	uint32	The index number of the property in the capItems array of an NBFIF_PROP_CAP structure.
propItem	NBFIF_PROP_ITEM	The property structure associated with the index.

NBFIF_PROP_CAPS Structure

The NBFIF_PROP_CAPS structure defines an array of properties that can apply to an interface. It contains the following fields:

Field name	Type	Description
capCount	uint32	The number of properties in the array.
capItems	PNBFIF_PROP_CAP_ITEM	An array of property structures.

NBFIF_PROP_ITEM Structure

The `NBFIF_PROP_ITEM` structure describes a property. It contains the following fields:

Field name	Type	Description
<code>propName</code>	<code>char [NBFIF_MAX_NAME]</code>	A string containing the name of the property.
<code>propType</code>	<code>NBFIF_PROP_TYPE</code>	The type of data that this property contains. One of the following enumerated values: <code>NBFIF_PROP_INTEGER</code> <code>NBFIF_PROP_INTEGER_ARRAY</code> <code>NBFIF_PROP_ETHER_ADDR</code> <code>NBFIF_PROP_STR_LIST</code> <code>NBFIF_PROP_MASK</code> <code>NBFIF_PROP_BOOLEAN</code>
<code>range</code>	<code>int32</code>	The range of allowed values for the property. A value of -1 means that the range of values is not limited, or that a range is not applicable. Otherwise, the meaning depends on the specified <code>propType</code> , as follows: <ul style="list-style-type: none"> ■ <code>integer</code>: Maximum acceptable value, starting at 0 ■ <code>ether_addr</code>: Maximum number of Ethernet addresses allowed ■ <code>str_list</code>: Maximum number of strings in the list
<code>restriction</code>	<code>NBFIF_PROP_RESTRICTION</code>	Whether the property can be read or written. One of the following constant values: <code>NBFIF_RESTR_READ_ONLY</code> <code>NBFIF_RESTR_WRITE_ONLY</code> <code>NBFIF_RESTR_READ_WRITE</code>
<code>possibleValue Hint</code>	<code>char [NBFIF_MAX_HINTCOUNT] [NBFIF_MAX_HINTSIZE]</code>	An array of strings indicating the possible values of the property. When you display properties to an end user, use this list to limit the values that can be entered.

SetProperty Method

Sets the value of an interface property.

```
void SetProperty (NBFIF_GET_SET_PROP_ITEM *propItem);
```

Argument	Description
propItem	A pointer to a property value structure in which you have: <ul style="list-style-type: none">■ Specified the desired property's index within an NBFIF_PROP_ITEM list■ Allocated a buffer containing the property's new value.

Returns Nothing.

Description Valid property values are:

Property	propItem.pValueBuf valid content
Duplex	10 or 100 (Mbps)
MAC Address	Cannot set this property
Speed	<ul style="list-style-type: none">■ Full■ Half■ Auto; this initiates automatic negotiation mode; you can access the negotiated Speed and Duplex property values using Get-Property method whenever the link is active

NBLinkwatch Class

The `NBLinkwatch` class allows you to monitor the network connection state of the Policy Accelerator's media access control (MAC) interfaces, which can be active (up) or inactive (down). A method allows you to check the state explicitly, and the object also checks the state automatically ten times per second.

You define a callback function to perform whatever action you want to occur when the connection state of an interface changes, or when you check the state. You specify the callback on object creation. The callback is invoked at the following times:

- When the connection state of either interface changes. Because the state is checked only 10 times each second, if the link is down for less than 0.1 seconds, the callback is not invoked.
- When you call the `checkLinks` method. The callback is invoked once for each interface, passing the current state.

The `NBLinkwatch` class is not derived from any other class. It contains the following methods:

Method	Description
<code>NBLinkwatch</code> Constructor	Instantiates the class.
<code>checkLinks</code> Method	Invokes the callback for the current link state.

NBLinkwatch Constructor

Creates an NBLinkwatch object.

```
NBLinkwatch (NBLinkwatchCallback *callback
             void *argCookie);
```

Argument	Description
callback	The service function to be called when a link state changes or is checked.
argCookie	A pointer to arbitrary data to be passed to the callback.

Returns A reference to the newly created object.

Description An object of this type allows you to monitor the link state, or network connection state, of the MAC interfaces. The callback you specify is invoked whenever the state of one of the interfaces changes, or when you call the `checkLinks` method.

Connection State Callbacks You must supply a callback that conforms to the following prototype:

```
void callback (char *interface, int state, void *argCookie)
```

Argument	Description
interface	The interface whose state is reported. Value is A or B.
state	The current state of the interface. 0 means that the link is down, and a non-zero value means that the link is active.
argCookie	A pointer to arbitrary data specified in the constructor.

The following example callback prints a message about the link state:

```
void mycallback (char *argname, int argstate, void *argCookie)
{
    if (argstate)
        printf ("Interface '%c' is now UP.\n", *argname);
    else
        printf ("Interface '%c' is now DOWN.\n", *argname);
}
```

checkLinks Method

Invokes the callback for each interface with the current link states.

```
void checkLinks ();
```

Returns Nothing.

Description This method invokes the callback method once for each interface, using the current link state.

NBRmon Class

The `NBRmon` class provides access to remote monitoring (RMON) block counters for each of the two media access control (MAC) interfaces on the Policy Accelerator. You can use the counters to construct RMON groups and management information bases (MIBs).

The Policy Accelerator constructs at least one object of the `NBRmon` class on startup. This object periodically reads counter values from the RMON block on the MAC interfaces. The initial query rate is once every 240 seconds.

RMON
Counters

Counter values are 64-bit values in network byte order, stored in a static table for each of the interfaces, `MAC_A` and `MAC_B`. The following table describes the counters that are available.

Counter ID	Description
<code>RxTotPkts</code>	Number of total packets received, including bad packets, broadcast packets, and multicast packets.
<code>RxTotOct</code>	Number of total octets of data received, including CRC and bad packets.
<code>RxBcastPkts</code>	Number of good broadcast packets received.
<code>RxMcastPkts</code>	Number of good multicast packets received.
<code>RxCRCAlignErr</code>	Number of packets received that were of the proper size ($64 \leq \text{packet length} \leq 1518$), but had a CRC error or an alignment error (a non-integral number of octets).
<code>RxAlignErr</code>	Number of received frames that did not have an integral number of octets (dribble).
<code>RxCRCErr</code>	Number of received frames that had CRC errors.
<code>RxUndSizePkts</code>	Number of packets received that were well formed but less than 64 bytes long.
<code>RxOversizePkts</code>	Number of packets received that were well formed but greater than 1518 bytes long.
<code>RxFragsPkts</code>	Number of packets received that were less than 64 bytes long and had either a CRC error or an alignment error.

Counter ID	Description
RxJabbers	Number of packets received that were more than 1518 bytes long and had either a CRC error or an alignment error.
RxPkts64	Number of packets received, including bad packets, that were 64 octets in length.
RxPkts64to127	Number of packets received, including bad packets, that were between 64 and 127 octets in length, inclusive.
RxPkts128to255	Number of packets received, including bad packets, that were between 128 and 255 octets in length, inclusive.
RxPkts256to511	Number of packets received, including bad packets, that were between 256 and 511 octets in length, inclusive.
RxPkts512to1023	Number of packets received, including bad packets, that were between 512 and 1023 octets in length, inclusive.
RxPkts1024to1518	Number of packets received, including bad packets, that were between 1024 and 1518 octets in length, inclusive.
RxGoodOct	Number of good octets received, including CRC but not including preamble/SFD. A good packet has no 4B/5B code violations, no dribble, good CRC, and proper length.
RxGoodPkts	Number of good packets received. A good packet has no 4B/5B code violations, no dribble, good CRC, and proper length.
RxDropPkts	Number of receive vectors that contain a 4B/5B coding error.
DropEvts	Number of Rx and Tx frames not counted in other counters due to contention for the RMON lock.
TxTotPkts	Total number of packets transmitted, including packets that were aborted.
TxTotOct	Total number of octets sent, including packets that were aborted.
TxBcastPkts	Number of broadcast packets transmitted, including packets that were aborted.
TxMcastPkts	Number of multicast packets transmitted, including packets that were aborted.
TxSglColPkts	Number of successfully transmitted packets that experienced exactly one collision.

Counter ID	Description
TxMultColPkts	Number of successfully transmitted packets that experienced more than one collision.
TxDeferred	Number of packets for which the first transmission was deferred because the medium was busy.
TxLateCol	Number of times a collision was detected more than 512 bit times into the transmission.
TxExcessCol	Number of packets for which transmission failed due to excessive collisions.
TxExcessDef	Number of packets for which transmission failed due to excessive deferral (>50,175 nibble clocks).
TxExcessLength	Number of packets for which transmission was aborted due to excessive length.
TxUnderun	Number of packets for which transmission failed due to data underrun.
TxTotCol	Total number of collisions seen during transmission.
TxCrcErr	Total number of CRC errors detected by the MAC core on transmit packets.

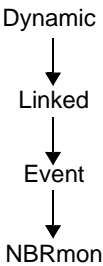
Methods in the Class

The `NBRmon` class contains the following methods:

Method	Description
<code>NBRmon</code> Constructor	Instantiates the class.
<code>NBRmon</code> Destructor	Deletes an <code>NBRmon</code> object and decrements the static reference counter.
<code>Init</code> Method	Initializes the RMON block counters.
<code>GetRmonCounters</code> Method	Retrieves receive and/or transmit counters.
<code>GetRXTXStats</code> Method	Retrieves receive and transmit counters for an interface.
<code>GetQueryRate</code> Method	Identifies the query rate for reading RMON block counters.
<code>SetQueryRate</code> Method	Sets the query rate in milliseconds for reading RMON block counters.

Class Derivation

The NBRmon class is derived from the Event class.



From this class	The NBRmon class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Event	Methods that enable NBRmon objects to read RMON counters.

See Also

- “NBInterfaceProp Class” on page 197
- “NBRmon Class” on page 208

NBRmon Constructor

Creates an NBRmon object.

```
NBRmon (uint32 uRate = MAX_RMON_RATE);
```

Argument	Description
uRate	Optional. The rate at which to read RMON block counters, in milliseconds. This number cannot be more than 240000. When you do not specify this argument, or specify a value that is out of range, the default rate of 240 seconds (240,000 milliseconds) is used.

Returns

A reference to the newly created object.

Description	<p>An <code>NBRmon</code> object reads RMON block counters. You can specify on creation how often the <code>NBRmon</code> object should query the RMON block for values. The constructor schedules events for reading counters based on the <code>uRate</code>.</p> <p>You can change the query rate after the object is created, using the <code>SetQueryRate</code> method.</p> <p>After calling the constructor, you must initialize the new object by calling its <code>Init</code> method.</p>
See Also	<code>Init</code> Method, <code>SetQueryRate</code> Method

NBRmon Destructor

Deletes an `NBRmon` object and decrements the static reference counter.

```
~NBRmon ( );
```

Description	When the reference counter becomes zero, counter values in the static table are reset to zero.
-------------	--

Init Method

Initializes the RMON block counters.

```
uint32 Init ( );
```

Returns	<code>ERR_SUCCESS</code> if successful, or an error number when unsuccessful.
Description	You must call this function after calling the <code>NBRmon</code> constructor. The method sets the query rate and reads the counters for the first time.

GetRmonCounters Method

Retrieves all receive and/or transmit counter values.

```
uint32 GetRmonCounters (IFACE Intf,
                        char* pBuffer,
                        uint32* pLength);

uint32 GetRmonCounters (IFACE Intf,
                        char* pBuffer,
                        uint32* pLength
                        uint32 RxTxFlag);
```

Argument	Description
Intf	The interface for which to get the counters. One of the following constants: MAC_A MAC_B
pBuffer	The location of a buffer which, on return, contains the retrieved counter values.
pLength	A pointer to a location which, on return, contains the number of bytes in the buffer.
RxTxFlag	Specifies which of the counters to retrieve. Optional. One of the following constants: GET_RX: Retrieve only receive counters. GET_TX: Retrieve only transmit counters. GET_RXTX: Retrieve both receive and transmit counters (default).

- Returns

ERR_SUCCESS if successful, or an error number when unsuccessful.
- Description

This method retrieves the values of the specified set of counters for the specified interface, and returns them all in the specified buffer.

You must allocate memory for the buffer of at least the size of RMON_RXTX_STATS.
- See Also

GetRXTXStats Method

GetRXTXStats Method

Retrieves all retrieve and transmit counter values.

```
uint32 GetRXTXStats (IFACE Intf,
                     RMON_RX_STATS& rxStats,
                     RMON_TX_STATS& txStats);
```

Argument	Description
Intf	The interface for which to get the counters. One of the following constants: MAC_A MAC_B
rxStats	The location of a buffer which, on return, contains the retrieved receive counter values.
txStats	The location of a buffer which, on return, contains the retrieved receive counter values.

Returns

ERR_SUCCESS if successful, or an error number when unsuccessful.

Description

This method retrieves the values of all counters for the specified interface, and returns them in the specified buffers.

- You must allocate memory for the rxStats buffer of at least the size of RMON_RX_STATS.
- You must allocate memory for the txStats buffer of at least the size of RMON_TX_STATS.

See Also

GetRmonCounters Method

GetQueryRate Method

Retrieves the current query rate for reading RMON block counters.

```
uint32 GetQueryRate ();
```

Returns Current query rate for the object, in milliseconds.

Description This method retrieves the current query rate. This is the number of milliseconds between each automatic reading of counters on the RMON block.

See Also NBRmon Constructor, SetQueryRate Method

SetQueryRate Method

Sets the query rate in milliseconds for reading RMON block counters.

```
void SetQueryRate (uint32 uMillSec);
```

Argument	Description
uMillSec	A number of milliseconds. Cannot be more than 240,000.

Returns Nothing.

Description This method sets the query rate. This is the number of milliseconds between each automatic reading of counters on the RMON block.

See Also NBRmon Constructor, GetQueryRate Method

NBStringMatchReport Class

The `NBStringMatchReport` class allows you to generate and access reports on the matching strings found by a string search in a packet buffer. A match report contains information about each matching string, including its length and location in the buffer.

An object of this type contains all of the generated reports for a string search in a specific buffer. You access individual reports by passing an index value to the accessor methods, where the first match found has an index of 0. Use the `reports` method to find the actual number of reports generated. The valid range for the indices is 0 to `reports()-1`.

If you are using the per-buffer callback to handle search results, you pass an object of this type to the `NBStringSearchEngine::SearchBuffer` call that initiates the search. The search engine fills in the object with the search results for a buffer, then passes the object to your callback when the search is complete for that buffer.



NOTE: To use the string search classes, include the following header file in your code:

```
#include <NBAction/NBStringSearch.h>
```

The `NBStringMatchReport` class is not derived from any other class. It contains the following methods:

Method	Description
<code>NBStringMatchReport</code> Constructor	Instantiates the class.
<code>end</code> Method	Finds the end of a matching string relative to the beginning of the search.
<code>len</code> Method	Finds the length of a matching string.
<code>matches</code> Method	Finds the number of matches found by a string search.
<code>reports</code> Method	Finds the number of match reports generated by a string search.
<code>sid</code> Method	Retrieves the string identifier of the search string that was matched.

Method	Description
start Method	Finds the beginning location of the buffer that was searched.
tag Method	Retrieves the string tag of the search string that was matched.

- See Also
- NBSearchContext Class, NBStringSearchEngine Class
 - “String Search Classes” on page 98
 - Chapter 10, “Finding Strings in Packets,” in *Developing Applications Using the IX-API SDK*

NBStringMatchReport Constructor

Creates an NBStringMatchReport object.

```
NBStringMatchReport (int max_reports);
```

Argument	Description
max_reports	The maximum number of match reports to be generated.

- Returns
- A reference to the newly created object.
- Description
- Constructs the object that contains match reports for the matching strings found by a string search. The number of reports generated is limited to the specified maximum, but the actual number generated may be smaller than the maximum.
- To limit the cost of memory allocation and initialization during critical-path buffer processing, you can create a pool of these objects during initialization and avoid creating them during searches.

end Method

Finds the end of a matching string relative to the beginning of the search.

```
int end (int idx);
```

Argument	Description
idx	The report index. The first match found is 0.

Returns The offset value of the last byte of the matching string relative to the beginning of the search, for the matching string specified by the index value.

Description Use the `start` method to find the starting location of the search. Use the `len` method to find the length of the matching string. Subtract the length from the end offset to find the starting location of the string relative to the starting location of the search.

len Method

Finds the length of a matching string.

```
int len (int idx);
```

Argument	Description
idx	The report index. The first match found is 0.

Returns The length in bytes of the matching string specified by the index value.

Description Subtract the length from the end offset to find the starting location of the matching string relative to the starting location of the search.

matches Method

Finds the number of matches found by a string search.

```
int matches ();
```

- Returns

The total number of matches recognized in the current buffer, including any match that started in the previous buffer.
- Description

If this number is greater than the `max_reports` value used to set the size of the report object, more matches were found than were reported.

reports Method

Finds the number of match reports generated.

```
int reports ();
```

- Returns

The number of match reports actually generated while processing the current buffer.
- Description

This number is less than or equal to the `max_reports` value used to set the size of the report object. It can be less than the number of matching strings found by the string search, as reported by the `matches` method.

sid Method

Retrieves the string identifier of the search string that was matched.

```
NBStringID sid (int idx)
```

Argument	Description
<code>idx</code>	The report index. The first match found is 0.

- Returns

A string identifier.

Description	<p>This method finds and returns the string identifier associated with the search string that was matched for the matching string specified by the index value. When you have multiple search strings, you can use either the string identifier or a tag you assign when adding search strings to determine which of the search strings was matched.</p> <p>String identifiers are sequential integers assigned by the search engine in the order in which search strings are added to the search engine collection.</p>
See Also	AddString Method in NBStringSearchEngine Class

start Method

Finds the beginning location of the buffer that was searched.

```
char * start ();
```

Returns	A pointer to the beginning of the data portion of the current buffer.
Description	<p>This method returns the pointer to the beginning of the data portion of the current buffer. This is the same value that was passed in the <code>start</code> parameter of the <code>NBStringSearchEngine::SearchBuffer</code> call used to initiate the search. It does <i>not</i> indicate where the current match starts. To find the beginning of the current match, use the <code>end</code> method and the <code>len</code> method. Subtract the length from the end offset to find the starting location of the matching string relative to the starting location of the search.</p>
See Also	SearchBuffer Method in NBStringSearchEngine Class

tag Method

Retrieves the string tag of the search string matched.

```
void * tag (int idx)
```

Argument	Description
idx	The report index. The first match found is 0.

Returns	A pointer to the tag associated with the search string that matched the string specified by the index value.
Description	A string tag is arbitrary data that you associate with a search string when you add it to the search engine’s collection. You can use either this tag or the automatically generated string identifier to determine which of several possible search strings was actually matched in a specific case.
See Also	AddString Method in NBStringSearchEngine Class

NBSearchContext Class

The `NBSearchContext` class contains configuration information that controls the operation of a string search, and maintains the state of a string search that continues into multiple packet buffers.

You extend this class to specify what to do with the search results in your application. You can define any or all of the callback functions as methods in your subclass of `NBSearchContext`.

There are two ways of acting upon search results:

- Generate a match report for a buffer using an `NBStringMatchReport` object. In this case, you provide a *per-buffer callback* that is executed when the search engine has finished searching a buffer.
- Take an action for each matching string. In this case, you provide a *per-match callback* that is executed each time a matching string is found.

The search always invokes the per-buffer callback when it has finished with the current buffer. You specify whether to invoke the per-match callback as well by setting a configuration option with the `SetOpt` method. Regardless of how you choose to handle matches, the per-buffer callback should dispose of the buffer.

A different kind of callback, the *per-reset callback*, is invoked when a reset action is completed, if the action could not be taken immediately.

You can specify that a search should continue into additional buffers as packets arrive, or that it should be limited to a single buffer. A search that is limited to one buffer is called a *simple* search. You specify whether to use a simple search by setting a configuration option with the `SetOpt` method.



NOTE: To use the string search classes, include the following header file in your code:

```
#include <NBAction/NBStringSearch.h>
```

The `NBSearchContext` class is not derived from any other class. It contains the following methods:

Method	Description
<code>NBSearchContext</code> Constructor	Instantiates the class.
<code>ActiveStrings</code> Method	Determines whether there is a potential match across buffer boundaries.
<code>SchedDelete</code> Method	Schedules the deletion of the search context object.
<code>SchedReset</code> Method	Resets the multiple-buffer state information in the search context object.
<code>SetOpt</code> Method	Sets search configuration options.
<code>SetPerBufferCallback</code> Method	Sets a callback to be invoked when the search of a buffer is complete.
<code>SetPerMatchCallback</code> Method	Sets a callback to be invoked when a matching string is found.
<code>SetPerResetCallback</code> Method	Sets a callback to be invoked on reset.

Example

The following ACE subclass definition contains references to the string search context and engine objects:

```
class CGetPkt : public Ace {
public:
    MyStrEngine str_engine;
    MyStrSearchCtx *test_search_obj;
    // a function to add a search string
    void AddStringToEngine( char *string, int user_id );
};
```

In this example, a set search uses the string search, so a member function of a set element creates the context object:

```
StreamElt::StreamElt (CGetPkt *ace, nuint32 k1, nuint32 k2,
                     nuint32 k3, nuint32 k4,
                     IP4Datagram *dgram)
: Elt_Stream(k1, k2, k3, k4)
{ ...
    search_obj = new MyStrSearchCtx (ace, this);
...}
```

The following defines a string context subclass:

```

class MyStrSearchCtx:public NBSearchContext {
public:
    CGetPkt *ace; // Pass this to search engine callback funcs.
    StreamElt *elt; // Pass this to search engine callback funcs.

    MyStrSearchCtx (CGetPkt *ace_, StreamElt *elt_ ,
                    int client);
    ~MyStrSearchCtx();
//declare callbacks
    static void OnEveryBuffer (void *ace, void *elt,
                               Buffer *buf,
                               NBStringMatchReport *rp);

    static int OnEveryMatch (void *ace, void *elt1,
                             Buffer *buf, NBStringID sid,
                             void *stringtag, int endoffset,
                             int matchlen, char *payload);
};

```

The constructor sets the options and callbacks:

```

MyStrSearchCtx::MyStrSearchCtx (CGetPkt *ace_, StreamElt *elt_)
{
    ace=ace_; // This object needs to know the owning
    elt=elt_; // ace and set element for use in its
              // callback functions

    // Specify a callback for each string match.
    SetOpt(NBS_OPT_PERSTR,1);

    // allow for simple search option:
    // if( ace->no_cross_packets )
    // SetOpt(NBS_OPT_SIMPLE,1);

    // Per-match callback. Called for every packet with a match
    SetPerMatchCallback (OnEveryMatch, ((void*)ace_),
                         ((void*)elt_) );

    // Per-buffer callback. Called when search done for each packet
    SetPerBufferCallback (OnEveryBuffer, ((void*)ace_),
                          ((void*)elt_) );
}

```

See Also

- NBStringMatchReport Class, NBStringSearchEngine Class
- “String Search Classes” on page 98
- Chapter 10, “Finding Strings in Packets,” in *Developing Applications Using the IX-API SDK*

NBSearchContext Constructor

Creates an `NBSearchContext` object.

```
NBSearchContext ( )
```

Returns A reference to the newly created object.

Description Constructs the object that holds configuration information for searches, as well as state information used internally during the search.

Because string searches are asynchronous, you cannot change the search state contained in the context object while a search is in progress. You must delete these objects using the `SchedDelete` method, rather than using a destructor or delete operation.

ActiveStrings Method

Determines whether there is a potential match across buffer boundaries.

```
int ActiveStrings ( );
```

Returns If there is an active string, 1; otherwise 0. If you are not using multiple buffers (that is, if the simple-search option is `TRUE`), always returns 0.

Description This method indicates whether there is an active string when the search has reached the end of the current buffer. An active string is one that potentially matches one of the search strings, but the end of the buffer occurs before the match can be proved or disproved.

SchedDelete Method

Schedules the deletion of the search context object.

```
void SchedDelete ()
```

Returns Nothing.

Description This method schedules the deletion of the search context object to occur when it is safe to reclaim the storage associated with it. The object is not deleted until all searches that use this context are completed.

No callback notification occurs for this method. Because string searches are executed asynchronously, this is the only method you can use to delete the search context object safely; do not delete a context object by any other means.

SchedReset Method

Resets the multiple-buffer state information in the search context object.

```
int SchedReset ();
```

Returns When successful, `NB_SUCCESS`. When not successful, `NB_PENDING`.

Description This method zeros out any multiple-buffer search state and resets the search context object to its initial state. The object is not reset until all searches that use this context are completed.

- If it is safe to do so, the method executes the reset operation immediately and returns `NB_SUCCESS`. It does not invoke the reset callback.
- If the asynchronous search has not yet completed, the method returns a value of `NB_PENDING` and schedules the reset operation for a later time when it will be safe. When the reset operation actually occurs, the search engine invokes the reset callback; see `SetPerResetCallback`.

SetOpt Method

Sets search configuration options.

```
int SetOpt (int optname,
            int cfgval)
```

Argument	Description
optname	The option to be changed. Possible values are: NBS_OPT_PERSTR NBS_OPT_SIMPLE
cfgval	The new value for the option. 0 is FALSE, 1 is TRUE.

Returns

When successful, `NB_SUCCESS`. When not successful, `NB_FAILURE`.

Description

This method sets configuration options for string searches in which this search context is used. The following options control whether you use the per-match callback and whether a search can span multiple buffers:

Option	Description
NBS_OPT_PERSTR	<p>Enable or disable callback invocation for each matching string.</p> <ul style="list-style-type: none"> ■ When <code>TRUE</code>, the per-match callback (specified using <code>SetPerMatchCallback</code>) is invoked for each matching string. ■ When <code>FALSE</code>, the per-match callback is not invoked. This is the default. <p>The per-buffer callback (specified using <code>SetPerBufferCallback</code>) is always invoked when the search is completed for a single buffer, regardless of whether the per-match callback is enabled.</p>
NBS_OPT_SIMPLE	<p>Whether searches for this context object can cross a buffer boundary.</p> <ul style="list-style-type: none"> ■ When <code>TRUE</code>, searches cannot cross boundaries, and any matching string must be contained within a single buffer. ■ When <code>FALSE</code>, searches can cross boundaries, and a matching string can be partially contained in successive buffers.

SetPerBufferCallback Method

Sets a callback to be invoked when the search of a buffer is complete.

```
void SetPerBufferCallback (NBStrPerBufferCallback callback,
                          void* context1,
                          void* context2)
```

Argument	Description
callback	The callback function.
context1	An arbitrary data pointer to pass to the callback function.
context2	An arbitrary data pointer to pass to the callback function.

Returns Nothing.

Description This method specifies the callback function to be invoked every time the processing of a complete buffer has taken place. The last two arguments are passed directly to the callback function, and can point to any data that you define for that function.

Per Buffer
Callbacks The callback function can take any action you want on the buffer. If you allocate a match report object when you initiate the search (by calling the `SearchBuffer` method of the `NBStringSearchEngine` object), the search engine fills in that object with the matching string information for the buffer, and the callback function can access the information through the object.

You must provide a callback that conforms to the following prototype:

```
(NBStrPerBufferCallback) callback (void* context1,
                                   void* context2,
                                   Buffer* buf,
                                   NBStringMatchReport* rp)
```

Argument	Description
context1	An arbitrary data pointer passed from the calling method.
context2	An arbitrary data pointer passed from the calling method.

Argument	Description
buf	A pointer to the packet buffer in which the search was executed.
rp	A pointer to the match report object provided in the call that initiated the search. If such an object has been allocated, the search engine fills it with the search results before passing it to the callback. If not using match report objects, this argument is NULL .

SetPerMatchCallback Method

Sets a callback to be invoked when a matching string is found.

```
void SetPerMatchCallback (NBStrPerMatchCallback callback,
                          void* context1,
                          void* context2)
```

Argument	Description
callback	The callback function.
context1	An arbitrary data pointer to pass to the callback function.
context2	An arbitrary data pointer to pass to the callback function.

Returns Nothing.

Description This method specifies the callback function to be invoked every time a matching string is found by a string search. The last two arguments are passed directly to the callback function and can point to any data that you define for that function.

Per Match Callbacks The callback function can take any desired action on the matching string. The search engine passes information about the matching string to the callback function. This is the same information that the search engine returns about each matching string in a match report object.

You must provide a callback that conforms to the following prototype:

```
(NBStrPerMatchCallback) callback (void* context1,
                                   void* context2,
                                   Buffer* buf,
                                   NBStringID sid,
                                   void* stringtag,
                                   int endoffset,
                                   int matchlen,
                                   char *data)
```

Argument	Description
context1	An arbitrary data pointer passed from the calling method.
context2	An arbitrary data pointer passed from the calling method.

Argument	Description
buf	A pointer to the packet buffer in which the search was executed.
sid	The string identifier of the matching string.
stringtag	A pointer to the tag of the matching string
endoffset	The offset in bytes of the end of the matching string relative to the start of the buffer (as provided to the call that initiated the search).
matchlen	The length in bytes of the matching string.
data	A pointer to the beginning of the data in the packet buffer.

This function must return one of the following values:

NBS_CONT	Continue execution, invoking the callback for subsequent string matches in the current buffer.
NBS_TERM	Terminate execution. Do not invoke the callback for subsequent string matches in the current buffer.

SetPerResetCallback Method

Sets a callback to be invoked on reset.

```
void SetPerResetCallback (NBStrPerResetCallback callback,  
                          void* context1)
```

Argument	Description
callback	The callback function.
context1	An arbitrary data pointer to pass to the callback function.

Returns Nothing.

Description This method specifies a callback function to be invoked every time you reset the search context object using the `SchedReset` method. The callback is invoked when the reset operation actually occurs, whether or not it occurs immediately. The callback can perform any action you want, using any data that you pass.

Per Reset Callbacks You must provide a callback that conforms to the following prototype:

```
(NBStrPerResetCallback) callback (void* context1)
```

Argument	Description
context1	An arbitrary data pointer passed from the calling method.

NBStringSearchEngine Class

The `NBStringSearchEngine` class provides a container for search strings, and allows you to search for matching strings in one or more packet buffers. Typically, an action function would call the `SearchBuffer` method to send the current buffer to a new or existing search.

A search engine object maintains a collection of search strings, or patterns to be matched. You use the `AddString` and `RemoveString` methods to specify the search strings. During a search, the engine looks for matches for all search strings currently in the collection.



NOTE: To use the string search classes, include the following header file in your code:

```
#include <NBAction/NBStringSearch.h>
```

Operating Modes and Callbacks

Because string searches are asynchronous, you cannot change the search state (by adding or removing search strings, for example) while a search is in progress. Similarly, you cannot start a search while you are changing the search state. To control this, an `NBStringSearchEngine` object has a current operating mode, which indicates whether you can initiate a search or change the collection of search strings. The operating modes are as follows:

Operating Mode	Description
Maintenance	In this state you can make changes to the list of search strings. No searches are currently in progress, and you cannot initiate a new search or send a new packet buffer to a continuing search.
Pending	This intermediate state occurs when a request for maintenance mode has not yet completed. You cannot change the search string list until the request is complete. Searches that are currently in progress are completed, but you cannot initiate a new search or send a new packet buffer to a continuing search.
Search	In this state you can initiate a new string search or send a new packet buffer to a continuing search. You cannot make changes to the list of search strings.

If the search engine is in the state where an action is allowed, that action occurs immediately. Similarly, if it is in the state where that action is not allowed, the action fails immediately. However, if the engine is in the intermediate state, the

action is scheduled for completion in the future. When this occurs, the method that initiated the action returns `NB_PENDING`. When the action is completed, the engine signals the completion by invoking a callback function that you provide for this purpose.

Initiating and Continuing Searches

You initiate a search using the `SearchBuffer` method, passing the current buffer with a new or reset context object. If no search is already in progress for the context object, the search engine starts a new search.

If the context specifies that the search can span multiple buffers (that is, if the simple-search flag is `FALSE`), you can call `SearchBuffer` again, passing another buffer with the same context object, to continue the search into the new buffer.

You can maintain multiple searches simultaneously, as long as each search is associated with its own context object.

Search Engine Methods

The `NBStringSearchEngine` class is not derived from any other class. It contains the following methods:

Method	Description
<code>NBStringSearchEngine</code> Constructor	Instantiates the class.
<code>AddString</code> Method	Adds a search string to the collection of strings to match against.
<code>ChangeOpMode</code> Method	Sets the operating mode.
<code>OpMode</code> Method	Finds the current operating mode.
<code>RemoveString</code> Method	Removes a search string from the collection of strings to match against.
<code>SchedDelete</code> Method	Schedules the deletion of the search engine object.
<code>SearchBuffer</code> Method	Initiates a string search.

Example

The following ACE subclass definition contains references to the string search context and engine objects:

```
class CGetPkt : public Ace {
public:
    MyStrEngine str_engine;
    MyStrSearchCtx *test_search_obj;
    // a function to add a search string
    void AddStringToEngine( char *string, int user_id );
};
```

The ACE constructor creates the engine object:

```
CGetPkt::CGetPkt (ModuleId id, char* name, Image* obj):
Ace (id, name, obj)
, str_engine( this )
{...}
```

The following defines a string engine subclass:

```
class MyStrEngine:public NBStringSearchEngine {
public:
    int nRequestAdded;
    int nActuallyAdded;
    CGetPkt *pktAce;
    NBStringID aStringID[200];
    int stringIDCount;
// constructor
    MyStrEngine (Ace *ace);
// A method to add search strings
    int AddStringToEngine( char *string, int user_id);
};
```

See Also

- NBStringMatchReport Class, NBSearchContext Class
- “String Search Classes” on page 98
- Chapter 10, “Finding Strings in Packets,” in *Developing Applications Using the IX-API SDK*

NBStringSearchEngine Constructor

Creates an NBStringSearchEngine object.

```
NBStringSearchEngine ();
```

Returns A reference to the newly created object.

Description Constructs the object that contains search strings and initiates a string search. The newly created object is in maintenance mode.

AddString Method

Adds a search string to the collection of strings to match against.

```
int AddString (char* string,
               void* stringtag,
               NBStringID* sid,
               NBStrCallback callback,
               void * context)
```

Argument	Description	
string	The string to match. Can contain constant strings and supported regular expressions, as follows:	
	Syntax	Matches
	A	The given single character, in this case A
	ABCD	The given substring, in this case ABCD
	[abc]	Any character from the given list, in this case a or b or c
	[a-f]	Any character in the given range, in this case a to f
	[a-fm-pt-z]	Any character in the list of ranges, in this case, a to f, m to p, t to z
	\Xyy	The byte value yy, where y is a hexadecimal digit
	\t	A tab character
	\n	A newline character
	\r	A linefeed character
	. (dot)	Any single character
	a*	0 or more of the previous character (except newline)
	a+	1 or more of the previous character (except newline)
	[^a]	Any character except those specified. For example: [^ 1-9] matches all characters except digits [^ \t] matches all characters except newline
stringtag	A pointer to arbitrary data to be associated with the string	

Argument	Description
<code>sid</code>	A pointer to a string identifier that you allocate. The method assigns the identifier value.
<code>callback</code>	The callback function that is invoked when the action is completed, if it has been delayed.
<code>context</code>	An arbitrary data pointer to pass to the callback function.

Returns When successful, `NB_SUCCESS`. When the object is in search mode, `NB_FAILURE`. When the operation cannot be completed immediately, `NB_PENDING`.

Description This method adds a new search string to the collection of strings to be matched.

There are two ways to identify this search string in the collection, so that when a search returns a match, you can tell which of multiple search strings was actually matched. You can use an identifier or a tag:

- The search engine assigns a string identifier and returns it at the location pointed to by `sid`. String identifiers are sequential integers that reflect the order in which search strings are added to the search engine's collection.
- The `stringtag` argument specifies arbitrary data to be associated with the string. You can use this to create a unique, nonsequential, or global identifier.

You can execute this action only when the engine is in maintenance mode.

- If the object is in search mode, the method returns `NB_FAILURE`. You must first request a mode change using the `ChangeOpMode` method.
- If the action can be executed immediately, the method returns `NB_SUCCESS`. It does not invoke the callback.
- If it is not yet safe to add the string, the method schedules the addition and returns `NB_PENDING`. When the action is completed, the engine notifies the application by invoking the callback function, passing it the tag associated with the added string.

Add String Callbacks

You must provide a callback that conforms to the following prototype:

```
(NBStrCallback) callback (int return_value,
                          void* context,
                          void* stringtag)
```

Argument	Description
return_value	Indicates whether the action completed successfully, either NB_SUCCESS or NB_FAILURE.
context	An arbitrary data pointer passed from the calling method.
stringtag	A pointer to arbitrary data associated with the string, passed from the calling method.

ChangeOpMode Method

Sets the operating mode.

```
int ChangeOpMode (int newmode,
                  NBStrCallback callback,
                  void * context)
```

Argument	Description
newmode	The new operation mode. Can be one of the following: NBS_MODE_MAINT NBS_MODE_SEARCH
callback	The callback function.
context	An arbitrary data pointer to pass to the callback function.

Returns When successful, NB_SUCCESS. When the request fails, NB_FAILURE. When the operation cannot be completed immediately, NB_PENDING.

Description This method issues a request to change the operating mode of the object.

- If the request fails due to an error, the method returns NB_FAILURE.
- If the action can be executed immediately, the method returns NB_SUCCESS and does not invoke the callback.
- If it is not yet safe to change the mode, the method schedules the change and returns NB_PENDING. When the action is completed, the engine notifies the application by invoking the callback function.

Change Mode Callbacks You must provide a callback that conforms to the following prototype:

```
(NBStrCallback) callback (int return_value,
                          void* context,
                          int oldmode,
                          int newmode)
```

Argument	Description
return_value	Indicates whether the action completed successfully, either NB_SUCCESS or NB_FAILURE.
context	An arbitrary data pointer passed from the calling method.

Argument	Description
oldmode	The previous mode state of the object, either NBS_MODE_MAINT or NBS_MODE_SEARCH.
newmode	The new mode state of the object, either NBS_MODE_MAINT or NBS_MODE_SEARCH.

OpMode Method

Finds the current operating mode.

```
int OpMode ();
```

Returns The current operating mode of the search engine object.

Description The operating mode of the engine object can be one of the following:

Operating Mode	Description
NBS_MODE_MAINT	You can make changes to the internal state of the object, such as adding new search strings. No buffer searching takes place.
NBS_MODE_PENDING	An intermediate state that occurs when a request for maintenance mode has not yet completed. No buffer searching takes place.
NBS_MODE_SEARCH	The object is ready to perform string searches.

Use the `ChangeOptMode` method to request a change to the operating mode.

RemoveString Method

Removes a search string from the collection of strings to match against.

```
int RemoveString (NBStringID sid,
                  NBStrCallback callback,
                  void* context)
```

Argument	Description
sid	The string identifier of the search string to be removed, as assigned by the AddString method.
callback	The callback function.
context	An arbitrary data pointer to pass to the callback function.

Returns When successful, NB_SUCCESS. When the object is in search mode, NB_FAILURE. When the operation cannot be completed immediately, NB_PENDING.

Description This method removes the string with the specified identifier from the collection of strings being searched for. You can execute this action only when the engine is in maintenance mode.

- If the object is in search mode, the method returns NB_FAILURE. You must first request a mode change using the ChangeOpMode method.
- If the action can be executed immediately, the method returns NB_SUCCESS and does not invoke the callback.
- If it is not yet safe to remove the string, the method schedules the removal and returns NB_PENDING. When the action is completed, the engine invokes the callback function, passing it the tag associated with the removed string.

Remove String Callbacks You must provide a callback that conforms to the following prototype:

```
(NBStrCallback) callback (int return_value,
                          void* context,
                          void *stringtag)
```

Argument	Description
return_value	Indicates whether the action completed successfully, either NB_SUCCESS or NB_FAILURE.

Argument	Description
context	An arbitrary data pointer passed from the calling method.
stringtag	A pointer to arbitrary data associated with the removed string.

SchedDelete Method

Schedules the deletion of the search engine object.

```
int SchedDelete (NBStrCallback callback,
                void* context)
```

Returns When successful, `NB_SUCCESS`. When the object is in search mode, `NB_FAILURE`. When the operation cannot be completed immediately, `NB_PENDING`.

Description This method schedules the deletion of the search engine object itself, to be completed when it is safe to free the object's memory.

You can execute this action only when the engine is in maintenance mode.

- If the object is in search mode, the method returns `NB_FAILURE`. You must first request a mode change using the `ChangeOpMode` method.
- If the action can be executed immediately, the method returns `NB_SUCCESS` and does not invoke the callback.
- If it is not yet safe to delete the string, the method schedules the deletion and returns `NB_PENDING`. When the action is completed, the engine notifies the application by invoking the callback function. By the time the callback is invoked, the object has been deleted and references to it are invalid.

Schedule Deletion Callbacks You must provide a callback that conforms to the following prototype:

```
(NBStrCallback) callback (int return_value,
                          void* context)
```

Argument	Description
return_value	Indicates whether the action completed successfully, either <code>NB_SUCCESS</code> or <code>NB_FAILURE</code> .
context	An arbitrary data pointer passed from the calling method.

SearchBuffer Method

Initiates a new string search or passes a new buffer to a multiple-buffer search.

```
int SearchBuffer (Buffer buf,
                  char *start,
                  int len,
                  NBSearchContext *sc,
                  NBStringMatchReport *rp)
```

Argument	Description
buf	The packet buffer in which to search.
start	A pointer to the location in the buffer at which to begin the search.
len	The maximum number of bytes to inspect in the search.
sc	A pointer to the search context object to be used to control the search. If the context contains a current multiple-buffer search state, that search continues into this buffer.
rp	A pointer to a match report object in which to store match reports. When NULL, no match reports are generated.

Returns

When successful, NB_SUCCESS. When the object is not in search mode, NB_FAILURE. When the search engine cannot accept the request, NB_BUSY.

Description

Depending on the state kept in the specified context object, this method initiates a new search or passes the buffer to an ongoing multiple-buffer search. The search is executed in the specified (current) packet buffer, matching the search strings currently in the engine object’s collection.

The search starts at the location pointed to by `start`, and continues for the number of bytes specified by `len`. The `start` parameter typically points to the beginning of the data portion of the buffer.

You can call this method only when the engine is in search mode.

- If the object is not in search mode, the method immediately returns NB_FAILURE. You must first request a mode change using the `ChangeOpMode` method.
- If the engine is currently unable to accept the request, the method returns NB_BUSY. In this case, you must retry the search later.
- If the search is successfully started, the method returns NB_SUCCESS.

Reporting Matches

The search engine reports matches according to the per-match search option set in the specified search context object. If the per-match option is `TRUE`, the search engine invokes the per-match callback as each match is found, passing information about the matching string. In any case, the search engine always invokes the per-buffer callback (if any) when the search has been completed for the buffer.

If you create and pass a match report object, the search engine generates match reports for the buffer, which the per-buffer callback can access through the match report object. If you use the per-match callback, you might not need to use match report objects.

You use the per-buffer callback to determine what to do with the buffer. If the buffer is needed after the per-buffer callback returns, you must use `Buffer::inccref()` to mark the buffer in use. You need not do this if the callback disposes of the buffer by passing it, dropping it, or sending it to another target.

Single- or Multiple-Buffer Searches

The simple-search option in the specified context object determines whether there is a multiple-buffer search in progress.

- If the simple-search option is true, the engine initiates a new search in the current buffer. The search terminates when it reaches the end of the buffer.
- If the simple-search option is false, the context object maintains the search state across buffer boundaries. The current buffer is passed to the ongoing search, which continues into the next buffer that is sent with the same context object.

Pool Class

The `Pool` class maintains a linked list of free objects and a large block from which new objects can be created. Objects are provided from the pool without any added overhead; the code that recycles the objects must determine to which pool to return the object.

The `Dynamic` class uses the `Tagged` class, derived from the `Pool` class, to quickly allocate objects of fixed sizes at specified offsets from specified power-of-two alignments. `Pool` objects restock the raw memory resources from the Policy Accelerator memory pool as required.

The `Pool` class is not derived from any other class. It contains the following methods:

Method	Description
<code>Pool</code> Constructor	Instantiates the class
<code>Pool</code> Destructor	Deletes a <code>Pool</code> object
<code>free</code> Method	Releases the specified object into the pool
<code>take</code> Method	Takes an object from the pool

Pool Constructor

Creates a `Pool` object.

```
Pool (size_t bytes,
      size_t align,
      int offset,
      int restock);
```

Argument	Description
<code>bytes</code>	The size of the objects to be allocated from this pool.
<code>align</code>	The modulus part of the alignment requirement of the objects.

Argument	Description
<code>offset</code>	The offset part of the alignment requirement of the objects.
<code>restock</code>	The minimum number of objects to add to the pool when it is empty.

Returns A reference to the newly created object.

Description Constructs the object that describes the contents of the memory pool and that contains the configuration control information for how future allocations will be handled.

The `offset` argument enables allocation of classes where a specific member needs to be strongly aligned; for example, objects from the `Buffer` class contain an element called “hard” that must start at the beginning of a 2048-byte-aligned region.

The `restock` argument controls how much memory is allocated from the surrounding environment when the pool is empty. Enough memory is allocated to contain at least the requested number of objects of the specified size, at the specified offset from the alignment modulus. Setting `restock` to zero prevents any attempt to allocate memory from the surrounding environment.

Pool Destructor

Deletes a `Pool` object.

```
~ Pool ();
```

Description When a `Pool` object is deleted, all memory allocated from the surrounding environment by the memory pool object is released. Externally stocked objects are not affected.

Objects residing in memory allocated by the pool from the surrounding environment must not be accessed after the pool has been deleted.

free Method

Releases the specified object into the pool.

```
void free (void * blk);
```

Argument	Description
blk	Pointer to the block of memory to free within the memory pool.

Returns Nothing.

Description Releases an object into the pool, so the memory containing it can be recycled. `NULL` pointers can be passed to this method but are ignored. You can manually stock a pool with appropriate objects by freeing objects (with the appropriate alignment characteristics) that were not originally allocated from the pool. Additionally, the `DEBUG` version of the library checks that released objects meet the alignment criteria for the pool, triggering an assertion failure if the memory is badly aligned.

take Method

Takes an object from the pool.

```
void * take ();
```

Returns A pointer to the appropriately allocated object, or `NULL` if the pool needs but cannot allocate additional memory.

Description Obtains an object from the pool, guaranteeing that it starts at the byte offset (specified in the `Pool` constructor) from the byte alignment (also specified in the `Pool` constructor) boundary, attempting to recycle recently released objects before acquiring additional resources from the surrounding environment.

If the `restock` argument was set to zero when the object was constructed, the `take` method does not acquire resources from the surrounding environment.

Rate Class

Use this class to track event rates and bandwidths so you can watch for rates that exceed desired values.

The `Rate` class provides a simple way to track event rates and bandwidths so that you can watch for rates that exceed desired values.

The `Rate` class contains the following methods:

Method	Description
Rate Constructor	Instantiates the class.
add Method	Adds the specified number of packets to the rate.
count Method	Calculates the best estimate of the current trailing rate of the events over the last (longer) period.
clear Method	Clears the count and associated internal historical state information to prevent triggering alarm code on every packet observed over the limit.

Class Derivation

The `Rate` class is derived from the `Event` class.



From this class	The <code>Rate</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

From this class	The Rate class inherits
Linked	Methods that enable objects to link to each other.
Event	Methods that enable a rate to be handled as a normal Policy Accelerator event. However, if the scheduling of the associated event is modified, the Rate class produces incorrect results.

Rate Constructor

Creates a Rate object with a specific sampling period.

```
Rate (Time period);
```

```
Rate (Time period,
      int divide);
```

Argument	Description
period	The sampling period.
divide	How finely to divide the period. Default value is 8.

Returns A reference to the newly created object.

Description The Rate constructor enables you to specify arbitrary sampling periods. You can optionally specify how finely to divide the period. Though larger divisors result in more precise rate measurement, they require more overhead because the Rate object schedules events for each of the shorter periods while there are events within the longer period.

add Method

Adds the specified number of events to the count.

```
void add (int howMany);
```

Argument	Description
howMany	Specifies the number of counts to add. Default value is 1.

Description Events can be packets, bytes, errors, or anything else for which the application wants to monitor the rate.

count Method

Calculates the best estimate of the current trailing rate of the events over the last (longer) period.

```
int count ();
```

Returns The trailing rate, in events-per-period.

clear Method

Clears the count and associated internal historical state information to prevent triggering alarm code on every packet observed over the limit.

```
void clear ();
```

Returns Nothing.

Search Class

The `Search` class is the data type returned by all set searching operations. An object of this type is returned by the `locate` method of `Set` subclasses; see `Set_setname` Class on page 256.

Use this class to discover the results of a search, and to manipulate the set for which the search was performed.

The `Search` class contains the following methods:

Method	Description
Result access methods	Return Boolean values indicating the result of the search. <ul style="list-style-type: none">■ <code>ran</code> Method: Indicates whether the search ran, or did not run because requirements for set membership were not met.■ <code>hit</code> Method: Indicates whether the search ran and found a matching record.■ <code>miss</code> Method: Indicates whether the search ran and did not find a matching record.
<code>insert</code> Method	Inserts the specified element into the place where the search result failed to find the designated set of keys.
<code>toElement</code> Method	Gets a pointer to the set element object that was found during the search.

Class Derivation

The `Search` class is derived from the `Dynamic` class.



From this class	The search class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

Example

The following example is taken from the `IPPairs` demo application. The single action function is called by single NCL rule for all IP packets. It receives the search object and a pointer to the IP portion of the current packet

```
ACTNF do_packet(Buffer *buf, DemoAce *ace,
                Search sr, Proto_ip *ip)
{
    if (sr.hit())
    {
        //Most common case: another hit on a conversation.
        pair * p = (pair *)sr.toElement(); //find matching element
        p->hit(); //increment elmnt's counter, reset expiration timer
        return RULE_CONT;
    }
    if (sr.miss()) // No matching element for this packet
    {
        // add new set element with current src & dst addresses
        // create new element
        pair * p = new pair(ip->src(), ip->dst());
        // add to set
        sr.insert(p);
        return RULE_CONT;
    }
    // If we got here, the search did not run.
    // This demo's NCL rule does not call this fn
    // in this case, but this would be where to handle it
    return RULE_CONT;
}
```

See Also

- “Set Management Classes” on page 101
- “Sets and Named Searches” and “Synchronizing NCL with Action Code” in Chapter 6
- Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*

hit Method

Indicates whether a matching record was found by the search.

```
bool hit ();
```

Returns TRUE if the `Search` object represents an NCL search that has been executed, and the search found a matching record. FALSE if the search was not executed or a matching record was not found.

Description For a search that was executed, either `hit` or `miss` returns TRUE, and the other FALSE. For a search that was not executed, both return FALSE.

insert Method

Inserts an element into a set.

```
void insert (Element *);
```

Returns Nothing.

Description Inserts the specified element into the place in the set where the search failed to find an element whose keys match the designated packet field values.

If the search keys match the keys used to build the new element, then subsequent searches for those keys will locate the object; otherwise, the behavior is undefined.

miss Method

Indicates whether the search failed to find a matching record.

```
bool miss ();
```

Returns TRUE if the `Search` object represents an NCL search that has been executed, and the search did not find a matching record. FALSE if the search was not executed or a matching record was found.

Description For a search that was executed, either `hit` or `miss` returns TRUE, and the other FALSE. For a search that was not executed, both return FALSE.

ran Method

Indicates whether a search ran.

```
bool ran ();
```

Returns TRUE if the `Search` object represents an NCL search that has been executed. FALSE if the prerequisites for running an NCL search were not met.

toElement Method

Retrieves a pointer to the set element object that was found during the search.

```
Element * toElement();
```

Returns A pointer to the matching element.

Description When the search succeeds in finding a set element whose keys match the packet field values, this method returns a pointer to the matching element.

You must cast the returned object pointer into the `Element` subclass you have created for the set.

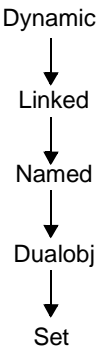
Set Class

The `Set` class is a base class used to construct classes that represent searchable sets. These are originally defined in Network Classification Language (NCL). For each set `setname`, the NCL compiler creates a subclass of the `Set` class named `Set_setname`. You must not create any further subclasses.

You do not create `Set` objects directly; instead, you use the constructor in the generated set subclass.

Class
Derivation

The `Set` class is derived from the `Dualobj` class.



From this class	The <code>Set</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The <code>ace</code> method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerators.

See Also

`Set_setname` Class

Set_setname Class

For each **set** directive defined in Network Classification Language (NCL), the compiler produces an adjusted **Set** subclass *Set_setname*, using the name of the set. You must not create any further subclasses.

The NCL compiler uses the number of words of key information to customize the argument list for the lookup function. It uses the set definition's *size_hint* to adjust a protected field within the class. It uses the number of key words (*nkeys*) to construct a *locate* method with the appropriate number of arguments.

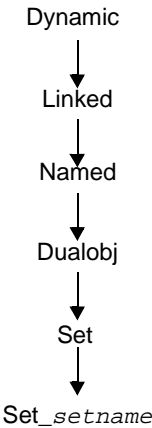
ACEs that manipulate sets must include an object of the customized **Set** class as a member of the ACE. Add a set declaration for each set to your subclass of the ACE class, with the same name that you declared for that set in the NCL file.

The *Set_setname* class contains the following methods:

Method	Description
<i>Set_setname</i> Constructor	Instantiates the class.
<i>first</i> Method	Finds the first member element of the set.
<i>locate</i> Method	Searches for a member of the set.
<i>next</i> Method	Finds the member element that follows a given element of the set.

Class Derivation

The *Set_setname* class is derived from the **Set** class.



From this class	The Set_setname class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The ace method and semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerators.
Set	All public methods.

Example

The following ACE subclass definition declares two set objects, and the constructor creates the set objects:

```
class CONNAce : public Ace {
public:
    CONNAce (ModuleId id, char* name, Image* obj);
    ~CONNAce ();
    Set_nets nets;
    Set_conns conns;
}

CONNAce::CONNAce (ModuleId id, char* name, Image* obj):
    Ace (id, name, obj)
    ,nets (id, this, "nets")
    ,conns (id, this, "conns")
{ }
```

The destructor of the custom set subclass (or the destructor of the ACE, if the set is defined as part of the ACE) should use the iterator functions to clean up set elements, using code like the following:

```
MyElement * scan;
MyElement * next;
...
scan = first(); //Set_myset.first() if in ACE
while (scan != NULL) {
    next = scan->next();
    delete scan;
    scan = next;
}
```

See Also

- “Set Management Classes” on page 101
- “Sets and Named Searches” and “Synchronizing NCL with Action Code” in Chapter 6
- Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*

Set_setname Constructor

Creates a *Set_setname* object.

```
Set_setname (ModuleId id,
             Ace * ace,
             char * name);
```

Argument	Description
id	The module identification number, assigned by the Resolver. .
ace	Pointer to the Ace object in the Policy Accelerator for the ACE that owns this set.
name	The name of the set. This is the same as the name specified in the NCL set definition.

Returns A reference to the newly created object.

Description Locates and connects the object to the data structures used by the classification code in response to set search directives.

You must create an object of this class in the accelerator module’s ACE object for each set that you have defined, giving the object the same name that was defined for the set in the NCL code.

first Method

Finds the first member element of the set.

```
Element* first ()
```

Returns A pointer to a set element, or `NULL`.

Description This method finds and returns the first member element of the set. If there are no elements, the method returns `NULL`. You can use this, together with the `next` method, to iterate through members of the set and delete them before deleting the set.

See Also “Deleting Sets” on page 125 of *Developing Applications Using the IX-API SDK*

locate Method

Searches the set for a record matching the specified keys.

```
Search locate (nuint32 k1, ... nuint32 kn);
```

Argument	Description
k1	The key values to match. You must pass the number of key values specified by <code>nkeys</code> in the NCL set definition.

Returns A `Search` object containing the results of the search.

Description This method examines the set and attempts to find the unique member of the set whose key values match the specified key values.

Note that this method does not have a variable number of arguments. You must pass the number of arguments defined for the specific set by the `nkeys` argument in the NCL `set` statement that declared and defined it.

Key values are network-ordered words. The compiler handles conversion between host and network order. See “Byte Order and Intermodule Communication” in Chapter 2.

See Also `Search` Class, “Sets and Named Searches” in Chapter 6.

next Method

Finds the member element following a given element of the set.

```
Element* next (Element *ep)
```

Argument	Description
ep	A pointer to a member element of the set.

Returns A pointer to a set element, or `NULL`.

Description This method finds and returns the member element of the set immediately after the specified element. If there are no elements, or if the specified element is the last one, the method returns `NULL`. You can use this, together with the `first` method, to iterate through members of the set and delete them before deleting the set.

See Also “Deleting Sets” on page 125 of *Developing Applications Using the IX-API SDK*

Tagged Class

The `Tagged` class is derived from the `Pool` class. It adds a small amount of overhead space as a prefix to the object, which contains a pointer back to the object's home pool. Code that frees objects taken from a `Tagged` class can do so without deciding where to return the object.

The `Dynamic` class uses this class to free tagged objects into the appropriate memory pool. You do not normally use the `Tagged` class directly.

If an object has strong alignment requirements, adding the `Tagged` overhead can cause much space to be wasted between the objects. For example, if the objects were 32 bytes long and were required to start on 32-byte boundaries, an additional word would cause another 28 bytes of padding to be wasted between adjacent objects.

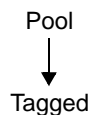
The `Tagged` class adds a second (static) version of the `take` method, which is passed the size of the object to be allocated. The `Tagged` class manages an appropriate set of pools based on possible object sizes, grouping objects of similar size together to limit the number of pools and allow sharing of real memory between objects of slightly different sizes. Currently, this interface has a hard upper limit (around 64KB) on the size of the objects that it is willing to allocate, and returns a null pointer if asked to go above that limit.

The `Tagged` class contains the following methods:

Method	Description
<code>free</code> Method	Releases the <code>Tagged</code> object into the appropriate pool so the memory containing it can be recycled.
<code>take</code> Method	Locates or creates a <code>Tagged</code> pool to manage objects of approximately the same size.

Class Derivation

The `Tagged` class is derived from the `Pool` class.



From this class	The Tagged class inherits
Pool	All public methods.

free Method

Releases the object at the specified address into the Tagged pool from which it was originally allocated, so that the storage can be reused.

```
static void free (void * blk);
```

Argument	Description
blk	Pointer to the block of memory to free within the memory pool.

Returns Nothing.

Description Unlike normal pools, it is not possible to manually stock a Tagged pool. Additionally, the debug version of the library checks that released objects meet the alignment criteria for the pool, triggering an assertion failure if the memory is badly aligned.

If you pass a null pointer, it is ignored.

See Also Pool Class, Tagged Class, take Method

take Method

Allocates objects from an automatically selected Tagged pool.

```
static void * take (size_t size);
```

Argument	Description
size	The size of the memory block to take.

Returns	A pointer to the appropriately allocated object, or NULL if the Tagged pool needs but is unable to allocate additional memory.
Description	<p>In addition to the member method <code>Tagged::take()</code> inherited from <code>Pool</code>, the <code>Tagged</code> class adds this static method that locates (or creates) a <code>Tagged</code> pool to manage objects of approximately the same size, and then allocates an object from it.</p> <p>Setting the <code>restock</code> argument to zero on object creation prevents any attempt to allocate memory from the surrounding environment. (See <code>Pool</code> Constructor on page 245.)</p>
See Also	<code>Dynamic Class</code> , <code>Pool Class</code>

Target Class

Use the `Target` class to create target objects in ACEs within a Policy Accelerator.

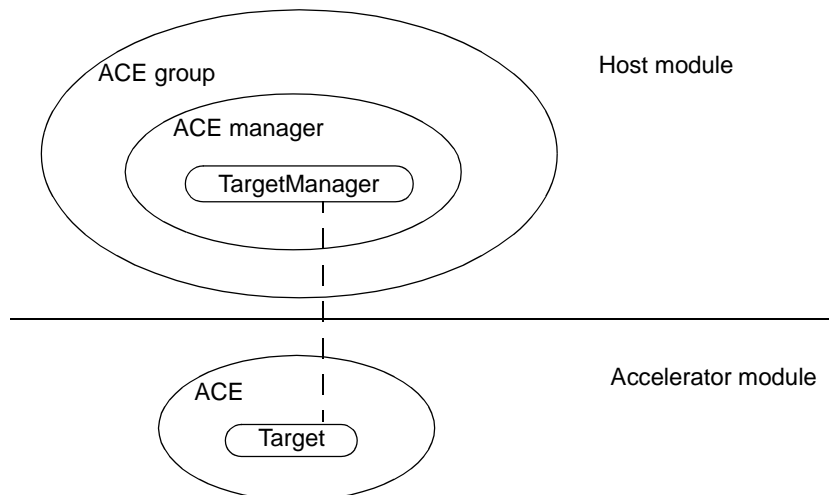
Targets represent paths to be taken through the set of ACEs by the various packet buffers. The Resolver works with the application to inform the `Target` objects of how the various ACEs, stacks and interfaces in the system are connected.

The `Target` objects represent the next bit of hardware or software that looks at a packet along a selected path. This might be another ACE within the same application, an ACE within a completely different application, a network transmission queue, or a built-in service for packet dropping or cryptography.

All ACEs start with two default `Target` objects, called the default pass and drop targets. You can create additional targets, representing additional directions packets can take upon leaving the ACE, in your `Ace` subclasses.

In actions, you use the `Target` object's `take` method to designate the target through which the buffer will be directed when the ACE's processing is complete.

Each `Target` object is associated with one `TargetManager` object with the same dictionary name in the host module.

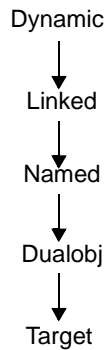


The `Target` class contains the following methods:

Method	Decription
Target Constructor	Instantiates the class.
take Method	Arranges for the specified buffer to be processed next by the service on the other end of this target.

Class Derivation

The `Target` class is derived from the `Dualobj` class.



From this class	The <code>Target</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The <code>ace</code> method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerators.

See Also

- Chapter 5, “Controlling Packet Flow,” of *Developing Applications Using the IX-API SDK*
- `TargetManager` Class in Chapter 3, “Host API.”

Target Constructor

Creates a `Target` object.

```
Target (ModuleId id,
        Ace * ace,
        char * name);
```

Argument	Description
id	The module identification number, assigned by the Resolver.
ace	Pointer to the <code>Ace</code> object in the Policy Accelerator for the ACE that owns this target.
name	The dictionary name of the target. This must be the same as the dictionary name of the associated <code>TargetManager</code> object in the host module.

Returns A reference to the newly created object.

take Method

Directs a packet to this target.

```
int take (Buffer * b);
```

Argument	Description
b	Pointer to the buffer.

Returns Returns an action-function return-value constant (see “Custom Action Functions” on page 117).

Description This method arranges for the specified buffer to be processed next by the service on the other end of this target.

Time Class

The `Time` class provides a common format for carrying a time value. Absolute, relative, and elapsed times are all handled identically. Various other classes use `Time` objects to specify absolute times and time intervals.

You can construct `Time` objects for specified numbers of standard time units (microseconds, milliseconds, seconds, minutes, hours, days and weeks). Use the access methods to extract standard time periods from a `Time` object.

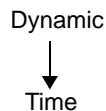
Conversion to and from sixty-four-bit unsigned integer values are automatic. Scalar operators and assignment operators allow you to manipulate time values using standard scalar numbers of standard time units.

The `Time` class contains the following methods and operators:

Method	Description
<code>Time</code> Constructor	Instantiates the class.
<code>curr</code> Method	Returns the current time.
Access Methods	Various accessor methods render time values as scalar multiples of standard time units, truncated toward zero.
Builder Methods	Various static methods construct time values corresponding to scalar multiples of standard time units.
Assignment Operators	Operators for manipulating the time values.
Conversion Operator	The <code>int64</code> operator converts time values to normal scalar types.

Class Derivation

The `Time` class is derived from the `Dynamic` class.



From this class	The <code>Time</code> class inherits
Dynamic	The semantics that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.

Time Constructor

Creates a `Time` object.

```
Time ();  
  
Time (int64 raw);
```

Argument	Description
<code>raw</code>	Specifies the initial value for the <code>Time</code> object.

Returns A reference to the newly created object.

Description Use the constructor with no argument to construct an uninitialized `Time` object. Use the constructor with an argument to construct a `Time` object that is initialized to the specified number of ticks of the highest resolution clocks efficiently accessible in the environment. (The C++ compiler uses the second constructor to perform implicit transparent conversion from the usual scalar values and `Time` objects.)

When you construct a time value, you can initialize it to a standard time-unit value using one of the `Time` builder methods. For example:
`Time timeout(Time::secs(5));`

curr Method

Returns the current time.

```
static Time curr ();
```

Returns A `Time` object containing the current time.

Description Returns the value of the high precision environmental clock as sampled at the top of the current real-time dispatch loop. Repeated calls to `Time::curr()` return identical values until the real time loop has completed and is recycled.

Access Methods

Use the following accessor methods to render a time value as a scalar multiple of standard time units, truncated toward zero.

```
int64 usec ();
int64 msec ();
int64 secs ();
int64 mins ();
int64 hour ();
int64 days ();
int64 week ();
```

Returns A number expressing the time in the specified unit.

Description These accessors, when called without arguments, return the value of the `Time` variable in the appropriate units, rounded down. For example, a time value corresponding to 5.9 seconds is reported as 5 by the `Time::secs()` method.

Builder Methods

Use the following static methods to construct time values corresponding to scalar multiples of the standard time units.

```
static Time usec (int64 t);  
static Time msec (int64 t);  
static Time secs (int64 t);  
static Time mins (int64 t);  
static Time hour (int64 t);  
static Time days (int64 t);  
static Time week (int64 t);
```

Argument	Description
t	Multiplier for constructing the time value.

Returns A `Time` object.

Description These builder methods return `Time` objects that express the specified length of time. For example, `Time::secs(5)` returns a time value representing five seconds.

Assignment Operators

The `Time` class defines the following set of assignment operations on `Time` objects, with standard semantics corresponding to the same expression written as a binary operator or assignment.

```
Time& operator += (Time t); //add time t to a time
Time& operator -= (Time t); //subtract time t from a time
Time& operator *= (int i); //multiply a time by i
Time& operator /= (int i); //divide a time by i
Time& operator %= (Time t); //modulus of time by time i
Time& operator <<= (int i); //shift time left by i
Time& operator >>= (int i); //shift time right by i
Time& operator |= (Time t); //OR time with time t
Time& operator ^= (Time t); //XOR time with time t
Time& operator &= (Time t); //AND time with time t
```

Argument	Description
t	The <code>Time</code> object with which to operate on the left-hand time.
i	A number by which to manipulate the left-hand time value.

Returns A reference to the `Time` object on the left side of the operator (which contains the manipulated value).

Conversion Operator

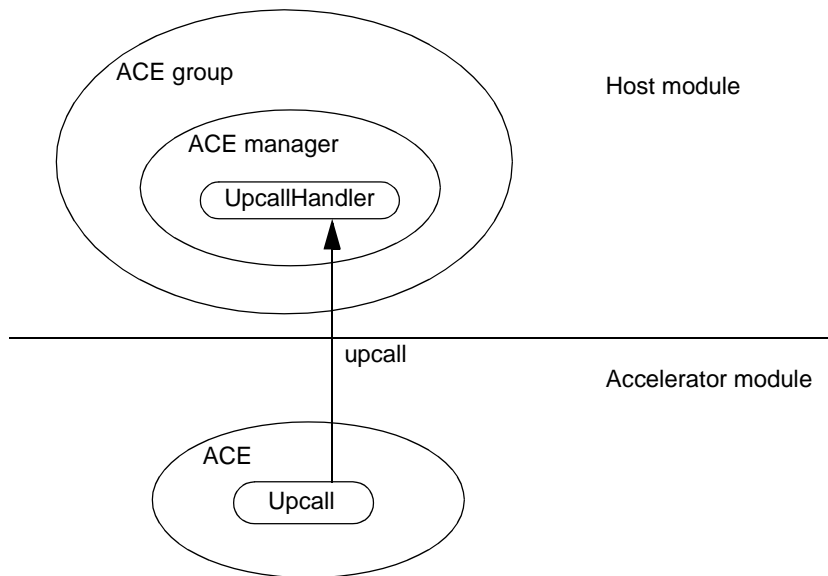
Use the `int64` conversion operator and the appropriate `Time` constructor to convert time values to scalar values. When they are converted, you can apply all normal scalar operators to `Time` objects. The compiler implicitly and transparently converts between `Time` objects and normal scalar types.

Upcall Class

Use this class to deliver messages from the Policy Accelerator to the host.

The `Upcall` class contains information that the Policy Accelerator requires to deliver messages from the Policy Accelerator to the proper service function in the proper application in the host.

Each `Upcall` object is associated with one `UpcallHandler` object with the same dictionary name in the host module. In response to the `Upcall` object's `call` method, the host executes the service function specified in the associated `UpcallHandler` object, passing it the specified message.



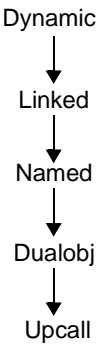
Upcalls and downcalls provide communication between Policy Accelerator actions and host functions. You use upcalls and downcalls to share information or to signal between the two modules. You can pass memory blocks, packet contents, or other messages.

The `Upcall` class contains the following methods:

Method	Description
<code>Upcall</code> Constructor	Instantiates the class.
<code>call</code> Method	Sends a message from the Policy Accelerator to an application in the host.

Class
Derivation

The `Upcall` class is derived from the `Dualobj` class.



From this class	The <code>Upcall</code> class inherits
Dynamic	Methods that enable objects to be allocated efficiently and recycled through Policy Accelerator-managed tagged memory pools.
Linked	Methods that enable objects to link to each other.
Named	Methods that enable the system to find objects by internal names.
Dualobj	The <code>ace</code> method and the semantics that enable it to be managed as an abstract object with state on both the host and Policy Accelerators.

Example

The following example is taken from the `BasicApp` demo application. The action file defines an ACE subclass, `NBBasicAce`, which contains an upcall object, and declares a method in that class that will create and send a message in an upcall:

```
class NBBasicAce : public Ace {  
public:
```

```

NBBasicAce (ModuleId id, char* name, Image* obj);
void peekPacketUpcall (Buffer *buf);
int packetCounter;
nuint32 msg;
protected:
    Upcall peekPacketUpcallHandle;
};

```

The constructor for the ACE subclass creates the upcall object as well as the ACE object:

```

NBBasicAce::NBBasicAce (ModuleId id, char* iname, Image* iobj):
    Ace (id, iname, iobj),
    peekPacketUpcallHandle (id, this, "peekPacketUpcall")
{
    packetCounter = 0;
}

```

The definition for the method that sends the upcall is as follows:

```

void NBBasicAce::peekPacketUpcall (Buffer *buf) {
    buf = buf; /* prevent "buf not used" compiler warning */
    packetCounter++;
    if ((packetCounter % 100) == 0) {
        msg = htonl (packetCounter);
        MessageBlock b ( (char *)&msg, sizeof (msg));
        Message m (b);
        peekPacketUpcallHandle.call(&m);
    }
}

```

An action function calls the message-sending method:

```

ACTNF action_all (Buffer* buf, NBBasicAce* ace) {
    ace->peekPacketUpcall (buf);
    return RULE_CONT;
}

```

See Also

- Message Class, MessageBlock Class
- UpcallHandler Class in Chapter 3, “Host API.”
- “Communication Between the Host and the Policy Accelerator” on page 104 of *Developing Applications Using the IX-API SDK*

Upcall Constructor

Creates an Upcall object.

```
Upcall (ModuleId id,
        Ace * ace,
        char * name);
```

Argument	Description
id	ACE Identifier assigned by the Resolver.
ace	Pointer to the Ace object in the Policy Accelerator.
name	The upcall's dictionary name. This must be the same as the dictionary name of the associated UpcallHandler object in the host module.

ReturnsA reference to the newly created object.

DescriptionYou cannot use the object until the Policy Accelerator receives a message from the Resolver containing addressing information for the Upcall with this name.

call Method

Sends a message from the Policy Accelerator to the host.

```
int call (Message * m);
```

Argument	Description
m	Pointer to the message to be sent.

ReturnsZero when successful, a negative number when unsuccessful.

DescriptionThe method returns asynchronously before the call has completed. The method returns -1 in the following cases:

- The upcall mechanism is not yet initialized by the Resolver
- Too many previous upcalls are waiting for processing
- The system cannot allocate host memory for the upcall

If the method does succeed, it means only that the call has been initiated, not that it has completed. Data for the message has not necessarily been copied from the source area until the call has completed.

When the call has completed, the message is passed to the corresponding `UpcallHandler` object's service function.



NOTE: Do not delete the message pointer after sending the call. Because the call is queued for asynchronous handling, the message could be deleted before the call is processed. By default, messages are automatically deleted when the call is complete. See “`MessageBlock Class`” on page 184 for information on alternative handling.

For information on creating the message argument, see “`Message Class`” on page 180 and “`MessageBlock Class`” on page 184.

Chapter 5

ASL Extensions for TCP/IP



The TCP/IP extensions to the Action Services Library (ASL) provide a set of class definitions that help with tasks common to TCP/IP-based applications. The methods span several protocol layers, and include operations such as IP fragment reassembly and TCP stream reconstruction.

This chapter has the following sections:

- Classes and Constants in the ASL TCP/IP Extensions
- Using the TCP/IP Classes
- IP Constant Definitions
- TCP Constant Definitions
- ASL TCP/IP Extension API

Classes and Constants in the ASL TCP/IP Extensions

This section introduces the ASL extension classes by functional area. Classes and methods are described in detail in the API section of this chapter, in alphabetical order.

General Checksum Support

The Internet checksum is used extensively within the TCP/IP protocols to provide reasonably high assurance that data has been delivered correctly. In particular, it is used by IP (for headers), TCP and UDP (for headers and data), ICMP (for headers and data), and IGMP (for headers). The ASL extensions provide one class for accessing checksums in a variety of protocols.

Class	Description
Internet Class	Computes checksums for various internet protocols.



IP Support

The ASL extensions provide a set of classes and constants that you use to process IP-layer data. These classes support fragmentation and reassembly of IP datagrams.

For address translation in IP, see “Network Address Translation (NAT)” on page 281.

For information on constants, see “IP Constant Definitions” on page 293.

Class	Description
IP4Addr Class	Manipulates 32-bit IP version 4 addresses.
IP4Header Class	Manipulates IP version 4 headers.
IP4Mask Class	Manipulates IP version 4 masks.
ReassemblyQueue Class	Reassembles IP version 4 fragments.
IP4Fragment Class	Manipulates IP version 4 fragments.
IP4Datagram Class	Manipulates IP version 4 datagrams.

UDP Support

The ASL extensions provide a class that you use to process UDP-layer data.

Class	Description
UDPHeader Class	Manipulates UDP headers.

TCP Support

The TCP protocol provides a connection-oriented stream service with state.

The `NBtcp.h` file contains TCP-specific definitions, including the TCP header, plus a facility to monitor the content and progress of an active TCP flow as a third party (that is, without having to be an endpoint). The classes and constants that you use to process TCP-layer data also support TCP stream reassembly and state following.

For address and port number translation of TCP, see “TCPNat Base Class” on page 355.

For information on states, return values, and other constants, see “TCP Constant Definitions” on page 295.

Class	Description
TCPHeader Class	Manipulates TCP headers.
TCPSegInfo Class	Manipulates TCP segment information.
TCPSeq Class	Manipulates TCP sequence numbers.
TCPEndpoint Class	Manipulates TCP endpoints.
TCPSession Class	Manipulates TCP sessions.

**Network
Address
Translation
(NAT)**

Network Address Translation (NAT) refers to the ability to modify various fields of different protocols so that the effective source, destination, or source and destination entities are replaced by an alternative.

In the ASL, the file `NBnat.h` contains the definitions to perform NAT for the IP, UDP, and TCP protocols. The NAT implementation uses incremental checksum computations, so performance should not degrade in proportion to packet size.

The ASL NAT classes can translate the following:

- source/destination IP addresses
- source/destination UDP port numbers
- source/destination TCP port and sequence numbers
- source/destination TCP port, sequence, and acknowledgement numbers.

The NAT methods modify the specified fields, and also rewrite any checksums that become outdated. For IP, NAT rewrites the IP header checksum. For UDP and TCP, NAT also rewrites the pseudoheader checksum, which covers the IP header source and destination addresses plus protocol fields.



NOTE: The NAT classes do not translate network addresses that might be embedded in a data portion of the packets (like those found in certain higher-layer protocols such as FTP and DNS). You are responsible for translating these addresses.

NAT classes for the IP protocol include the following:

Class	Description
IP4NAT Base Class	This pure virtual class is a base from which the usable IP NAT subclasses are derived.
IP4SNat Class	Modifies the source IP addresses in an IP packet.
IP4DNat Class	Modifies the destination IP addresses in an IP packet.
IP4SDNat Class	Modifies both the source and destination IP addresses in an IP packet.

NAT classes for the UDP protocol include the following:

Class	Description
UDPNat Base Class	This pure virtual class is a base from which the usable UDP NAT subclasses are derived.
UDPSNat Class	Modifies the source IP addresses and (optionally) ports in a UDP packet.
UDPDNat Class	Modifies the destination IP addresses and (optionally) ports in a UDP packet.
UDPSDNat Class	Modifies both the source and destination IP addresses and (optionally) ports in a UDP packet.

NAT classes for the TCP protocol include the following:

Class	Description
TCPNat Base Class	This pure virtual class is a base from which the usable TCP NAT subclasses are derived.
TCPSNat Class	Modifies the source IP addresses and (optionally) ports and/or sequence number in a TCP packet.
TCPDNat Class	Modifies the destination IP addresses and (optionally) ports and/or ACK number in a TCP packet.
TCPSDNat Class	Modifies both the source and destination IP addresses and (optionally) ports and/or sequence and ACK number in a TCP packet.

Using the TCP/IP Classes

This section gives examples of how to use the TCP/IP extensions in your applications.



NOTE: The methods in the TCP/IP extensions generally use network byte order. When you use a host with a different native byte order (for example, a Pentium or PC-compatible machine) you must convert values between host and network byte order as needed. See “Byte Order and Intermodule Communication” in Chapter 2, “System Types and Methods.”

Using Header Classes

The following simple example shows how you can use the header classes to access parts of a packet, or find embedded packets. Notice that the header classes contain static methods which you can use directly. You do not need to instantiate these classes.

Suppose your application has the following rule in the NCL file:

```
rule { tcp } { handleTCP (tcp) }
```

Your action file would contain an action function definition in the following form:

```
ACTFN handleTCP (Buffer* buf, NBBasicAce* ace,
                  TCPHeader* pTcpHdr)
{
    // access fields
    uint16 sport = pTcpHdr->sport ();
    uint16 dport = pTcpHdr->dport ();
    // body of function ...
}
```

Similarly, you could use the IP header class to access the TCP protocol contained in the IP protocol, as follows:

```
rule { ip } { handleIP (ip) }

ACTFN handleIP (Buffer* buf, NBBasicAce* ace,
                 IP4Header* pIpHdr)
{
    // access TCP portion of packet
    TCPHeader* pTcpHdr = (TCPHeader) pIpHdr->payload ();
    // body of function ...
}
```

Using Header Classes and NAT

The following example shows how to use the protocol and NAT classes in an action function to perform network address translation. Notice that the NAT classes, like the header classes, contain static methods that you can use directly. You do not need to instantiate these classes.

```
typedef struct pseudo_hdr // Pseudo header for TCP and
{
    // UDP checksum computations:
    uint32 srcAddr; // Source IP address.
    uint32 dstAddr; // Dest IP address.
    uint8 zero; // A zero byte.
    uint8 protocol; //Protocol from IP header.
    uint16 length; // UDP/TCP length.
    uint16 srcPort; // UDP/TCP src port.
    uint16 dstPort; // UDP/TCP dst port.
} pseudo_hdr;

//-----
// NatNetworkFlow
// Purpose: 1. Perform NAT on a network flow.
//          2. Forward the NATed packet.
//-----
ACTNF NatNetworkFlow( // NAT a Network Flow we control:
    Buffer* pBuf, // In: Ptr to packet buffer.
    FaAce* pAce, // In: Ptr to Forwarding Agent ACE.
    IP4Header* pIp4Hdr, // In: Ptr to IP4 header; w/datagram.
    Affinity* pAffinity // In: Ptr to an Affinity for the flow.
{
    DbgTrace ("NatNetworkFlow");
    Dbg (cout << "\tIP src: " << pIp4Hdr->src () << " IP dst: "
    <<
        pIp4Hdr->dst() << endl);
    Dbg (cout << "pBuf: " << *pBuf << endl);
    nuint16* pData;
    nuint16 olddata_checksum;
    nuint16 newdata_checksum;
    IP4Addr newSrcAddr, newDstAddr;
    nuint16 newSrcPort, newDstPort;
    nuint16 oldSrcPort, oldDstPort;
    bool adjustChecksum;
    uint32 fwdToAddr;
    TCPHeader* pTcpHdr = NULL;
    UDPHeader* pUdpHdr = NULL;
    IP4Addr oldSrcAddr = pIp4Hdr->src ();
    IP4Addr oldDstAddr = pIp4Hdr->dst ();
    uint16 segLength = 0;
    uint8 protocol= pIp4Hdr->proto();
    ACE_RESULT result;

    //
    // Perform Network Address Translation (NAT) ...
```

```

// Perform both source and dest NAT.
// If either src or dest NAT addrs or ports are zero,
// leave them as is.
//
// Conditionally perform IP src/dst NAT ...
//
newSrcAddr = pAffinity->m_pNatInfo->srcAddr;
newDstAddr = pAffinity->m_pNatInfo->dstAddr;

if (newSrcAddr != 0 || newDstAddr != 0)
{
// Adjust the IP src/dst addr and checksum.
nuint16* pData = (nuint16*)pIp4Hdr + 6;
olddata_checksum = Internet::cksum (pData, 8 );
if (newSrcAddr != 0)
    pIp4Hdr->src () = newSrcAddr;
if (newDstAddr != 0)
    pIp4Hdr->dst () = newDstAddr;
newdata_checksum = Internet::cksum (pData, 8 );
pIp4Hdr->cksum () = Internet::incrcksum (pIp4Hdr->cksum
()),

olddata_checksum,

newdata_checksum );
}
//
// Conditionally perform TCP or UDP src/dst port NAT ...
//
switch (protocol)
{
case TCP:
    pTcpHdr = (TCPHeader*)pIp4Hdr->payload ();
    segLength = (uint16) (pIp4Hdr->datalen () -
                        pIp4Hdr->hlen ());

    oldSrcPort = pTcpHdr->sport ();
    oldDstPort = pTcpHdr->dport ();
    newSrcPort = pAffinity->m_pNatInfo->srcPort;
    newDstPort = pAffinity->m_pNatInfo->dstPort;
    adjustChecksum = true;
    break;

case UDP:
    pUdpHdr = (UDPHeader*)pIp4Hdr->payload ();
    segLength = pUdpHdr->len ().raw_;
    oldSrcPort = pUdpHdr->sport ();
    oldDstPort = pUdpHdr->dport ();
    newSrcPort = pAffinity->m_pNatInfo->srcPort;
    newDstPort = pAffinity->m_pNatInfo->dstPort;

```

```

        adjustChecksum = (pUdpHdr->cksum () != 0);
        break;

    default:
        ASSERTFAIL ( "NatNetworkFlow - Unsupported protocol"
);
    case ICMP:
        newSrcPort = 0;
        newDstPort = 0;
        adjustChecksum = false;
        break;
}

if (newSrcPort != 0 || newDstPort != 0)
{ // Adjust the src/dst port and conditionally the checksum.
    pData = (nuint16*)pIp4Hdr->payload ();
    if (newSrcPort != 0)
        *pData = newSrcPort;
    if (newDstPort != 0)
        * (pData+1) = newDstPort;
}

if (adjustChecksum)
{
    pseudo_hdr old_pseudo_hdr;
    pseudo_hdr new_pseudo_hdr;

// Ensure seg length is a multiple of 16 bits.
    if (segLength & 0x0001) segLength++;

// Setup 'old' pseudo header.
    old_pseudo_hdr.srcAddr = oldSrcAddr.raw_;
    old_pseudo_hdr.dstAddr = oldDstAddr.raw_;
    old_pseudo_hdr.zero = 0;
    old_pseudo_hdr.protocol = protocol;
    old_pseudo_hdr.length = segLength;
    old_pseudo_hdr.srcPort = oldSrcPort.raw_;
    old_pseudo_hdr.dstPort = oldDstPort.raw_;

// Setup 'new' pseudo header.
    new_pseudo_hdr.srcAddr = (newSrcAddr.raw_) ?
        newSrcAddr.raw_ : oldSrcAddr.raw_;
    new_pseudo_hdr.dstAddr = (newDstAddr.raw_) ?
        newDstAddr.raw_ : oldDstAddr.raw_;
    new_pseudo_hdr.zero = 0;
    new_pseudo_hdr.protocol = protocol;
    new_pseudo_hdr.length = segLength;
    new_pseudo_hdr.srcPort = (newSrcPort.raw_) ?
        newSrcPort.raw_ : oldSrcPort.raw_;

```

```

new_pseudo_hdr.dstPort = (newDstPort.raw_ ?
                          newDstPort.raw_ : oldDstPort.raw_);

olddata_checksum = Internet::cksum ((nuint16*)
                                   &old_pseudo_hdr,
                                   sizeof (old_pseudo_hdr));
newdata_checksum = Internet::cksum ((nuint16*)
                                   &new_pseudo_hdr,
                                   sizeof (new_pseudo_hdr));

if (protocol == TCP)
    pTcpHdr->cksum () = Internet::incrcksum (pTcpHdr-
>cksum(),

olddata_checksum,

newdata_checksum);
else
    pUdpHdr->cksum () = Internet::incrcksum(pUdpHdr->cksum(),

olddata_checksum,

newdata_checksum);
}
//
// Forward the packet ...
//
    ForwardPacket( pBuf, pAce );
//
// Set result to RULE_REWROTE - sent modified buffer to a
target.
//
    result = RULE_REWROTE;
    return result;
} /* NatNetworkFlow */

```

Using IP Datagram Classes

The following example illustrates how to use the IP datagram classes to assemble IP fragments.

```

class FragQueue : public Elt_FragQueue
// Fragment hold/reassembly queue:
{
public:
    FragQueue(           // Constructor:
        IP4Header* pIP4Hdr // In: Ptr to IP4 header, w/frag.
    )
    : Elt_FragQueue (pIP4Hdr->src (), pIP4Hdr->dst (),
                    pIP4Hdr->id ()), m_pDatagram (NULL)

```

```

    {
        m_pDatagram = new IP4Datagram();
        if (!m_pDatagram)
        {
            ASSERTFAIL( "FragQueue - Failure creating IP4Datagram" );
        }

        // Expiration set to 10 sec, or 1 msec to clean-up due to
        // failure.
        expire((m_pDatagram) ? Time::secs(10) : Time::usec(1),
            (ExpireMFp)&Expired);
    }

    virtual ~FragQueue(void) // Destructor:
    {
        // Deleting a datagram deletes any fragments within,
        // and releases fragment buffers for subsequent
        // classification and processing.
        //
        if (m_pDatagram)
            delete m_pDatagram;
    }

    void Expired(void) // Called when frag queue entry expires.
    { delete this; } // Delete self.

    bool InsertFrag (IP4Fragment* pFrag)
    // Insert frag into hold/reassembly datagram:
    // In: Ptr to fragment.
    {
        ASSERT(m_pDatagram);
        bool OK = (m_pDatagram && pFrag && pFrag->hdr ()) ?
            (m_pDatagram->insert (pFrag) == 0) : false;
        if (!OK)
        {
            ASSERTFAIL ("InsertFrag-Failure inserting IP4Fragment");
            if (pFrag)
                delete pFrag;
        }
    }

    // Expiration reset to 10 sec, or 1 msec to clean-up
    // due to failure.
    expire( (OK) ? Time::secs (10) : Time::usec (1),
        (ExpireMFp)&Expired);
    return OK;
}

// Coalesce/Join datagram frags into a single un-fragmented
// packet
bool CoalesceFrag(void)
{
    IP4Header* pIp4Hdr;

```



```

size_t dataLen;
bool allOK= true;
char* pAppend = NULL;
IP4Fragment* pFrag = m_pDatagram->head();
Buffer* Buf = pFrag->buf();
size_t totalLen= sizeof (IP4Header) + pFrag->datalen ();

// Append to the first fragment's buffer
// (if sufficient buffer space available)
// the payload of subsequent datagram frags.
//
while ((pFrag = pFrag->next ()))
{
    if (allOK)
    { // Append frag's payload to 1st buffer.
        dataLen  = pFrag->datalen ();
        totalLen += dataLen;
        pAppend = pBuf->append ( dataLen );
        if (pAppend)
            memcpy ( pAppend, pFrag->payload (), dataLen );
        else
            allOK = false;
    }
// Expire/delete current frag's buffer.
    pFrag->buf ()->decref ();
}

if (allOK)
{ // Reset IP header flags, packet len, and header
checksum.
    pFrag = m_pDatagram->head();
    pIp4Hdr = pFrag->hdr ();
    pIp4Hdr->offset ()= 0;
    pIp4Hdr->len () = htons ((short)totalLen );
    pIp4Hdr->cksum () = 0;
    pIp4Hdr->cksum () = Internet::cksum
((uint16*)pIp4Hdr,
                                sizeof
(IP4Header) );
}
else
{
    ASSERTFAIL ("CoalesceFrag-Failure joining/appending
                frags" );
// Expire/delete first frag's buffer.
    pBuf->decref ();
}

```

```

        return allOK;
    }

    bool AllFragmentsReceived (void)
    {
        ASSERT(m_pDatagram);
        return (m_pDatagram) ? m_pDatagram->complete() : true;
    }
protected:
    IP4Datagram*m_pDatagram; // Ptr to IP datagram.
};
//-----
// Fragmented
// Purpose: Return whether or not an IP packet is a fragment.
//-----
static inline bool Fragmented (IP4Header* pIp4Hdr )
{ // Return whether or not an IP packet is fragmented
    bool isFrag, moreFragments;
    uint16 ipFlagsAndOffset = ntohs (pIp4Hdr->offset ());
    uint16 ipFragOffset = ipFlagsAndOffset & IP_OFFMASK;

    if (ipFlagsAndOffset & IP_DF)
    {
        isFrag = moreFragments = false;
    }
    else
    {
        moreFragments = ipFlagsAndOffset & IP_MF;
        isFrag = (moreFragments || ipFragOffset);
    }
    return isFrag;
} /* Fragmented */

//-----
// HandleFrag
// Purpose:
//      1. Handle a fragmented IP packet;
//      in a sequence of IP fragments.
//      2. Queue and defer processing of IP fragments.
//      3. If all of the fragments composing an IP datagram
//      have been received/queued, coalesce the fragments
//      into a non-fragmented datagram.
//      Then delete and/or release fragment buffers.
//-----
ACTNF HandleFrag ( // Handle a fragmented IP packet:
    Buffer* pBuf,           // In: Ptr to packet buffer.
    FaAce* pAce,           // In: Ptr to Forwarding Agent ACE.
    IP4Header* pIp4Hdr // In: Ptr to IP4 header; w/fragment.
{

```

```

DbgTrace ("HandleFrag");
Dbg (cout << "\tIP4Header: " << pIp4Hdr << endl);

FragQueue* entry;
ACE_RESULT result = RULE_CONT;

do
{
    // If the buffer is busy, defer processing it.
    if (pBuf->busy()){
        result = RULE_DEFER;
        break;
    }
    // Ensure packet is a fragment.
    if (!Fragmented( pIp4Hdr ))
    {
        ASSERTFAIL( "HandleFrag - Not an IP fragment" );
        break;
    }
    // Check frag queue for sibling frags.
    Search fragQueue = pAce->m_FragQueue.locate(pIp4Hdr-
>src(),

pIp4Hdr->dst(),

pIp4Hdr->id() );
    if (fragQueue.hit ())
    {
        // Sibling frags exist. Get the frag queue
        // entry (which houses sibling frags) into
        // which the current frag is to be inserted.
        entry = (FragQueue*)fragQueue.toElement();
        if (entry == NULL)
        {
            ASSERTFAIL( "HandleFrag - Failure locating frag
                        queue set entry" );

            break;
        }
    }
    else
    {
        // No sibling frags in frag queue.
        // Create a new frag queue set entry.
        entry = new FragQueue( pIp4Hdr );
        if (entry == NULL)
        {
            ASSERTFAIL( "HandleFrag - Failure creating frag
                        queue set entry" );

            break;
        }
    }
}

```

```

        fragQueue.insert( entry );
    }

    // Create a frag and insert it into the frag queue.
    IP4Fragment* pFragment = new IP4Fragment (pBuf, pIp4Hdr );
    if (! (pFragment && pFragment->hdr ()))
    {
        ASSERTFAIL( "HandleFrag - Failure creating
                    IP4Fragment" );
        if (pFragment)
            delete pFragment;
        break;
    }

    if (!entry->InsertFrag( pFragment ))
    {
        ASSERTFAIL( "HandleFrag - Failure inserting entry into
                    frag queue set" );
        break;
    }

    // If all frags received, then we're done;
    // reassemble and dispatch a non-fragmented packet.
    // Otherwise set the result to retain the frag buffer.

    // If all frags have been received ...
    if (entry->AllFrgsReceived ())
    {
        // Coalesce/join datagram frags
        // into a single non-fraged packet.
        entry->CoalesceFrgs();
        // Delete the frag queue entry; doing
        // so deletes queued frags, and releases
        // their buffers for subsequent processing.
        delete entry;
    }
    result = RULE_DONE;
} while (false);
return result;
} /* HandleFrag */

```

IP Constant Definitions

In addition to the IP header itself, a number of definitions are provided for manipulating fields of the IP header with specific semantic meanings. The IP constants fall into the following categories:

- IP Fragmentation
- IP Service Type
- IP Precedence
- IP Option Definitions
- IP Options Field Offsets

IP Fragmentation

The following constants define flags that control fragmentation of IP datagrams:

Define	Value	Description
IP_DF	0x4000	Do not fragment flag (RFC 791, p. 13)
IP_MF	0x2000	More fragments flag (RFC 791, p. 13)
IP_OFFMASK	0x1FFF	Mask for determining the fragment offset from the IP header offset field
IP_MAXPACKET	65535	Maximum IP datagram size

IP Service Type

The following constants define the IP type of service byte (not commonly used):

Define	Value	Reference
IPTOS_LOWDELAY	0x10	RFC 791, p. 12
IPTOS_THROUGHPUT	0x08	RFC 791, p. 12
IPTOS_RELIABILITY	0x04	RFC 791, p. 12
IPTOS_MINCOST	0x02	RFC 1349

IP Precedence The following constants define IP precedence. All are from RFC 791, p. 12 (not widely used):

Define	Value
IPTOS_PREC_NETCONTROL	0xE0
IPTOS_PREC_INTERNETCONTROL	0xC0
IPTOS_PREC_CRITIC_ECP	0xA0
IPTOS_PREC_FLASHOVERRIDE	0x80
IPTOS_PREC_FLASH	0x60
IPTOS_PREC_IMMEDIATE	0x40
IPTOS_PREC_PRIORITY	0x20
IPTOS_PREC_ROUTINE	0x00

IP Option Definitions The following constant definitions support IP options. All definitions are from RFC 791, pp. 15-23.

Define	Value	Description
IPOPT_COPIED(o)	((o)&0x80)	A macro that returns TRUE if the option 'o' is to be copied upon fragmentation
IPOPT_CLASS(o)	((o)&0x60)	A macro giving the option class for the option 'o'
IPOPT_NUMBER(o)	((o)&0x1F)	A macro giving the option number for the option 'o'
IPOPT_CONTROL	0x00	Control class
IPOPT_RESERVED1	0x20	Reserved
IPOPT_DEBMEAS	0x40	Debugging and/or measurement class
IPOPT_RESERVED2	0x60	Reserved
IPOPT_EOL	0	End of option list
IPOPT_NOP	1	No operation
IPOPT_RR	7	Record packet route

Define	Value	Description
IPOPT_TS	68	Time stamp
IPOPT_SECURITY	130	Provide s, c, h, tcc
IPOPT_LSRR	131	Loose source route
IPOPT_SATID	136	Satnet ID
IPOPT_SSRR	137	Strict source route
IPOPT_RA	148	Router alert

IP Options Field Offsets

The following constants define the offsets to fields in options other than EOL and NOP.

Define	Value	Description
IPOPT_OPTVAL	0	Option ID
IPOPT_OLEN	1	Option length
IPOPT_OFFSET	2	Offset within option
IPOPT_MINOFF	4	Minimum value of offset

TCP Constant Definitions

A number of constant definitions are provided for use with TCP. These fall into the following categories:

- TCP Control Bits
- TCP Options
- TCP Session State
- TCP Return Codes

TCP Control Bits

The following constants define the standard TCP control bits:

Define	Value	Description
TH_FIN	0x01	TCP FIN indication (closing connection)
TH_SYN	0x02	TCP SYN indication (synchronize connection, sequence number init)
TH_RST	0x04	Connection reset
TH_PUSH	0x08	TCP “push” indication (often indicates segment emptied send buffer)
TH_ACK	0x10	TCP ACK field value valid indicator
TH_URG	0x20	TCP Segment contains urgent data

TCP Options

The following constants define TCP options:

Define	Value	Description
TCPOPT_EOL	0	End of option list
TCPOPT_NOP	1	No operation (used for padding)
TCPOPT_MAXSEG	2	Maximum segment size (MSS)
TCPOPT_SACK_PERMITTED	4	Selective Acknowledgements available
TCPOPT_SACK	5	Selective Acknowledgements in this segment
TCPOPT_TIMESTAMP	8	Time stamps
TCPOPT_CC	11	For T/TCP (see RFC 1644)
TCPOPT_CCNEW	12	For T/TCP
TCPOPT_CCECHO	13	For T/TCP

TCP Session State

TCP operates as an 11-state finite state machine. Most of the states are related to connection establishment and tear-down. By following certain control bits in the TCP headers of segments passed along a connection, it is possible to infer the TCP state at each endpoint of a TCP connection and to monitor the data exchanged between them. The TCP/IP ASL extensions provide a facility to

follow TCP state transitions and also to inspect data exchanged across a TCP connection. The facility is designed to handle TCP segments in fragmented IP datagrams, if necessary.



NOTE: States less than `TCPS_ESTABLISHED` indicate connections not yet established.

States greater than `TCPS_CLOSE_WAIT` are those where the user has closed.

States greater than `TCPS_CLOSE_WAIT` and less than `TCPS_FIN_WAIT_2` indicate TCP is awaiting an ACK of FIN.

The following constants indicate states in the TCP finite state machine:

Define	Value	Description
<code>TCPS_CLOSED</code>	0	Closed
<code>TCPS_LISTEN</code>	1	Listening for connection
<code>TCPS_SYN_SENT</code>	2	Active open, have sent SYN
<code>TCPS_SYN_RECEIVED</code>	3	Have sent and received SYN
<code>TCPS_ESTABLISHED</code>	4	Established
<code>TCPS_CLOSE_WAIT</code>	5	Received FIN, waiting for closed
<code>TCPS_FIN_WAIT_1</code>	6	Have closed, sent FIN
<code>TCPS_CLOSING</code>	7	Closed exchanged FIN; awaiting FIN ACK
<code>TCPS_LAST_ACK</code>	8	Had FIN and close; await FIN ACK
<code>TCPS_FIN_WAIT_2</code>	9	Have closed, FIN is ACKed
<code>TCPS_TIME_WAIT</code>	10	In 2*MSL quiet wait after close

The following predicates on states allow classification of state values:

Define	Description
TCPS_HAVERCVDSYN (<i>s</i>)	TRUE if state <i>s</i> is one for which a SYN has been received—that is, <i>s</i> >= TCPS_SYN_RECEIVED
TCPS_HAVEESTABLISHED (<i>s</i>)	TRUE if state <i>s</i> is one for which a connection has been established— that is, <i>s</i> >= TCPS_ESTABLISHED
TCPS_HAVERCVDFIN (<i>s</i>)	TRUE if state <i>s</i> is one for which a FIN has been received—that is, <i>s</i> >= TCPS_TIME_WAIT

TCP Return Codes

The following definitions are for TCP stream reassembly and indicate to the caller the disposition of TCP segments processed by the state following and stream reassembly methods described below.

The following codes indicate high-level conditions (good, bad, and state change):

Define	Value	Description
TSEG_NORMAL	0x0	Normal
TSEG_BAD	0x1	Bad TCP segment (bad ACK field, for example)
TSEG_STATE_CHANGE	0x2	Segment changed TCP state machine
TSEG_QUEUED	0x4	Segment was queued, caller loses ownership

The following codes indicate specific errors:

Define	Value	Description
TSEG_BADLEN	0x10	Bad length
TSEG_BADSUM	0x20	Segment contained bad TCP checksum
TSEG_BADACK	0x40	Bad ACK field (too low or high)
TSEG_NOSYN	0x80	Expected a SYN but did not receive one

The following codes indicate conditions that might be interesting to applications but that are not necessarily errors:

Define	Value	Description
TSEG_FRAG	0x1000	TCP segment was fragmented (but ok)
TSEG_RESET	0x2000	Segment contained an RST indication
TSEG_SYNDATA	0x4000	SYN contained data
TSEG_CLOSE	0x8000	Segment caused state transition to CLOSED
TSEG_HDRLOC	0x10000	TCP header was not in first fragment
TSEG_OLAP_FRONT	0x100000	Segment's front overlapped queued data
TSEG_OLAP_END	0x200000	Segment's end overlapped queued data

ASL TCP/IP Extension API

This section provides details of each TCP/IP extension class. Classes are listed in alphabetical order. Within each class, the constructor and destructor are listed first, followed by the remaining methods in alphabetical order.

The ASL TCP/IP extension API contains the following classes:

Class	Description
Internet Class (page 302)	Computes checksums for various Internet protocols.
IP4Addr Class (page 308)	Manipulates 32-bit IP version 4 addresses.
IP4Datagram Class (page 310)	Manipulates IP version 4 datagrams.
IP4DNat Class (page 316)	Modifies the destination IP addresses in an IP packet.
IP4Fragment Class (page 318)	Manipulates IP version 4 fragments.
IP4Header Class (page 324)	Manipulates IP version 4 headers.
IP4Mask Class (page 331)	Manipulates IP version 4 masks.
IP4NAT Base Class (page 333)	Represents IP version 4 network address translation. A pure virtual base class.
IP4SDNat Class (page 335)	Modifies both the source and destination IP addresses in an IP packet.
IP4SNat Class (page 337)	Modifies the source IP addresses in an IP packet.
ReassemblyQueue Class (page 339)	Reassembles IP version 4 fragments.
TCPDNat Class (page 343)	Modifies the destination IP addresses and (optionally) ports and/or ACK number in a TCP packet.
TCPEndpoint Class (page 346)	Manipulates TCP endpoints.

Class	Description
TCPHeader Class (page 350)	Manipulates TCP headers.
TCPNat Base Class (page 355)	Represents TCP network address translation. A pure virtual base class.
TCPSDNat Class (page 359)	Modifies both the source and destination IP addresses and (optionally) ports and/or sequence and ACK number in a TCP packet.
TCPSegInfo Class (page 362)	Manipulates TCP segment information.
TCPSeq Class (page 365)	Manipulates TCP sequence numbers.
TCPSession Class (page 368)	Manipulates TCP sessions.
TCPSNat Class (page 371)	Modifies the source IP addresses and (optionally) ports and/or sequence number in a TCP packet.
UDPDNat Class (page 374)	Modifies the destination IP addresses and (optionally) ports in a UDP packet.
UDPHeader Class (page 376)	Manipulates UDP headers.
UDPNat Base Class (page 378)	Represents UDP network address translation. A pure virtual base class.
UDPSDNat Class (page 381)	Modifies both the source and destination IP addresses and (optionally) ports in a UDP packet.
UDPSNat Class (page 384)	Modifies the source IP addresses and (optionally) ports in a UDP packet.

Internet Class

The `Internet` class contains static methods that compute checksums. The Internet checksum is used extensively in the TCP/IP protocols to provide assurance that data has been delivered correctly. In particular, it is used by IP (for headers), TCP and UDP (for headers and data), ICMP (for headers and data), and IGMP (for headers).

The Internet checksum is defined to be the one's complement of the sum of a region of data, where the sum is computed using 16-bit words and one's complement addition.

Several Internet RFCs describe methods to compute this checksum:

- RFC 1936 describes a hardware implementation.
- RFC 1624 and RFC 1141 describe incremental updates.
- RFC 1071 describes a number of mathematical properties of the checksum and how to compute it quickly. It also includes a copy of IEN 45 (from 1978), which describes motivations for the design of the checksum.

RFCs are available from <http://www.rfc-editor.org>.

The methods are declared as static within this class, so you can use them without instantiating the class. Applications using these methods must include the file `NBip.h`. The class contains the following methods:

Method	Description
<code>apasum</code> Method	Computes the Internet checksum of the IP source and destination addresses and the TCP ACK field.
<code>apsasum</code> Method	Computes the Internet checksum of the IP source and destination addresses, ports, and the TCP ACK and sequence numbers.
<code>apsum</code> Method	Computes the Internet checksum of the IP source and destination addresses and the two 16-bit words, which are the port numbers for TCP and UDP, immediately following the IP header.
<code>apssum</code> Method	Computes the Internet checksum of the IP source and destination addresses, ports, and TCP sequence number.
<code>asum</code> Method	Computes the Internet checksum of the IP source and destination addresses.

Method	Description
cksum Method	Computes the Internet checksum of a data block.
incrcksum Method	Computes an Internet checksum incrementally.
psum Method	Computes the two's-complement sum of a region of data taken as 16-bit words.

apasum Method

Computes the Internet checksum of the IP source and destination addresses and the TCP ACK field.

```
static nuint16 apasum (IP4Header* hdr);
```

Parameter	Description
hdr	Pointer to the header

Returns The computed checksum.

apsasum Method

Computes the Internet checksum of the IP source and destination addresses, ports, and the TCP ACK and sequence numbers.

```
static nuint16 apsasum (IP4Header* hdr);
```

Parameter	Description
hdr	Pointer to the header

Returns The computed checksum.

apsum Method

Computes the Internet checksum of the IP source and destination addresses and the two 16-bit words, which are the port numbers for TCP and UDP, immediately following the IP header.

```
static nuint16 apsum (IP4Header* hdr);
```

Parameter	Description
hdr	Pointer to the header

Returns The computed checksum.

apssum Method

Computes the Internet checksum of the IP source and destination addresses, ports, and TCP sequence number.

```
static nuint16 apssum (IP4Header* hdr);
```

Parameter	Description
hdr	Pointer to the header

Returns The computed checksum.

asum Method

Computes the Internet checksum of the IP source and destination addresses.

```
static nuint16 asum (IP4Header* hdr);
```

Parameter	Description
hdr	Pointer to the header

Returns The computed checksum.

cksum Method

Computes the Internet checksum of a data block.

```
static nuint16 cksum (nuint16* base,  
                     int len);
```

Parameter	Description
base	The starting address of the data
len	The number of bytes of data

Returns The computed checksum.

Description This method computes the Internet checksum of the specified data block, and returns it in the same byte order as the underlying data, which is commonly network byte order.

The method works properly for data aligned to any byte boundary, but can perform significantly better for 32-bit aligned data.

incrcksum Method

Computes an Internet checksum incrementally.

```
static nuint16 incrcksum (nuint16 ocksum,
                          nuint16 odsum,
                          nuint16 ndsum);
```

Parameter	Description
ocksum	The original checksum
odsum	The checksum of the old data
ndsum	The checksum of the new (replacement) data

Returns The computed checksum.

Description This method computes a new checksum from the original checksum for a region of data, a checksum for a block of data to be replaced, and a checksum of the new data to replace the old data.

This method is especially useful when small regions of packets are modified and checksums must be updated appropriately (for example, for decrementing IP TTL fields or rewriting address fields for NAT).

psum Method

Computes the two's-complement sum of a region of data taken as 16-bit words.

```
static uint32 psum (nuint16* base,
                   int len);
```

Parameter	Description
base	The starting address of the data
len	The number of bytes of data

Returns The two's-complement sum.

Description This method computes the two's-complement 32-bit sum of the specified data, handled as an array of 16-bit words. You can generate the Internet checksum for the specified data region by folding any carry bits above the low-order 16 bits and taking the one's complement of the resulting value.

IP4Addr Class

The `IP4Addr` class represents 32-bit IP version 4 addresses. Because the class is derived from `nuint32`, you can generally handle IP addresses as 32-bit integers in network byte order.

The `IP4Addr` class is defined in `NBip.h`. The class contains the following methods:

Method	Description
<code>IP4Addr</code> Constructor	Instantiates the class.
<code>bcast</code> Method	Identifies a local network broadcast.
<code>mcast</code> Method	Identifies an IP version 4 multicast address.

IP4Addr Constructor

Instantiates the `IP4Addr` class.

```
IP4Addr (nuint32 an);  
  
IP4Addr (uint32 ah);  
  
IP4Addr ();
```

Parameter	Description
<code>an</code>	The address as an unsigned 32-bit word in network byte order
<code>ah</code>	The address as an unsigned 32-bit word in host byte order

Returns	The newly created object.
Description	This class has three constructors for creating IP (version 4) addresses with or without an initial value. Pass the initial value as an unsigned 32-bit word in either host or network byte order, or pass no parameter to create the object with no initial value.

Example The following example creates two `IP4Addr` objects, each of which refer to the IP address 128.32.12.4. Note the use of the `htonl` ASL function to convert the host 32-bit word into network byte order:

```
#include "NBip.h"
uint32 myhaddr = (128 << 24) | (32 << 16) | (12 << 8) | 4;
nuint32 mynaddr = htonl((128 << 24) | (32 << 16) | (12 << 8) | 4);
IP4Addr ip1(myhaddr);
IP4Addr ip2(mynaddr);
```

bcast Method

Identifies a local network broadcast.

```
bool bcast ();
```

Returns `TRUE` if all the address bits are 1-bits, otherwise `FALSE`.

Description This method returns a Boolean value indicating whether all of the bits in the IP address are 1, indicating a local network broadcast.

mcast Method

Identifies an IP version 4 multicast address.

```
bool mcast ();
```

Returns `TRUE` if the address is an IP version 4 multicast address, otherwise `FALSE`.

Description This method returns a Boolean value indicating whether the IP address is an IP version 4 multicast address; that is, a member of the class D address space.

IP4Datagram Class

The `IP4Datagram` class represents a collection of IP fragments, which might represent a complete IP4 datagram. This class supports the fragmentation, reassembly, and grouping of IP fragments. Objects of this class contain a doubly-linked list of `IP4Fragment` objects, sorted by IP offset.

When IP fragments are inserted into a datagram to perform reassembly, the coalescing of data between fragments is *not* performed automatically. The `IP4Datagram` object can determine whether it contains a complete set of fragments, but it does not automatically reconstruct a contiguous buffer of the original datagram's contents.

The `IP4Datagram` class is defined in `NBip.h`. The class contains the following methods:

Method	Description
<code>IP4Datagram</code> Constructor	Instantiates the class.
<code>IP4Datagram</code> Destructor	Frees the datagram object and its contained fragments.
<code>checksum</code> Method	Determines or controls whether the IP header checksum is checked during IP reassembly.
<code>complete</code> Method	Determines whether all fragments composing the original datagram are present.
<code>fragment</code> Method	Breaks a datagram into fragments.
<code>fragmented</code> Method	Determines whether the datagram is fragmented.
<code>head</code> Method	Finds the first IP fragment in the datagram.
<code>insert</code> Method	Inserts a fragment into the datagram.
<code>len</code> Method	Finds the length of the datagram.
<code>nfrags</code> Method	Determines the number of fragments currently present in the datagram.

IP4Datagram Constructor

Instantiates the IP4Datagram class.

```
IP4Datagram ();  
  
IP4Datagram (IP4Fragment* frag);
```

Parameter	Description
frag	Pointer to a doubly-linked list of fragments.

- Returns

A reference to the newly created object.
- Description

Use the constructor without parameters to create a fresh datagram; for example, to start the process of reassembly.

Pass the `frag` parameter to place an existing list of fragments into the datagram object immediately upon creation.

IP4Datagram Destructor

Frees the datagram object and its contained fragments.

```
~IP4Datagram ();
```

- Description

The destructor calls the destructors for each of the fragments composing the datagram and frees the datagram object.

checkcksum Method

Determines or controls whether the IP header checksum is checked during IP reassembly.

```
bool checkcksum ( );
```

```
void checkcksum (bool on);
```

Parameter	Description
on	TRUE if IP header checksums should be verified, FALSE if they should not be verified.

Returns When you pass no parameter, returns `TRUE` if checksum verification is enabled, otherwise `FALSE`. When you pass the parameter, returns nothing.

Description When you pass the `on` parameter, IP header checksum verification is turned on or off. When set to `TRUE`, IP header checksums are verified when a fragment is inserted into a datagram. When set to `FALSE`, these checksums are ignored.

When you pass no parameter, this method determines the current state of verification.

complete Method

Determines whether all fragments composing the original datagram are present.

```
bool complete ( );
```

Returns `TRUE` when all fragments composing the original datagram are present, otherwise `FALSE`.

fragment Method

Breaks a datagram into fragments.

```
IP4Datagram* fragment (int mtu);
```

Parameter	Description
mtu	The maximum transmission unit (MTU) size limiting the maximum fragment size

- Returns

A pointer to an `IP4Datagram` object containing a doubly-linked list of `IP4Fragment` objects.
- Description

This method breaks an IP datagram into a series of IP fragments, each of which fits in the packet size specified by `mtu`.

The method allocates `Buffer` objects to contain the newly-formed IP fragments and links them together. It returns the head of the doubly-linked list of fragments. The original datagram is not affected.

The original datagram object (the one fragmented) is **not** freed by this method. You are responsible for freeing the original fragment when it is no longer needed.
- See Also

`IP4Fragment` Class, `fragment` Method

fragmented Method

Determines whether the datagram is fragmented.

```
bool fragmented ();
```

- Returns

`TRUE` if the datagram is fragmented (whether or not it is complete), otherwise `FALSE`.

head Method

Finds the first IP fragment in the datagram.

```
IP4Fragment* head ();
```

Returns The address of the first IP fragment in the datagram's linked list of fragments.

insert Method

Inserts a fragment into the datagram.

```
int insert (IP4Fragment* frag);
```

Parameter	Description
frag	Pointer to the fragment being inserted

Returns Zero on success. On failure, a 32-bit code where each bit indicates an outcome of the insertion attempt (see Description).

Description This method attempts to reassemble the overall datagram by checking the IP offset and ID fields.

The return code is a 32-bit word where each bit indicates a different error or unusual condition. The first bit, `IPD_INSERT_ERROR`, is set whenever any of the other conditions are encountered.

Code Bit	Error Condition
<code>IPD_INSERT_ERROR</code>	One of the errors occurred.
<code>IPD_INSERT_OH</code>	Head overlapped. This flag is set if the front of the new fragment is overlapped by data already inserted into the datagram. In this case the new data is truncated and the data already inserted is used.

Code Bit	Error Condition
IPD_INSERT_OT	Tail overlapped. This flag is set when data at the end of the new fragment overlaps data that has already been inserted into the fragment. In this case, the new data is used and the data that was already in the buffer is trimmed.
IPD_INSERT_CKFAIL	IP header checksum failed (if enabled).

✓ **len Method**

Finds the length of the datagram.

```
int len ();
```

Returns The length of the entire datagram in bytes, including all contained fragments.

Description If the datagram contains multiple fragments, only the size of the first fragment header is included in the returned value. The value is only meaningful if the datagram is complete.

nfrags Method

Determines the number of fragments currently present in the datagram.

```
int nfrags ();
```

Returns The number of fragments.

IP4DNat Class

The IP4DNat class is derived from the IP4Nat class. It defines the class of objects implementing destination rewriting for IP datagrams and fragments.

The IP4DNat class is defined in NBnat.h. The class contains the following methods:

Method	Description
IP4DNat Constructor	Instantiates the class.
rewrite Method	Rewrites the destination addresses in an IP datagram or fragment.

IP4DNat Constructor

Creates anIP4DNat object.

```
IP4DNat (IP4Addr* newdst);
```

Parameter	Description
newdst	Pointer to the new destination address for IP NAT.

Returns A reference to the newly created object.

rewrite Method

Rewrites the destination addresses in an IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite.

Returns Nothing.

Description This method replaces the destination addresses in the specified datagram or fragment with the address specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- When the method is applied to a fragment, it affects only the specified fragment.

IP4Fragment Class

The IP protocol adapts its datagram size using an operation known as fragmentation. Fragmentation divides an initial, possibly large, IP datagram into a sequence of IP fragments. Each fragment is handled as an independent packet until it is received and reassembled at the original datagram's ultimate destination.

The `IP4Fragment` class represents a single IP packet containing an IP header, which might not be complete. A datagram or fragment is *complete* if it represents or contains all the fragments necessary to represent an entire IP-layer datagram. See also `IP4Datagram` Class, which represents a collection of fragments.

Conventional IP routers never reassemble fragments but instead route them independently, leaving the destination host to reassemble them. In some circumstances, however, for applications running on the Policy Accelerator, you might want to reassemble fragments (for example, to simulate the operation of the destination host). See `ReassemblyQueue` Class.

The `IP4Fragment` class is defined in `NBip.h`. The class contains the following methods:

Method	Description
<code>IP4Fragment</code> Constructor	Instantiates the class.
<code>IP4Fragment</code> Destructor	Frees the fragment.
<code>hdr</code> Method	Retrieves the address of the IP header at the beginning of the fragment.
<code>buf</code> Method	Retrieves the <code>Buffer</code> object containing the IP fragment.
<code>complete</code> Method	Determines whether the fragment represents a complete IP datagram.
<code>datalen</code> Method	Retrieves the number of bytes in the IP payload portion of the IP fragment.
<code>first</code> Method	Determines whether the fragment is the first fragment of a datagram.
<code>fragment</code> Method	Breaks a complete IP datagram fragment into further fragments.

Method	Description
next Method	Finds the next fragment in the list.
optcopy Method	Copies options from one header to another during IP fragmentation.
payload Method	Finds the payload in the IP fragment.
prev Method	Finds the previous fragment in the list.

IP4Fragment Constructor

Creates an IP4Fragment object.

```
IP4Fragment (Buffer* bp, IP4Header* iph);  
  
IP4Fragment (int maxiplen, IP4Header* iph);
```

Parameter	Description
bp	Pointer to the Buffer object containing the IP fragment.
maxiplen	The maximum size of the fragment being created; used to size the allocated Buffer object.
iph	Pointer to the IP4 header in the fragment. Default is 0.

Returns A reference to the newly created object.

Description When you process IP fragments in ACE action code, you pass the `Buffer` object containing the IP fragment to the constructor for the fragment object. The second argument is the location of the IP header within the buffer.

When you create IP fragments during IP datagram fragmentation, you pass a size parameter instead of the `Buffer` object. This form of the constructor allocates a new `Buffer` object and copies the IP header pointed to by `iph` into the new buffer. (If the specified header contains IP options, this constructor copies only those options that should be copied during fragmentation.)

See Also `Buffer` Class

IP4Fragment Destructor

Frees the fragment.

```
~IP4Fragment ();
```

hdr Method

Retrieves the address of the IP header at the beginning of the fragment.

```
IP4Header* hdr ();
```

Returns A pointer to the IP header.

buf Method

Retrieves the `Buffer` object containing the IP fragment.

```
Buffer* buf ();
```

Returns A pointer to the `Buffer` object, or `NULL` if no buffer is associated with the fragment.

complete Method

Determines whether the fragment represents a complete IP datagram.

```
bool complete ();
```

Returns `TRUE` when the fragment is complete, otherwise `FALSE`.

Description This method determines whether the fragment represents a complete IP datagram. The fragment is complete when the fragment offset field is zero and there are no additional fragments.

datalen Method

Retrieves the number of bytes in the IP payload portion of the IP fragment.

```
int datalen ();
```

Returns The number of bytes in the IP payload.

first Method

Determines whether the fragment is the first fragment of a datagram.

```
bool first ();
```

Returns TRUE when the fragment is the first fragment of a datagram, otherwise FALSE.

fragment Method

Breaks a complete IP datagram fragment into further fragments.

```
IP4Datagram* fragment (int mtu);
```

Parameter	Description
mtu	The maximum transmission unit (MTU) size limiting the maximum fragment size

Returns A pointer to an IP4Datagram object containing a doubly-linked list of IP4Fragment objects.

Description This method breaks a complete IP fragment into a series of smaller IP fragments, each of which fits in the packet size specified by mtu.

 The method allocates Buffer objects to contain the newly-formed IP fragments and links them together. It returns the head of the doubly-linked list of fragments. The original fragment is not affected.

The original fragment object (the one fragmented) is *not* freed by this method. You are responsible for freeing the original fragment when it is no longer needed.

next Method

Finds the next fragment of a doubly-linked list of fragments.

```
IP4Fragment*& next ();
```

Returns A reference to the internal linked-list pointer.

Description Use this method to link together fragments when they are reassembled (in datagrams) or queued. Fragments are linked together in a doubly-linked list fashion with `NULL` pointers indicating the endpoints in the list.

optcopy Method

Copies options from one header to another during IP fragmentation.

```
static int optcopy (IP4Header* src,
                   IP4Header* dst);
```

Parameter	Description
src	Pointer to the source IP header containing options
dst	Pointer to the destination, into which the source header should be copied

Returns The number of bytes of options present in the destination IP header.

Description Use this static method to copy options from one header to another during IP fragmentation. The method copies only those options that are supposed to be copied during fragmentation (that is, those options *x* where the macro `IPOPT_COPIED(x)` is `TRUE`).

payload Method

Finds the payload in the IP fragment.

```
unsigned char * payload ();
```

Returns Returns the address of the payload.

Description This method finds and returns the address of the first byte of data in the IP fragment following the header and options.

prev Method

Finds the previous fragment in the list.

```
IP4Fragment*& prev ();
```

Returns A reference to the internal linked-list pointer.

Description Use this method to link together fragments when they are reassembled (in datagrams) or queued. Fragments are linked together in a doubly-linked list fashion with `NULL` pointers indicating the endpoints in the list.

IP4Header Class

The `IP4Header` class represents the standard IP version 4 header, where sub-byte-sized fields have been merged to reduce byte-order dependencies. In addition to the standard field accessors, this class includes methods that compute other information about the header.

The methods are declared as static within this class, so you can use them without instantiating the class. Use the `IP4Header` class to access IP headers received in live network packets.

The class is defined in `NBip.h`, and contains the following methods:

Method	Description
<code>cksum</code> Method	Accesses the IP checksum.
<code>datalen</code> Method	Computes the length of the payload portion of the IP packet.
<code>dst</code> Method	Accesses the IP destination address.
<code>hl</code> Method	Computes or modifies the length of the IP header.
<code>hlen</code> Method	Computes the number of bytes in the IP header, including options.
<code>id</code> Method	Accesses the IP identifier.
<code>len</code> Method	Accesses the datagram or fragment length.
<code>offset</code> Method	Accesses the fragmentation flags and fragment offset.
<code>optbase</code> Method	Computes the location of the first IP option in the IP header.
<code>payload</code> Method	Computes the address of the first byte of payload data in the IP packet.
<code>proto</code> Method	Accesses the IP protocol value.
<code>psum</code> Method	Computes the IP value for a pseudo-checksum.
<code>src</code> Method	Accesses the IP source address.
<code>tos</code> Method	Accesses the IP type-of-service value.
<code>ttl</code> Method	Accesses the IP time-to-live value.

Method	Description
ver Method	Retrieves or assigns the version number of the IP header.
vhl Method	Accesses version and header lengths.

cksum Method

Accesses the IP checksum.

```
uint16& cksum ();
```

Returns A reference to the IP checksum.

datalen Method

Computes the length of the payload portion of the IP packet.

```
uint16 datalen ();
```

Returns The number of bytes in the payload portion of the IP packet.

Description This method computes and returns the number of bytes in the payload portion of the IP packet. It computes this value by subtracting the IP header length from the IP length field. The value can be zero.

dst Method

Accesses the IP destination address.

```
IP4Addr& dst ();
```

Returns A reference to the IP destination address.

hl Method

Computes or modifies the length of the IP header.

```
int hl ();
```

```
void hl (int h);
```

Parameter	Description
h	The header length (in 32-bit words) to assign to the IP header.

Returns The number of 32-bit words in the IP header, or nothing.

Description When you pass no parameter, this method computes and returns the number of 32-bit words in the IP header. If you pass the parameter, the method modifies the header length field to the specified length.

hlen Method

Computes the number of bytes in the IP header, including options.

```
int hlen ();
```

Returns The number of bytes in the IP header, including options.

id Method

Accesses the IP identifier (used for fragmentation).

```
uint16& id ();
```

Returns A reference to the IP identification field.

len Method

Accesses the IP datagram or fragment length.

```
uint16& len ();
```

Returns A reference to the length of the IP datagram or fragment in bytes.

offset Method

Accesses the fragmentation flags and fragment offset.

```
uint16& offset ();
```

Returns A reference to the word containing fragmentation flags and fragment offset.

optbase Method

Computes the location of the first IP option in the IP header.

```
unsigned char * optbase ();
```

Returns The address of the first option or the first byte of the payload.

Description This method computes and returns the location of the first IP option in the IP header, if any are present. If no options are present, it returns the address of the first byte of the payload.

payload Method

Computes the address of the first byte of payload data in the IP packet.

```
unsigned char * payload ();
```

Returns The address of the first byte of the payload.

Description This method computes and returns the location of the first byte of the payload, beyond any options that may be present.

proto Method

Accesses the IP protocol value.

```
uint8& proto ();
```

Returns A reference to the IP protocol byte.

psum Method

Computes the IP value for a pseudo-checksum.

```
uint32 psum ();
```

Returns The computed sum.

Description This method computes the 16-bit one's complement sum of the source and destination IP addresses plus the 8-bit protocol field (in the low-order byte).

This method is used internally by the ASL library, but can also be useful to some applications. For example, you can use it to compute pseudo-header checksums for UDP and TCP.

src Method

Accesses the IP source address.

```
IP4Addr& src ();
```

Returns A reference to the IP source address.

tos Method

Accesses the IP type-of-service value.

```
uint8& tos ();
```

Returns A reference to the IP type-of-service byte.

ttl Method

Accesses the IP time-to-live value.

```
uint8& ttl ();
```

Returns A reference to the IP time-to-live byte.

ver Method

Retrieves or assigns the version number of the IP header.

```
int ver ();
```

```
void ver (int v);
```

Parameter	Description
v	The version number.

Returns The version number, or nothing.

Description When you pass no parameter, this method retrieves and returns the version field of the IP header (which should be 4). If you pass the parameter, the method modifies the version field to the specified number.

vhl Method

Accesses the IP version and header lengths.

```
nuint8& vhl ();
```

Returns A reference to the byte containing the IP version and header length.

IP4Mask Class

Masks are often applied to IP addresses to determine network or subnet numbers, CIDR blocks, and so on. The `IP4Mask` class represents a 32-bit mask that can be applied to an IPv4 address.

IP masks are assumed to be left-contiguous groups of 1s followed by 0s.

The `IP4Mask` class is defined in `NBip.h`. The class contains the following methods:

Method	Description
<code>IP4Mask</code> Constructor	Instantiates the class.
<code>bits</code> Method	Retrieves the length of the mask.
<code>leftcontig</code> Method	Determines whether the 1-bits in the mask are left-contiguous.

IP4Mask Constructor

Creates an `IP4Mask` object.

```
IP4Mask (nuint32 mn);

IP4Mask (uint32 mh);

IP4Mask ();
```

Parameter	Description
<code>mh</code>	32-bit mask in host byte order
<code>mn</code>	32-bit mask in network byte order

Returns The newly created object.

Description This class has three constructors for creating masks with or without an initial value. Pass the initial value as an unsigned 32-bit word in either host or network byte order, or pass no parameter to create the object with no initial value.

bits Method

Retrieves the length of the mask.

```
int bits ();
```

Returns The number of left-contiguous 1-bits in the mask, or -1 if the 1-bits in the mask are not left-contiguous.

Example The following example creates a subnet mask with 25 bits, and sets up a message buffer containing a string that describes the form of the mask (using the common “slash notation” for subnet masks).

```
#include NBip.h
uint32 mymask = 0xffffffff80; // 255.255.255.128 or /25
IP4Mask ipm(mymask);
int nbits = ipm.bits();
if (nbits >= 0) {
    sprintf(msgbuf, "Mask is of the form /%d", nbits);
}
else {
    sprintf(msgbuf, "Mask is not left-contiguous!");
}
```

leftcontig Method

Determines whether the 1-bits in the mask are left-contiguous.

```
bool leftcontig ();
```

Returns TRUE if all the 1-bits in the mask are left-contiguous, otherwise FALSE.

Description This method returns a Boolean value indicating whether the 1-bits in the mask are left-contiguous.

IP4NAT Base Class

The `IP4Nat` class provides a base class for other IP Version 4 NAT classes. Do not create objects of type `IP4Nat` directly. Instead, use objects of type `IP4SNat`, `IP4DNat`, and `IP4SDNat`.

- Use the `IP4SNat` Class to modify only the source IP address.
- Use the `IP4DNat` Class to modify only the destination IP address.
- Use the `IP4SDNat` Class to modify both the source and destination IP addresses.

IP address translation maps an IP datagram or fragment with source and destination IP address (`s1`, `d1`) to the same datagram or fragment with a new address pair (`s2`, `d2`).

This class is defined in `NBnat.h`. It contains the following method:

Method	Description
<code>rewrite</code> Method	A pure virtual method, which is defined in the subclasses.

rewrite Method

Rewrites the addresses in the specified fragment or datagram.

```
virtual void rewrite (IP4Datagram* dp) = 0;

virtual void rewrite (IP4Fragment* fp) = 0;
```

Parameter	Description
dp	Pointer to the datagram to rewrite. When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
fp	Pointer to the single fragment to rewrite. When the method is applied to a fragment, it affects only the specified fragment.

Returns Nothing.

Description This pure virtual method is defined in the subclasses. It performs address rewriting in a specific fashion implemented by the specific subclass (that is, source, destination, or source/destination combination).

IP4SDNat Class

The IP4SDNat class is derived from the IP4Nat class. It defines the class of objects implementing source and destination rewriting for IP datagrams and fragments.

The IP4SDNat class is defined in NBnat.h. The class contains the following methods:

Method	Description
IP4SDNat Constructor	Instantiates the class.
rewrite Method	Rewrites the source and destination addresses in an IP datagram or fragment.

IP4SDNat Constructor

Creates an IP4SDNat object.

```
IP4SDNat (IP4Addr* newsrc,
          IP4Addr* newdst);
```

Parameter	Description
nesrc	Pointer to the new source address for IP NAT.
newdst	Pointer to the new destination address for IP NAT.

Returns A reference to the newly created object.

rewrite Method

Rewrites the source and destination addresses in an IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite.

Returns Nothing.

Description This method replaces the source and destination addresses in the specified datagram or fragment with the addresses specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- When the method is applied to a fragment, it affects only the specified fragment.

Example The following example illustrates the use of an `IP4DNat` object to replace a destination address. In this example, `ipa1` is an address to be placed in the IP packet's destination address field, `buf` points to the ASL buffer containing an IP packet to rewrite, and `iph` points to the IP header of the packet contained in the buffer.

```
IPDNat *ipd = new IPDNat(&ipa1); // create IP DNat object
IP4Fragment ipf(buf, iph); // create IP fragment object
ipd->rewrite(&ipf); // rewrite fragment's header
```


IP4SNat Class

The IP4SNat class is derived from the IP4Nat class. It defines the class of objects implementing source rewriting for IP datagrams and fragments.

The IP4SNat class is defined in NBnat.h. The class contains the following methods:

Method	Description
IP4SNat Constructor	Instantiates the class.
rewrite Method	Rewrites the source addresses in an IP datagram or fragment.

IP4SNat Constructor

Creates an IP4SNat object.

```
IP4SNat (IP4Addr* newsrc);
```

Parameter	Description
newsrc	Pointer to the new source address for IP NAT.

Returns A reference to the newly created object.

rewrite Method

Rewrites the source addresses in an IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite.

Returns Nothing.

Description This method replaces the source addresses in the specified datagram or fragment with the address specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- When the method is applied to a fragment, it affects only the specified fragment.

ReassemblyQueue Class

The `ReassemblyQueue` class is a container class used to reconstruct TCP streams from TCP segments that have been “snooped” on a TCP connection. This class contains a linked list of `TCPSegInfo` objects, each of which corresponds to a single TCP segment (and a corresponding complete IP datagram). The purpose of this class is not only to contain the segments but to reassemble received segments as they arrive and present them in proper sequence number order for applications to read. Applications are generally able to read data on the connection in order, one segment at a time.

The `ReassemblyQueue` class is defined in `NBTcp.h`. It contains the following methods:

Method	Description
<code>ReassemblyQueue</code> Constructor	Instantiates the class.
<code>add</code> Method	Inserts a complete IP datagram containing a TCP segment into the reassembly queue.
<code>clear</code> Method	Removes all queued segments from the reassembly queue and frees their storage.
<code>empty</code> Method	Indicates whether the reassembly queue contains segments.
<code>read</code> Method	Provides application access to the contiguous data currently queued in the reassembly queue.

ReassemblyQueue Constructor

Creates a `ReassemblyQueue` object.

```
ReassemblyQueue (TCPSeq& rcvnxt)
```

Parameter	Description
<code>rcvnxt</code>	A reference to the next TCP sequence number to expect. This sequence number is updated by the <code>add</code> method so that it always indicates the next in-order TCP sequence number expected.

Returns A reference to the newly created object.

Description This object provides for reassembly of TCP streams based on sequence numbers contained in TCP segments.

The sequence number is updated as additional segments are inserted into the object. If a segment is inserted which is not contiguous in sequence number space, it is considered “out of order” and is queued in the object until the “hole” (that is, data between it and the previous in-sequence data) is filled.

A `ReassemblyQueue` object is used internally by the TCP stream reconstruction facility.

add Method

Inserts a complete IP datagram containing a TCP segment into the reassembly queue.

```
uint32 add (IP4Datagram* dp);
```

Parameter	Description
<code>dp</code>	A pointer to a complete IP datagram containing a TCP segment. The datagram can be fragmented but must be complete.

Returns A 32-bit integer code indicating the disposition of TCP segments. See TCP Return Codes on page 298.

clear Method

Removes all queued segments from the reassembly queue and frees their storage.

```
void clear ()
```

Returns Nothing.

empty Method

Indicates whether the reassembly queue contains segments.

```
bool empty ()
```

Returns TRUE if the reassembly queue contains no segments, otherwise FALSE.

read Method

Provides application access to the contiguous data currently queued in the reassembly queue.

```
int read (TCPSegInfo*& ts);
```

Parameter	Description
ts	A pointer to a TCPSegInfo object.

Returns The number of bytes contained in the next in-sequence TCP segment referenced by the specified TCPSegInfo object.

Description This method returns an integer indicating the number of bytes of in-sequence data, and sets the ts parameter to point to a TCPSegInfo object.

You declare the `ts` argument as a pointer to a `TCPSegInfo` object, but need not allocate memory for it. The `read` method allocates the `TCPSegInfo` objects and assigns the provided pointer to a single such object for each call. Objects provided to the caller become the responsibility of the caller; no references to them are retained by the `ReassemblyQueue` object.

The `ReassemblyQueue` object can process TCP segments contained in fragmented IP datagrams. Thus, the datagram referred to by the `TCPSegInfo::segment()` method might refer to a fragmented (but complete) IP datagram. In this case, keep in mind that the TCP segment's data area is not contiguous in memory, but is split across multiple fragments inside the datagram object.

TCPDNat Class

The `TCPDNat` class is derived from the `TCPNat` class. It defines the class of objects implementing destination address and (optionally) port number and ACK number rewriting for complete and fragmented TCP segments.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a TCP header. Use the appropriate `TCPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “`IP4NAT Base Class`” on page 333.

The `TCP4DNat` class is defined in `NBnat.h`. The class contains the following methods:

Method	Description
<code>TCPDNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the destination addresses, port number (if enabled), and ACK number (if enabled) in a TCP datagram or fragment.

TCPDNat Constructor

Creates a `TCPDNat` object.

```
TCPDNat (IP4Addr* newdaddr);
```

```
TCPDNat (IP4Addr* newdaddr,
         nuint16 newdport);
```

```
TCPDNat (IP4Addr* newdaddr,
         nuint16 newdport,
         long ackoff);
```

Parameter	Description
<code>newdaddr</code>	Pointer to the new destination address to use.
<code>newdport</code>	The new destination port number to use, if port rewriting is enabled.
<code>ackoff</code>	A relative constant amount by which to modify the ACK number, if sequence/ACK number rewriting is enabled. Can be positive or negative.

Returns A reference to the newly created object.

Description Use the one-argument constructor to create TCP NAT objects that rewrite only the destination addresses in the IP header (and update the IP header checksum and TCP pseudo-header checksum appropriately).

Use the two-argument constructor to create NAT objects that also rewrite the destination port numbers in the TCP header.

Use the three-argument constructor create NAT objects that, in addition to rewriting the destination IP addresses and port numbers, also modify the ACK numbers by the specified amounts.

rewrite Method

Rewrites the destination addresses, port number (if enabled), and ACK number (if enabled) in a TCP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
<code>dp</code>	Pointer to the datagram to rewrite.
<code>fp</code>	Pointer to the fragment to rewrite. The fragment must represent a complete TCP/IP datagram.

Returns Nothing.

Description	<p data-bbox="357 131 1279 199">This method replaces the destination addresses in the specified datagram or fragment with the addresses specified when the object was constructed.</p> <p data-bbox="357 217 1279 321">If port number rewriting is enabled, the method also replaces the destination port numbers in the specified datagram or fragment with the numbers specified when the object was constructed.</p> <p data-bbox="357 338 1279 407">If ACK number rewriting is enabled, the method also modifies the ACK number by the amount specified when the object was constructed.</p> <ul data-bbox="357 425 1279 571" style="list-style-type: none"><li data-bbox="357 425 1279 494">■ When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.<li data-bbox="357 503 1279 571">■ To apply the method successfully to a fragment, the fragment must represent a complete TCP/IP datagram.
-------------	---

TCPEndpoint Class

The `TCPEndpoint` class represents a single endpoint of a TCP connection. In TCP, a connection is identified by a 4-tuple of two IP addresses and two port numbers. Each endpoint is identified by a single IP address and port number. A TCP connection (or session) has two endpoints, and the TCP object is therefore associated with two endpoint objects.

Each endpoint contains the TCP finite machine state as well as a `Reassembly-Queue` object that contains queued data. The `TCPSession` class uses the `TCPEndpoint` class internally.

The `TCPEndpoint` class is defined in `NBTcp.h`. It contains the following methods:

Method	Description
<code>TCPEndpoint</code> Constructor	Instantiates the class.
<code>TCPEndpoint</code> Destructor	Deletes all queued TCP segments and frees the object's memory.
<code>init</code> Method	Initializes a TCP endpoint.
<code>process</code> Method	Processes an incoming or outgoing TCP segment relative to the TCP endpoint object.
<code>reset</code> Method	Resets the endpoint internal state to <code>TCPS_CLOSED</code> and clears any queued data.
<code>state</code> Method	Gets the current state in the TCP finite state machine associated with the TCP endpoint.

TCPEndpoint Constructor

Creates a TCPEndpoint object.

```
TCPEndpoint ()
```

Returns A reference to the newly created object.

Description The TCPEndpoint class is created in an empty state and is unable to determine which endpoint of a connection it represents. After instantiating this object and before using it, call the initialization method (see “init Method” on page 347).

TCPEndpoint Destructor

Deletes all queued TCP segments and frees the object’s memory.

```
~TCPEndpoint ()
```

init Method

Initializes a TCP endpoint.

```
void init (IP4Addr* myaddr,  
          uint16 myport);
```

Parameter	Description
myaddr	A pointer to the IP address identifying this TCP endpoint
myport	The port number (in network byte order) identifying this TCP endpoint

Returns Nothing.

Description This method initializes a TCP endpoint object by associating it with an IP address and port number. After initialization, use the `process` method to do subsequent processing of IP datagrams and fragments containing TCP segments and ACKs.

process Method

Processes an incoming or outgoing TCP segment relative to the TCP endpoint object.

```
uint32 process (IP4Datagram* pd);
```

Parameter	Description
pd	A pointer to a complete IP datagram containing a TCP segment

Returns A 32-bit integer code indicating the disposition of TCP segments. See “TCP Return Codes” on page 298.

Description This method operates on a complete datagram. Given that the `TCPEndpoint` object is not actually the literal endpoint of the TCP connection itself, it must infer state transitions at the literal endpoints based upon observed traffic. Thus, it must monitor both directions of the TCP connection to properly follow the state at each literal endpoint.

reset Method

Resets the endpoint internal state to `TCPS_CLOSED` and clears any queued data.

```
void reset ();
```

Returns Nothing.

state Method

Gets the current state in the TCP finite state machine associated with the TCP endpoint.

```
int state ()
```

Returns A constant indicating the internal state; see “TCP Session State” on page 296.

TCPHeader Class

The `TCPHeader` class represents the standard TCP header. It contains member methods that provide direct access to the header fields, and a method for obtaining a pointer immediately beyond the TCP header (the payload).

The methods are declared as static within this class, so you can use them without instantiating the class. Refer to the `TCPHeader` class to access TCP headers received in live network packets.

The `TCPHeader` class is defined in `NBtcp.h`. It contains the following methods:

Method	Description
<code>ack</code> Method	Accesses the TCP acknowledgement number.
<code>cksum</code> Method	Accesses the TCP pseudoheader checksum.
<code>dport</code> Method	Accesses the TCP destination port number.
<code>flags</code> Method	Accesses the TCP flag bits.
<code>hlen</code> Method	Determines or controls the length of the TCP header.
<code>off</code> Method	Accesses the length of the TCP header.
<code>optbase</code> Method	Finds the first option in the TCP packet.
<code>payload</code> Method	Finds the payload data in the TCP packet.
<code>seq</code> Method	Accesses the TCP sequence number.
<code>sport</code> Method	Accesses the TCP source port number.
<code>urp</code> Method	Accesses the TCP urgent pointer field.
<code>win</code> Method	Accesses the TCP window advertisement.
<code>window</code> Method	Finds the RFC1323-style window in the TCP packet.

ack Method

Accesses the TCP acknowledgement number.

```
TCPSeq& ack ();
```

Returns A reference to the TCP acknowledgement number.

cksum Method

Accesses the TCP pseudoheader checksum.

```
nuint16& cksum ();
```

Returns A reference to the TCP pseudoheader checksum. TCP checksums are not optional.

dport Method

Accesses the TCP destination port number.

```
nuint16& dport ();
```

Returns A reference to the TCP destination port number.

flags Method

Accesses the TCP flag bits.

```
nuint8& flags ();
```

Returns A reference to the byte containing the six flag bits and two reserved bits.



NOTE: Because the returned value contains the reserved bits, you must mask these out to read the significant flag values.

hlen Method

Determines or controls the length of the TCP header.

```
int hlen ();  
  
void hlen (int bytes);
```

Parameter	Description
bytes	Specifies the number of bytes in the TCP header.

Returns When you pass no parameter, the number of option bytes in the TCP header. When you pass a parameter, nothing.

Description If you pass no parameter, this method returns the number of option bytes in the TCP header. When you pass the `bytes` parameter, the method sets the number of option bytes in the TCP header to the specified value.

off Method

Accesses the length of the TCP header.

```
nuint8 off ();
```

Returns The number of 32-bit words in the TCP header, including TCP options.

optbase Method

Finds the first option in the TCP packet.

```
unsigned char * optbase ()
```

Returns A pointer to the first byte of data beyond the urgent pointer field of the TCP header.

Description	This method finds the address of the first option in the TCP header, if any options are present. If no options are present, it returns the address of the first payload byte (which can be urgent data if the <code>URG</code> bit is set in the <code>flags</code> field).
-------------	---

payload Method

Finds the payload data in the TCP packet.

```
unsigned char * payload ();
```

Returns	A pointer to the first byte of payload data (beyond the TCP header) in the TCP packet.
---------	--

seq Method

Accesses the TCP sequence number.

```
TCPSeq& seq ();
```

Returns	A reference to the TCP sequence number.
---------	---

sport Method

Accesses the TCP source port number.

```
nuint16& sport ();
```

Returns	A reference to the TCP source port number.
---------	--

urp Method

Accesses the TCP urgent pointer field.

```
nuint16& urp ();
```

Returns A reference to the TCP urgent pointer field.

win Method

Accesses the TCP window advertisement.

```
nuint16& win ();
```

Returns A reference to the TCP window advertisement field (unscaled).

window Method

Finds the RFC1323-style window in the TCP packet.

```
uint32 window (int wshift)
```

Parameter	Description
wshift	The window shift value, or number of left-shift bit positions by which to scale the window field.

Returns The receiver’s advertised window in the segment, in bytes.

Description This method finds the window advertisement contained in the segment, taking into account the use of TCP large windows (see RFC 1323). Use this method with RFC1323-style window scaling.

TCPNat Base Class

The `TCPNat` class is a base class for other TCP NAT classes. Do not create objects of type `TCPNat` directly. Instead, use objects of type `TCPSNat`, `TCPDNat`, and `TCPSDNat`.

- Use the `TCPSNat` Class to modify only the IP source address.
- Use the `TCPDNat` Class to modify only the IP destination address.
- Use the `TCPSDNat` Class to modify both the source and destination IP addresses.

TCP NAT is similar to UDP NAT, in that you can specify in the constructor whether port numbers as well as IP layer addresses should be rewritten. For TCP, you can also specify whether sequence and acknowledgement numbers should be rewritten. Sequence number and ACK number rewriting are tied together. When you enable this feature, source-rewriting modifies the sequence number field of the TCP segment, while destination-rewriting modifies the ACK field.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a TCP header. Use the appropriate `TCPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “`IP4NAT` Base Class” on page 333.

This class is defined in `NBnat.h`. It contains the following methods:

Method	Description
<code>TCPNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the specified fragment or datagram. This pure virtual method is defined in the subclasses.
<code>ports</code> Method	Controls or determines whether port rewriting is enabled.
<code>seqs</code> Method	Controls or determines whether sequence/ACK rewriting is enabled.

TCPNat Constructor

Creates a `TCPNat` object.

```
TCPNat (bool doports,
        bool doseqs);
```

Parameter	Description
doports	When <code>TRUE</code> , port number rewriting is enabled. When <code>FALSE</code> , it is disabled.
doseqs	When <code>TRUE</code> , sequence/ACK number rewriting is enabled. When <code>FALSE</code> , it is disabled.

Returns A reference to the newly created object.

Description The constructor parameters indicate whether port number rewriting and sequence/ACK number rewriting are enabled.

rewrite Method

Rewrites the addresses in the specified fragment or datagram.

```
virtual void rewrite (IP4Datagram* dp) = 0;

virtual void rewrite (IP4Fragment* fp) = 0;
```

Parameter	Description
dp	Pointer to the datagram to rewrite. When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
fp	Pointer to the single fragment to rewrite. When the method is applied to a fragment, it affects only the specified fragment.

Returns Nothing.

Description This pure virtual method is defined in the derived classes. It performs address rewriting in a specific fashion implemented by the specific derived classes (that is, source, destination, or source/destination combination).

ports Method

Controls or determines whether port rewriting is enabled.

```
bool ports ();

void ports (bool p);
```

Parameter	Description
p	When TRUE , enables port rewriting. When FALSE , disables port rewriting. When absent, the method returns the current state.

Returns When you pass no parameter, **TRUE** if the NAT object is configured to rewrite TCP port numbers, **FALSE** if it is not. When you pass the parameter, returns nothing.

seqs Method

Controls or determines whether sequence/ACK rewriting is enabled.

```
bool seqs ();

void seqs (bool s);
```

Parameter	Description
s	When TRUE , enables sequence/ACK rewriting. When FALSE , disables sequence/ACK rewriting. When absent, the method returns the current state.

Returns When you pass no parameter, `TRUE` if the NAT object is configured to rewrite TCP sequence/ACK numbers, `FALSE` if it is not. When you pass the parameter, returns nothing.

TCPSDNat Class

The `TCPSDNat` class is derived from the `TCPNat` class. It defines the class of objects implementing source and destination address and (optionally) port number and sequence number/ACK number rewriting for complete and fragmented TCP segments.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a TCP header. Use the appropriate `TCPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “`IP4NAT Base Class`” on page 333.

This class is defined in `NBnat.h`. It contains the following methods:

Method	Description
<code>TCPSDNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the source addresses, port number (if enabled), and sequence/ACK numbers (if enabled) in a TCP datagram or fragment.

TCPSDNat Constructor

Creates a TCPSDNat object.

```
TCPSDNat (IP4Addr* newsaddr,
          IP4Addr* newdaddr);

TCPSDNat (IP4Addr* newsaddr,
          nuint16 newsport,
          IP4Addr* newdaddr,
          nuint16 newdport);

TCPSDNat (IP4Addr* newsaddr,
          nuint16 newsport,
          long seqoff,
          IP4Addr* newdaddr,
          nuint16 newdport,
          long ackoff);
```

Parameter	Description
newsaddr	Pointer to the new source address to use.
newsport	The new source port number to use, if port rewriting is enabled.
seqoff	A relative constant amount by which to modify the sequence number, if sequence number rewriting is enabled. Can be positive or negative.
newdaddr	Pointer to the new destination address to use.
newdport	The new destination port number to use, if port rewriting is enabled.
ackoff	A relative constant amount by which to modify the ACK number, if sequence/ACK number rewriting is enabled. Can be positive or negative.

Returns A reference to the newly created object.

Description Use the two-argument constructor to create TCP NAT objects that rewrite only the source and destination addresses in the IP header (and update the IP header checksum and TCP pseudo-header checksum appropriately).

Use the four-argument constructor to create NAT objects that also rewrite the source and destination port numbers in the TCP header.

Use the six-argument constructor create NAT objects that, in addition to rewriting the source and destination IP addresses and port numbers, also modify the TCP sequence and ACK numbers by the specified amounts.

rewrite Method

Rewrites the source addresses, port number (if enabled), and sequence/ACK numbers (if enabled) in a TCP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite. The fragment must represent a complete TCP/IP datagram.

Returns Nothing.

Description This method replaces the source and destination addresses in the specified datagram or fragment with the addresses specified when the object was constructed.

If port number rewriting is enabled, the method also replaces the source and destination port numbers in the specified datagram or fragment with the numbers specified when the object was constructed.

If sequence number rewriting is enabled, the method also modifies the sequence and ACK numbers by the amounts specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- To apply the method successfully to a fragment, the fragment must represent a complete TCP/IP datagram.

TCPSegInfo Class

The `TCPSegInfo` class is a container class for TCP segments that have been queued during TCP stream reconstruction and can be read by applications (using the `read` Method of the `ReassemblyQueue` Class).

When segments are queued, they are maintained in a doubly-linked list sorted by sequence number. This doubly-linked list can contain “holes.” That is, it can contain segments that are not adjacent in the space of sequence numbers because some data is missing in between.

The `TCPSegInfo` class is defined in `NBtcp.h`. It contains the following methods:

Method	Description
<code>data</code> Method	Finds the first byte of data.
<code>endseq</code> Method	Gets the sequence number of the last byte of data.
<code>flags</code> Method	Retrieves flags.
<code>len</code> Method	Gets the length of the segment.
<code>next</code> Method	Finds the next segment in the list.
<code>prev</code> Method	Finds the previous segment in the list.
<code>segment</code> Method	Finds the datagram containing the TCP segment.
<code>startseq</code> Method	Gets the starting sequence number.

data Method

Finds the first byte of data.

```
u_char *& data ();
```

Returns A reference to a pointer to the first byte of data.

endseq Method

Gets the sequence number of the last byte of data.

```
TCPSeq endseq ();
```

Returns The sequence number of the last byte of data.

flags Method

Retrieves flags.

```
uint32& flags ();
```

Returns A reference to flags associated with the segment.

len Method

Gets the length of the segment.

```
long& len ();
```

Returns A reference to the data length in the segment in bytes.

next Method

Finds the next segment in the list.

```
TCPSeqInfo*& next ();
```

Returns A reference to a pointer to the next `TCPSeqInfo` object of the forward linked list, or `NULL` if this is the last segment.

prev Method

Finds the previous segment in the list.

```
TCPSegInfo*& prev ();
```

Returns A reference to a pointer to the previous TCPSegInfo object of the forward linked list, or NULL if this is the first segment.

segment Method

Finds the datagram containing the TCP segment.

```
IP4Datagram*& segment ();
```

Returns A reference to a pointer to the datagram containing the TCP segment.

startseq Method

Gets the starting sequence number.

```
TCPSeq& startseq ();
```

Returns A reference to the starting sequence number for the segment.

TCPSeq Class

The `TCPSeq` class represents a TCP sequence number data type, and associated operators. TCP sequence numbers are 32-bit unsigned numbers that can wrap beyond $2^{32}-1$. Use these objects to handle sequence numbers like ordinary scalar types (such as unsigned integers).

TCP uses sequence numbers to keep track of an active data transfer. Each unit of data transfer is called a *segment*, and each segment contains a range of *sequence numbers*. Sequence numbers are in byte units.

If a TCP connection is open and data transfer is progressing from computer A to B, TCP segments will be flowing from A to B and *acknowledgements* will be flowing from B toward A. The acknowledgements indicate to the sender the amount of data the receiver has received.

TCP is a bi-directional protocol; that is, data can be flowing simultaneously from A to B and from B to A. In such cases, each segment (in both directions) contains data for one direction of the connection and acknowledgements for the other direction of the connection. Both sequence numbers (sending direction) and acknowledgement numbers (reverse direction) use TCP sequence numbers as the data type in the TCP header.

The `TCPSeq` class is defined in `NBtcp.h`. It contains the following methods and operators:

Method	Description
<code>TCPSeq</code> Constructor	Instantiates the class.
<code>image</code> Method	Retrieves the sequence number in network byte order.
<code>val</code> Method	Retrieves the sequence number in host byte order.
<code>TCPSeq</code> Operators	Manipulate and compare sequence numbers.

TCPSeq Constructor

Creates a TCPSeq object.

```
IP4Addr (nuint32 jn);  
  
IP4Addr (uint32 jh);  
  
IP4Addr ();
```

Parameter	Description
jn	The sequence number as an unsigned 32-bit word in network byte order
jh	The sequence number as an unsigned 32-bit word in host byte order

Returns	The newly created object.
Description	This class has three constructors for creating sequence numbers with or without an initial value. Pass the initial value as an unsigned 32-bit word in either host or network byte order, or pass no parameter to create the object with no initial value.

image Method

Retrieves the sequence number in network byte order.

```
nuint32 image ();
```

Returns	The sequence number in network byte order.
---------	--

val Method

Retrieves the sequence number in host byte order.

```
uint32 val ();
```

Returns The sequence number in host byte order.

TCPSeq Operators

The `TCPSeq` class defines the following arithmetic and relational operators to manipulate and compare sequence numbers:

Operator	Description
<code>TCPSeq TCPSeq + long</code>	Add a signed integer to a TCP sequence number, returning another TCP sequence number. A binary operator.
<code>&TCPSeq += long</code>	Add a signed integer to a TCP sequence number, returning another TCP sequence number. An assignment operator.
<code>long TCPSeq - TCPSeq</code>	Subtract one TCP sequence number from another, returning difference as a signed integer.
<code>TCPSeq TCPSeq - long</code>	Subtract a signed integer from a TCP sequence number, returning difference as a TCP sequence number. A binary operator.
<code>&TCPSeq -= long</code>	Subtract a signed integer from a TCP sequence number, returning difference as a TCP sequence number. An assignment operator.
<code>TCPSeq++, ++TCPSeq</code>	Increment a TCP sequence number.
<code>TCPSeq--, --TCPSeq</code>	Decrement a TCP sequence number.
<code>==, !=</code>	Compare two TCP sequence numbers for equality or inequality.
<code>>, <, >=, <=</code>	Compare two TCP sequence numbers for order.

TCPSession Class

The `TCPSession` class represents a bi-directional TCP connection. It includes two `TCPEndpoint` objects that each include a reassembly queue. When the `TCPSession` object is able to process all data sent on the connection in either direction, it has a reasonably complete picture of the progress and data exchanged across the connection.

The `TCPSession` class is defined in `NBtcp.h`. It contains the following methods:

Method	Description
<code>TCPSession</code> Constructor	Instantiates the class.
<code>TCPSession</code> Destructor	Deletes all TCP segments queued and frees the object's memory.
<code>process</code> Method	Processes a TCP segment on the connection.
<code>client</code> Method	Gets the client endpoint object embedded in the session object.
<code>server</code> Method	Gets the server endpoint object embedded in the session object.

TCPSession Constructor

Creates a `TCPSession` object.

```
TCPSession (IP4Datagram* dp);
```

Parameter	Description
<code>pd</code>	A pointer to a complete IP datagram containing the first TCP segment on the connection

Returns A reference to the newly created object.

Description You create a `TCPSession` object when a TCP segment arrives on a new connection. The session object determines, from the contents of the segment, which endpoint is considered the client (the active opener—generally the sender of the first SYN), and which is considered the server (the passive opener—generally the sender of the first SYN+ACK).

In the event of simultaneous active opens, a rare case when both endpoints send SYN packets, the session object considers the sender of the first SYN to be the client. (The terms client and server are only loosely defined here, and do not affect the proper operation of the object.)

TCPSession Destructor

Deletes all TCP segments queued and frees the object’s memory.

```
~TCPSession ()
```

process Method

Processes a TCP segment on the connection.

```
uint32 process (IP4Datagram* pd);

uint32 process (IP4Datagram* pd,
               uint32& clcode,
               uint32& srvcode);
```

Parameter	Description
pd	A pointer to a complete IP datagram containing a TCP segment.
clcode	On return, a reference to the client’s <code>process</code> method return code.
srvcode	On return, a reference to the server’s <code>process</code> method return code.

Returns A 32-bit integer code indicating the disposition of TCP segments. See TCP Return Codes on page 298. The value returned is the bitwise OR of the client’s and server’s return codes.

Description	<p>This method processes a TCP segment on the connection by passing the data-gram to each endpoint's <code>process</code> method.</p> <ul style="list-style-type: none">■ When you call the method with only one parameter, it returns the bitwise OR of the client's and server's <code>process</code> methods.■ When you call the method with all three parameters, it also fills in the indi-vidual client and server return codes.
-------------	---

client Method

Gets the client endpoint object embedded in the session object.

```
TCPEndpoint& client ();
```

Returns	A reference to the client <code>TCPEndpoint</code> object.
---------	--

server Method

Gets the server endpoint object embedded in the session object.

```
TCPEndpoint& server ();
```

Returns	A reference to the server <code>TCPEndpoint</code> object.
---------	--

TCPSNat Class

The `TCPSNat` class is derived from the `TCPNat` class. It defines the class of objects implementing source address and (optionally) port number and sequence number rewriting for complete and fragmented TCP segments.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a TCP header. Use the appropriate `TCPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “IP4NAT Base Class” on page 333.

The `TCP4SNat` class is defined in `NBnat.h`. The class contains the following methods:

Method	Description
<code>TCPSNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the source addresses, port number (if enabled), and sequence number (if enabled) in a TCP datagram or fragment.

TCPSNat Constructor

Creates a `TCPSNat` object.

```
TCPSNat (IP4Addr* newsaddr);
```

```
TCPSNat (IP4Addr* newsaddr,  
         nuint16 newsport);
```

```
TCPSNat (IP4Addr* newsaddr,  
         nuint16 newsport,  
         long seqoff)
```

Parameter	Description
newsaddr	Pointer to the new source address to use.
newsport	The new source port number to use, if port rewriting is enabled.
seqoff	A relative constant amount by which to modify the sequence/ACK number, if sequence number rewriting is enabled. Can be positive or negative.

Returns A reference to the newly created object.

Description Use the single-argument constructor to create TCP NAT objects that rewrite only the addresses in the IP header (and update the IP header checksum and TCP pseudo-header checksum appropriately).

Use the two-argument constructor to create NAT objects that also rewrite the source port number in the TCP header.

Use the three-argument constructor to create NAT objects that, in addition to rewriting the source IP address and source port number, also modify the TCP sequence number by the specified amount.

rewrite Method

Rewrites the source addresses, port number (if enabled), and sequence number (if enabled) in a TCP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite. The fragment must represent a complete TCP/IP datagram.

Returns Nothing.

Description This method replaces the source addresses in the specified datagram or fragment with the address specified when the object was constructed.

If port number rewriting is enabled, the method also replaces the source port number in the specified datagram or fragment with the number specified when the object was constructed.

If sequence number rewriting is enabled, the method also modifies the sequence number by the amount specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- To apply the method successfully to a fragment, the fragment must represent a complete TCP/IP datagram.

UDPDNat Class

The `UDPDNat` class is derived from the `UDPNat` class. It defines the class of objects implementing destination address and (optionally) port number rewriting for complete and fragmented UDP datagrams.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a UDP header. Use the appropriate `UDPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “IP4NAT Base Class” on page 333.

The `UDPDNat` class is defined in `NBNat.h`. The class contains the following methods:

Method	Description
<code>UDPDNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the destination addresses and port number (if enabled) in a UDP/IP datagram or fragment.

UDPDNat Constructor

Creates a `UDPDNat` object.

```
UDPDNat(IP4Addr* newdaddr);
```

```
UDPDNat (IP4Addr* newdaddr,
         nuint16 newdport);
```

Parameter	Description
<code>newdaddr</code>	Pointer to the new destination address to use.
<code>newdport</code>	The new destination port number to use, if port rewriting is enabled.

Returns A reference to the newly created object.

Description Use the single-argument constructor to create UDP NAT objects that rewrite only the destination addresses in the IP header (and update the IP header checksum and UDP pseudo-header checksum appropriately).

Use the two-argument constructor to create NAT objects that also rewrite the destination port number in the UDP header. For fragmented UDP datagrams, the port numbers are generally present only in the first fragment.

rewrite Method

Rewrites the destination addresses and port number (if enabled) in a UDP/IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite. The fragment must represent a complete UDP/IP datagram.

Returns Nothing.

Description This method replaces the destination addresses in the specified datagram or fragment with the address specified when the object was constructed. If port number rewriting is enabled, the method also replaces the destination port number in the specified datagram or fragment with the number specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- To apply the method successfully to a fragment, the fragment must represent a complete UDP/IP datagram.

UDPHeader Class

The `UDPHeader` class represents the standard UDP header. The UDP protocol is relatively simple, and is represented in ASL by a single class. For address and port number translation of UDP, see `UDPNat` Base Class on page 378.

The methods are declared as static within this class, so you can use them without instantiating the class. Refer to the `UDPHeader` class to access UDP headers received in live network packets.

The class is defined in `NBudp.h`, and contains the following methods:

Method	Description
<code>cksum</code> Method	Accesses the UDP pseudoheader checksum.
<code>dport</code> Method	Accesses the destination UDP port number.
<code>len</code> Method	Accesses the length of the UDP header.
<code>payload</code> Method	Finds the payload in the UDP packet.
<code>sport</code> Method	Accesses the source UDP port number.

cksum Method

Accesses the UDP pseudoheader checksum.

```
uint16& cksum ();
```

Returns A reference to the UDP pseudoheader checksum. UDP checksums are optional; a value of all zero bits indicates no checksum was computed.

dport Method

Accesses the destination UDP port number.

```
nuint16& dport ();
```

Returns A reference to the destination UDP port number.

len Method

Accesses the UDP header length.

```
nuint16& len ();
```

Returns A reference to the UDP length field.

payload Method

Finds the payload in the UDP packet.

```
unsigned char * payload ();
```

Returns The address of the first byte of payload data.

Description Finds and returns the address of the first byte of payload data beyond the UDP header in the UDP packet.

sport Method

Accesses the source UDP port number.

```
nuint16& sport ();
```

Returns A reference to the source UDP port number.

UDPNat Base Class

The `UDPNat` class is a base class for other UDP NAT subclasses. Do not create objects of type `UDPNat` directly. Instead, use objects of type `UDPSNat`, `UDPDNat`, and `UDPSDNat`.

- Use the `UDPSNat` Class to modify only the source IP address.
- Use the `UDPDNat` Class to modify only the destination IP address.
- Use the `UDPSDNat` Class to modify both the source and destination IP addresses.

UDP NAT is similar to IP NAT, except that for UDP NAT you can specify in the constructor that port numbers as well as IP layer addresses should be rewritten.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a UDP header. Use the appropriate `UDPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “`IP4NAT` Base Class” on page 333.

This class is defined in `NBnat.h`. It contains the following methods:

Method	Description
<code>UDPNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the specified fragment or datagram. This pure virtual method is defined in the subclasses.
<code>ports</code> Method	Controls or determines whether port rewriting is enabled.

UDPNat Constructor

Creates a `UDPNat` object.

```
UDPNat (bool doports);
```

Parameter	Description
doports	When <code>TRUE</code> , port number rewriting is enabled. When <code>FALSE</code> , it is disabled.

Returns A reference to the newly created object.

Description The constructor takes a parameter that indicates whether port number rewriting is enabled.

rewrite Method

Rewrites the addresses in the specified fragment or datagram.

```
virtual void rewrite (IP4Datagram* dp) = 0;
```

```
virtual void rewrite (IP4Fragment* fp) = 0;
```

Parameter	Description
dp	Pointer to the datagram to rewrite. When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
fp	Pointer to the single fragment to rewrite. When the method is applied to a fragment, it affects only the specified fragment.

Returns Nothing.

Description This pure virtual method is defined in the derived classes. It performs address rewriting in a specific fashion implemented by the specific derived classes (that is, source, destination, or source/destination combination).

ports Method

Controls or determines whether port rewriting is enabled.

```
bool ports ();  
  
void ports (bool p);
```

Parameter	Description
p	When TRUE, enables port rewriting. When FALSE, disables port rewriting. When absent, the method returns the current state.

Returns When you pass no parameter, TRUE if the NAT object is configured to rewrite UDP port numbers, FALSE if it is not. When you pass the parameter, returns nothing.

UDPSDNat Class

The `UDPSDNat` class is derived from the `UDPNat` class. It defines the class of objects implementing source and destination address and (optionally) port number rewriting for complete and fragmented UDP datagrams.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a UDP header. Use the appropriate `UDPNat` class for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “IP4NAT Base Class” on page 333.

The `UDPSDNat` class is defined in `NBnat.h`. The class contains the following methods:

Method	Description
<code>UDPSDNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites both the source and destination addresses and port numbers (if enabled) in a UDP/IP datagram or fragment.

UDPSDNat Constructor

Creates a `UDPSDNat` object.

```
UDPSDNat (IP4Addr* newsaddr,
          IP4Addr* newdaddr);
```

```
UDPSNnat (IP4Addr* newsaddr,
          nuint16 newsport,
          IP4Addr* newdaddr,
          nuint16 newdport);
```

Parameter	Description
<code>newsaddr</code>	Pointer to the new source address to use.
<code>newsport</code>	The new source port number to use, if port rewriting is enabled.

Parameter	Description
<code>newdaddr</code>	Pointer to the new destination address to use.
<code>newdport</code>	The new destination port number to use, if port rewriting is enabled.

Returns A reference to the newly created object.

Description Use the two-argument constructor to create UDP NAT objects that rewrite only the source and destination addresses in the IP header (and update the IP header checksum and UDP pseudo-header checksum appropriately).

Use the four-argument constructor to create NAT objects that also rewrite the source and destination port numbers in the UDP header. For fragmented UDP datagrams, the port numbers are generally present only in the first fragment.

rewrite Method

Rewrites both the source and destination addresses and port numbers (if enabled) in a UDP/IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
<code>dp</code>	Pointer to the datagram to rewrite.
<code>fp</code>	Pointer to the fragment to rewrite. The fragment must represent a complete UDP/IP datagram.

Returns Nothing.

Description This method replaces the source and destination addresses in the specified datagram or fragment with the addresses specified when the object was constructed. If port number rewriting is enabled, the method also replaces the source and destination port numbers in the specified datagram or fragment with the numbers specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.

- To apply the method successfully to a fragment, the fragment must represent a complete UDP/IP datagram.

UDPSNat Class

The `UDPSNat` class is derived from the `UDPNat` class. It defines the class of objects implementing source address and (optionally) port number rewriting for complete and fragmented UDP datagrams.



NOTE: When performing NAT on a fragmented datagram, only the first fragment has a UDP header. Use the appropriate `UDPNat` subclass for the first fragment, then use the corresponding `IP4Nat` subclass for NAT on subsequent fragments. See also “IP4NAT Base Class” on page 333.

The `UDPSNat` class is defined in `NBnat.h`. The class contains the following methods:

Method	Description
<code>UDPSNat</code> Constructor	Instantiates the class.
<code>rewrite</code> Method	Rewrites the source addresses and port number (if enabled) in a UDP/IP datagram or fragment.

UDPSNat Constructor

Creates a `UDPSNat` object.

```
UDPSNat (IP4Addr* newsaddr);
```

```
UDPSNat (IP4Addr* newsaddr,
         nuint16 newsport);
```

Parameter	Description
<code>newsaddr</code>	Pointer to the new source address to use.
<code>newsport</code>	The new source port number to use, if port rewriting is enabled.

Returns A reference to the newly created object.

Description Use the single-argument constructor to create UDP NAT objects that rewrite only the addresses in the IP header (and update the IP header checksum and UDP pseudo-header checksum appropriately).

Use the two-argument constructor to create NAT objects that also rewrite the source port number in the UDP header. For fragmented UDP datagrams, the port numbers are generally present only in the first fragment.

rewrite Method

Rewrites the source addresses and port number (if enabled) in a UDP/IP datagram or fragment.

```
void rewrite (IP4Datagram* dp);
```

```
void rewrite (IP4Fragment* fp);
```

Parameter	Description
dp	Pointer to the datagram to rewrite.
fp	Pointer to the fragment to rewrite. The fragment must represent a complete UDP/IP datagram.

Returns Nothing.

Description This method replaces the source addresses in the specified datagram or fragment with the address specified when the object was constructed. If port number rewriting is enabled, the method also replaces the source port number in the specified datagram or fragment with the number specified when the object was constructed.

- When the method is applied to a datagram, each of the fragment headers composing the datagram is rewritten.
- To apply the method successfully to a fragment, the fragment must represent a complete UDP/IP datagram.

Chapter 6

Network Classification Language



This chapter describes the syntax of Network Classification Language (NCL), which you use to write the classification part of an ACE. It describes how to create classification rules and sets, and how to use the NCL compiler to generate header files that synchronize your action code with your NCL code.

The chapter contains the following sections:

- Overview
- NCL Rules File Structure and Elements
- Protocol Definitions
- Predicate Definitions
- Sets and Named Searches
- Rules and Actions
- Synchronizing NCL with Action Code

Overview

The accelerator module of an IX-API SDK application contains *classification rules* that you specify in NCL, which call the *action functions* that you specify in C and C++, using the Action Services Library (ASL).

You do not compile the NCL rules file for an ACE. Classification rules are compiled on the fly by a fast incremental compiler provided with the IX-API SDK. A dynamic linker/loader included with the IX-API SDK links the classification rules with the action implementations and loads the resulting combination into the Policy Accelerator.

NCL rules files have the extension `.ncl`. You specify the source code NCL file in the makefile (`myrulefile.ncl`). You pass its name (`myrulefile`) to the `load` method of the host's `AceManager` object in the initialization function in the

action code file. The ACE manager loads the file into the ACE for the accelerator module. For more information, see “Initialization Function” on page 172 in Chapter 4, “Action Services Library.”

The classification rules that you define for an ACE can only call those action functions that you have defined in the action code file that is part of the same ACE. For more information on defining action functions, see “Action Functions” on page 115 in Chapter 4, “Action Services Library.”

In addition to defining the classification rules, the NCL rules file defines sets and searches for the ACE. These sets must also be represented by ASL objects in the action code. You use the NCL compiler as a standalone command line tool to generate a header file that defines ASL subclasses to represent these sets and searches on the action side of the ACE. You can modify this header file, then include it in your action code file before you compile the action code. For more information, see “Synchronizing NCL with Action Code” on page 411.

See Also

Chapter 8, “Communication Within an Application,” in *Developing Applications Using the IX-API SDK*

NCL Rules File Structure and Elements

An NCL rules file, like a C or C++ file, begins with file inclusion statements and constant definitions. It then has *protocol definitions*, followed by *rules*. Between or within these two main sections, you can define predicates, sets, and searches.

Because the compiler does not allow forward references, you must define a predicate or search before you can use it as part of a protocol definition, in another predicate or search, or in a rule.

NCL rules files can contain the following preprocessor elements:

- Include Files (page 389)
- Symbolic Constants (page 389)
- Comments (page 390)

NCL rules files also contain the following language-specific elements:

- Names for various structures (page 392), which you must construct according to specific naming conventions

- Keywords (page 392), which identify statement types and parts of statements
- Operators (page 392), which you use to construct Boolean or constant expressions

Include Files

Use the `#include` keyword to include other NCL files within the compilation unit so that you can reuse existing code. The syntax is as follows:

```
#include <filename>
#include "filename"
```

Argument	Description
<i>filename</i>	Name of the file to be included, enclosed in double-quotes. This can be any legal file name supported by the host.

The `#include` directive must start on a new line but can include spaces immediately preceding the pound sign (`#`). No spaces are allowed between `#` and `include`.

- When the *filename* argument is enclosed in angle brackets, the preprocessor searches for the file in the directory `%NBPATH%\include\NBncl`.
- When the *filename* argument is enclosed in double quotes, the preprocessor searches for the file in the directory containing the NCL rules file. If the file is not found, it searches for the file in the directory `%NBPATH%\include\NBncl`.

Examples

```
#include "myproto.def" // protocol definitions
#include <stdrules.rul> // standard rules
```

Symbolic Constants

Use the `#define` keyword to define symbolic constants in NCL. You use symbolic constants to represent arbitrary numbers.

The syntax for the `#define` directive is:

```
#define symbolic-name constant-value
```

Argument	Description
<i>symbolic-name</i>	Name of the constant. By convention, all uppercase (to distinguish from variables).
<i>constant-value</i>	Value of the constant. Cannot include spaces unless they are delimited by parentheses or quotation marks.

The `#define` directive must start on a new line but can include spaces immediately preceding the pound character (`#`). No spaces are allowed between `#` and `define`. Use the line continuation character (`\`) to continue beyond one physical line.

Value Formats

NCL supports the following standard and domain-specific formats for constant values:

- Conventional decimal and hexadecimal formats
Standard hexadecimal constants are defined as in C, with a leading `0x` prefix. Numbers with no prefix are decimal.
For example: `#define TELNET_PORT 23`
For data smaller than four bytes in length, unsigned extension to four bytes is performed automatically.
- Dotted-quad form for IP version 4 addresses
For example: `#define IP_ADDR 10.2.6.13`
- Colon-separated hexadecimal for Ethernet and IP version 6 addresses
For example: `#define MAC_ADDR c6:1e:f0:34:7a:93`

Comments

NCL supports C style for comments that can have multiple lines, and C++ style for single-line comments. Comments can occur anywhere in the program.

- C style comments use `/*` to indicate the start of a comment, and `*/` to indicate the end. You cannot nest comments in this form.
- C++ style comments use `//` to indicate the start of the comment. These comments stop at the end of the line. Because all characters are discarded from the `//` to the end of the line, you can nest comments in this form.

The following are examples of legal comments:

```
/* C style comment on single line */
// C++ style comment; compiler ignores to end-of-line
```

```

/* C style comments across multiple lines
   second line
   third line */
// C++ style // still ignored to end-of-line
/* C style // C++ style embedded, ignored */

```

The following are examples of *illegal* comment syntax:

```

/ * space between comment delimiters in C style */
/ / space between slashes in C++ style
/* C-style /* nesting */ This part not in comment */
// /* Mixed styles - next line not in comment
*/

```

Names

An NCL rules file can contain the following types of named entities:

- Constants
- Protocols
- Protocol fields
- Predicates
- Sets
- Searches
- Rules

Names are case sensitive, must begin with an alphabetic character, and can include alphanumeric characters and underscores.

For example, legal names could include the following:

```

set_tcp_udp
IsIP
isIPv6
set_udp_ports

```

You cannot use NCL keywords as names (see Keywords following). Names cannot contain operators, any special characters other than underscores, or begin with numbers. The following are examples of illegal names:

```

6_byte_ip           //ILLEGAL name: begins with number
set_tcp+udp         //ILLEGAL name: contains operator

```

Keywords

Keywords are a special set of words that have preassigned meanings in NCL. Keywords identify statements and parts of statements. You cannot use NCL keywords as names. The following table lists all NCL keywords with references to the sections in which they are described.

Keyword	Description
#define	Creates readable symbolic constants. See “Symbolic Constants” on page 389.
#include	Includes files in the compilation unit so that you can reuse existing code. See “Include Files” on page 389.
at	Activates the starting offset of the protocol. Used in the <code>demux</code> statement in a protocol definition. See “Identifying Nested Protocols” on page 399.
default	Specifies a default protocol. Used in the <code>demux</code> statement in a protocol definition. See “Identifying Nested Protocols” on page 399.
demux	Indicates how demultiplexing should be performed to identify nested protocols. See “Identifying Nested Protocols” on page 399.
intrinsic	Declares intrinsics (compiled functions) in the TCP/IP protocol definitions. See “Using the Built-in TCP/IP Protocol Definition” on page 396.
protocol	Identifies a protocol definition and its name. See “Protocol Definitions” on page 395.
requires	Includes an optional Boolean expression in a named search. See “Defining Named Searches” on page 404.
rule	Defines a rule. See “Rules and Actions” on page 406.
search	Defines a named search. See “Defining Named Searches” on page 404.
set	Defines a set. See “Defining a Set” on page 403.
size_hint	Defines the expected number of members of a set. See “Defining a Set” on page 403.
with	Introduces a conditional clause in rule execution. See “Conditional Rule Execution” on page 410.

Operators

NCL supports arithmetic, logical, relational, and bit-wise binary operators for use in constructing Boolean and constant expressions. You can use expressions in protocol field definitions, predicates, and searches.

Arithmetic Operators

Arithmetic operators result in scalar quantities, which you typically use for comparisons. Addition, subtraction, and logical shifts are not supported for fields larger than four bytes.

The shift operators do logical shifts. Arithmetic shifts are not available. The shift amount is a compile-time constant.

✓ **NOTE:** NCL does not support multiplication, division, or modulo operators.

NCL supports the following arithmetic and grouping operators in field and predicate definitions:

Operator	Description
()	Grouping operator
+	Addition
-	Subtraction
<<	Logical left shift
>>	Logical right shift

Logical and Relational Operators

Logical and relational operators result in Boolean values. NCL supports the following logical and relational operators:

Operator	Description
&&	Logical AND
	Logical OR
!	Logical NOT
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Operator	Description
==	Equal to
!=	Not equal to

Bit-wise Operators

NCL supports the following bit-wise operators for masking and setting bits:

Operator	Description
&	Bit-wise AND
	Bit-wise OR
^	Bit-wise exclusive OR
~	Bit-wise one's complement

Precedence

The following table shows the precedence and associativity of all NCL operators, in decreasing order of precedence:

Operator	Associativity
() []	Left to right
! ~	Right to left
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right

Protocol Definitions

A protocol definition names and describes a protocol. It names and describes the header fields that make up the protocol, and describes the relationship among multiple protocols. You generally define protocols in the first main section of an NCL rules file, immediately after the file inclusions and constant definitions, and before the rules section.

The keyword `protocol` identifies a protocol definition and its name. The syntax is as follows:

```
protocol protocol_name {
    field_name { field_description }
    intrinsic fn_name {}
    predicate predicate_name { predicate_description }
    demux {
        boolean_exp { protocol_name at offset }
        default { protocol_name at offset }
    }
}
```

The `protocol` statement includes any number of declarations for named fields (see “Defining Protocol Fields” on page 398) and a demultiplexing construct that identifies nested protocols (see “Identifying Nested Protocols” on page 399.) It can contain any number of predicate definitions, and can use previously defined predicates (see “Predicate Definitions” on page 402.)

The built-in TCP/IP protocol definitions also contain *intrinsic*s, or built-in functions (see “Intrinsic Functions” on page 397).

You access the fields and predicates in a protocol by specifying the protocol name and the field or predicate name separated by the dot operator. For example:

```
ip.length
ip.bcast
ip.chksumvalid
```

Example Protocol Definition

The following example is the definition for the IP protocol. For more information on this protocol, see “Using the Built-in TCP/IP Protocol Definition” on page 396.

```
protocol ip {
    vers { (ip[0:1] & 0xf0) >> 4 }
    hlength { ip[0:1] & 0x0f }
    hlength_b { hlength << 2 }
    tos { ip[1:1] }
```

```

length { ip[2:2] }
id { ip[4:2] }
flags { (ip[6:1] & 0xe0) >> 5 }
fragoffset { ip[6:2] & 0x1fff }
ttl { ip[8:1] }
proto { ip[9:1] }
chksum { ip[10:2] }
src { ip[12:4] }
dst { ip[16:4] }
intrinsic chksumvalid {}
predicate bcast { dst == 255.255.255.255 }
predicate mcast { (dst & 0xf0000000) == 0xe0000000 }
predicate frag { fragoffset != 0 || (flags & 2) != 0 }
demux {
    ( proto == 6 ) { tcp at hlength_b }
    ( proto == 17 ) { udp at hlength_b }
    ( proto == 1 ) { icmp at hlength_b }
    ( proto == 2 ) { igmp at hlength_b }
    default { unknownIP at hlength_b }
}
}

```

- The name `ip` identifies the protocol being defined.
- The protocol definition includes fields that correspond to portions of the IP header comprising one or more bytes. For more information on field definitions, see “Defining Protocol Fields” on page 398.
 - The fields `vers`, `hlength`, `flags`, and `fragoffset` have special operations that extract certain bits from the IP header.
 - The field `hlength_b` holds the length of the header in bytes computed using the `hlength` field (which is in units of 32-bit words).
- The predicates `bcast`, `mcast`, and `frag` can be useful in defining other rules or predicates. For more information on predicate definitions, see “Predicate Definitions” on page 402.
- The `demux` statement indicates that this protocol can contain any of four other protocols. For more information on nested protocols, see “Identifying Nested Protocols” on page 399.

Using the Built-in TCP/IP Protocol Definition

The IX-API SDK distribution includes NCL rules files that define the TCP/IP protocol. You can include these files in your application, and also use them as templates for defining other protocols. The sample files are located in the following directory:

SDKinstallpath/include/NBncl

For more information on manipulating TCP/IP packets, see Chapter 5, “ASL Extensions for TCP/IP,”

Intrinsic Functions

The TCP/IP protocol definitions make use of internally implemented, compiled functions called *intrinsic*s. Intrinsic functions provide convenient or highly optimized functions that are not easily expressed using the standard language constructs.

You can refer to the intrinsic functions in expressions. For example, the definition of the IP protocol uses the intrinsic function `checksumvalid`. If you have included the IP protocol definition, you can use the expression `ip.chksum-valid` when creating expressions for predicates, searches, or rules.

You cannot define new intrinsic functions, change their implementations, or use the `intrinsic` keyword in your own protocol definitions.

In the protocol definitions for TCP/IP, intrinsic functions are specified by the keyword `intrinsic` followed by the intrinsic name. The intrinsic functions generate 32-bit checksums, and validate checksums by returning a Boolean value. The following table lists the intrinsic functions provided in NCL:

Intrinsic Name	Functionality
<code>ip.chksumvalid</code>	Checks the validity of the <code>ip</code> header checksum. Returns <code>TRUE</code> if it is valid.
<code>ip.genchksum</code>	Generates the <code>ip</code> header checksum and returns it as a 32-bit value. The checksum is always 0 for a valid packet. If the current packet is a fragment the checksum has no meaning.
<code>tcp.chksumvalid</code>	Checks the validity of the <code>tcp</code> pseudo checksum. Returns <code>TRUE</code> if the current packet is a fragment.
<code>tcp.genchksum</code>	Generates the <code>tcp</code> pseudo checksum and returns it as a 32-bit value. If the current packet is a fragment and is not the first packet, returns the partial checksum over the <code>ip</code> payload.
<code>udp.chksumvalid</code>	Checks the validity of the <code>udp</code> pseudo checksum. Returns <code>TRUE</code> if the current packet is a fragment.
<code>udp.genchksum</code>	Generates the <code>udp</code> pseudo checksum and returns it as a 32-bit value. If the current packet is a fragment, returns the partial checksum.

Defining Protocol Fields

Protocol fields are elements of protocol definitions that specify the location and size of portions of a packet header.

You declare and define fields within a protocol definition by specifying a field name, followed by the offset relative to a protocol (usually the containing protocol), and the field length in bytes.

Specify fields using the following syntax:

```
field_name { protocol_name [offset:size] }
```

Argument	Description
<i>field_name</i>	The name of a range field defined to reside at <i>offset</i> bytes from the beginning of the protocol header belonging to <i>protocol_name</i> .
<i>protocol_name</i>	The name of the protocol header.
<i>offset</i>	The number of bytes offset from the beginning of the specified protocol header (<i>protocol_name</i>). Can be a constant expression.
<i>size</i>	Size of the field in bytes. Can be a constant expression.

The field definitions act as access methods to the areas within the protocol header or payload. In a predicate or search, you retrieve a field value from the current packet using the following syntax:

```
protocol.field_name
```

Because the offset is relative, you can specify a particular header field without knowing the absolute offset of any particular protocol header. In the following example, the location of the four-byte field `dst` is specified at byte offset 16 from the beginning of the IP protocol header.

```
dst { ip[16:4] }
```

You can specify offsets and sizes using *constant expressions*. A constant expression is any legal expression using constants and operators that evaluates to a constant. For example:

```
val { ip[22:(myconst + (4<<5) + (2 | 3) - (2|3))] }
```

Fields use byte-oriented units, and NCL stores all values in network byte order. (See “Byte Order Issues” on page 10 in Chapter 2, “System Types and Methods.”) You can define fields using a combination of byte ranges and shift,

mask or grouping operations. Use mask and shift operations to access non-byte-sized header fields. In the following example, the field `ver` is a half-byte-sized field at the beginning of the IP header.

```
ver { (ip[0:1] & 0xf0) >> 4 }
```

Examples

The following example declares the field `dest_addr` as a four-byte field located at offset six bytes from the start of the protocol `MyProto`:

```
dest_addr { MyProto[6:4] }
```

The following example declares the field `bit_flags` as a bit field. Because it crosses a byte boundary, two bytes are used with a mask and right-shift operation to get the field value.

```
bit_flags { (MyProto[10:2] & 0xff0) >> 8 }
```

Identifying Nested Protocols

Protocols can contain other protocols. You use the keyword `demux` in a protocol definition to identify other protocols that can be nested within it. (The keyword comes from *demultiplexing*, the process of extracting nested protocols from the protocols that contain them.) The `demux` statement, if used, must be the last statement in the protocol definition.

The syntax for the `demux` statement is as follows:

```
demux {  
    boolean_exp { protocol_name at offset }  
    ...  
    default { protocol_name at offset }  
}
```

Argument	Description
<i>boolean_exp</i>	A Boolean expression that must succeed for the associated protocol to become active.
<i>protocol_name at offset</i>	The name of the nested protocol and its starting offset from the beginning of the enclosing protocol.
default { <i>protocol_name at offset</i> }	A protocol to be activated if all other expressions fail. This clause is optional.

The `demux` statement can include any number of expressions. Each is associated with a protocol that can be contained in the protocol being defined. When the Boolean expression succeeds, the associated nested protocol is selected.

When a packet arrives, the Policy Accelerator parses the protocol definitions to classify the packet. When it reaches the `demux` statement, the parser evaluates the expressions in the order in which they appear. The first expression that evaluates to `TRUE` identifies the nested protocol, and the parser continues into the indicated protocol definition.

After a field value in the current packet causes an expression to succeed and its nested protocol to be selected, the parser does not continue into the remaining expressions in the `demux` statement. For this reason, the protocol most likely to occur should be the first one in the sequence. If you include a default protocol it must be the last expression in the `demux` statement, indicated by the keyword `default`. The `default` expression always succeeds.

While a protocol is being parsed, the protocol's name becomes a Boolean expression that evaluates to `TRUE`. For example, if the IP protocol is currently being parsed, the expression `ip` evaluates to `TRUE`. (An exception to this occurs when you pass a protocol's name to an action from a rule. In this case, the name evaluates to a pointer to that protocol's starting position in the packet. For more information, see "Passing Action Arguments" on page 408.)

Example

The following is an example of a `demux` declaration.

```
demux {
  (length == 10) { proto_a at offset_a }
  (flags && predicate_x) { proto_b at offset_b }
  default { proto_default at offset_default } }
```

- Protocol `proto_a` occurs at offset `offset_a` if the expression `length` equals ten.
- Protocol `proto_b` occurs at offset `offset_b` if `flags` is `TRUE`, `predicate_x` (a pre-defined Boolean expression) is `TRUE`, and `length` is not equal to 10.
- The default protocol, `proto_default`, is defined here so that packets not matching the predefined criteria can be processed.

The following is the `demux` statement from the IP protocol definition:

```
demux {
  ( proto == 6 ) { tcp at hlength_b }
  ( proto == 17 ) { udp at hlength_b }
  ( proto == 1 ) { icmp at hlength_b }
  ( proto == 2 ) { igmp at hlength_b }
```



```
default { unknownIP at hlength_b }
```

- The statement indicates that this protocol can contain four types of nested protocols (apart from the default) under different conditions.
- The demultiplexing key is the protocol type specified by the value of the `proto` field of the containing protocol.
- Each type of nested protocol begins at offset `hlength_b` relative to the start of the IP header. This is a field with calculated value, defined earlier in the protocol.

Extending Protocol Definitions

You can extend previously-defined protocols by providing additional declarations for new fields outside protocol definitions. You can also define new predicates to be associated with a previously defined protocol. (See “Predicate Definitions” on page 402.)

You can extend only those protocols that are previously defined in the same file or in an included file. You can use this feature to create alternate definitions of a protocol by putting the base definition in an include file, then extending it differently in different files.

Adding Fields

Use the following syntax to add a field to a previously defined protocol:

```
protocol_name.field_name { definition }
```

Argument	Description
<i>protocol_name</i>	The name of a previously defined protocol.
<i>field_name</i>	The name of the new field.
<i>definition</i>	The offset and size definition of the new field.

The following example declares a new field called `newfield` for the protocol `xx`:

```
xx.newfield { xx[10:4] }
```

Adding Predicates

Use the following syntax to add a predicate to a previously defined protocol:

```
predicate protocol_name.pred_name { definition }
```

Argument	Description
<i>protocol_name</i>	The name of a previously defined protocol.
<i>pred_name</i>	The name of the new predicate.
<i>definition</i>	The Boolean expression that defines the predicate.

The following example declares a new predicate called `newpred` for the protocol `xx`:

```
predicate xx.newpred { xx[8:2] != 10 }
```

Predicate Definitions

Predicates are named Boolean expressions that use protocol field accessors, other Boolean expressions, and previously defined predicates as operands. You name predicates so that you can reuse them in rules, other predicates, and so on.

Use the following syntax to declare and define a predicate:

```
predicate predicate_name { boolean_expression }
```

Argument	Description
<i>predicate_name</i>	The name of the new predicate.
<i>boolean_expression</i>	The Boolean expression that defines the predicate.

You can define predicates inside or outside protocol definitions. You can also associate a new predicate with a previously defined protocol; see “Extending Protocol Definitions” on page 401.

Because the compiler does not allow forward references, you must define a predicate before you can refer to it. You can refer to a previously defined named predicate in a protocol definition, in another predicate, in a search, or in a rule.

If an inactive protocol or its field is used in the expression, that part of the expression evaluates to 0, or `FALSE`. The name of an active protocol evaluates to `TRUE`.

Example

In the following example, the second predicate, `isNewTelnet`, makes use of the first predicate, `isTcpSyn`:

```
predicate isTcpSyn { tcp && (tcp.flags & 0x02) != 0 }
predicate isNewTelnet { isTcpSyn && (tcp.dport == 23) }
```

Sets and Named Searches

NCL and the Action Services Library (ASL) together support data tables called *sets*. Sets associate application-defined data with packets. You define *named searches* associated with a specific set, which determine whether the current packet has a matching element in the set, based on the values of specified fields.

Searchable sets define collections of packets which are associated with each other by virtue of their *contents*. If you want to form collections on *structural* criteria, such as “the set of all packets with IP header lengths greater than twenty bytes,” use a classification predicate rather than a searchable set.

The set is implemented as a data table in the Policy Accelerator, and is created on initialization. Each element in the set must contain the specified number of key values, followed by any amount of arbitrary data. The named searches defined for a given set identify the correspondence between the key values and protocol fields.

You use NCL to declare the existence and suggest the size of a set, and to define the searches that evaluate set membership. You modify the *contents* of sets using actions. Actions can retrieve data from a set or place data in a set, based on the results of searches. For more information, see “Set Management Classes” on page 101 in Chapter 4, “Action Services Library,” and Chapter 9, “Using Sets of Data to Classify Packets,” in *Developing Applications Using the IX-API SDK*.

Every set and search that you define in NCL must correspond to a set and search defined in action code. To ensure this, you generate action code directly from your NCL code; see “Synchronizing NCL with Action Code” on page 411.

Defining a Set

Use the keyword `set` to declare and name a set, specify the number of key values, and suggest the size. The syntax is as follows:

```
set set_name
  < nkeys > {
    size_hint { expected_population }
  }
```

Argument	Description
<i>set_name</i>	The name of the set.
<i>nkeys</i>	The number of keys for any search on the set. Key values are four bytes or less in length. If the field values corresponding to the keys are longer, split them into multiple keys. You can specify a maximum of seven keys.
<i>expected_population</i>	<p>The number of members you expect the set to have. Must be a power of two: 1024, 2048, 4096, 8192, 16384, 32768, or 65536.</p> <p>This number does not place a strict limit on the population of the set. However, as the set size grows beyond the hint value, the search time might slowly increase.</p> <p>This clause is optional.</p>

Choosing the Size Hint

The size hint is based on the number of elements to be used in a set. When choosing the size hint, balance the amount of memory consumed for the set against the search performance. Too large a size consumes memory unnecessarily. For example, before adding any elements, a set with a size hint of 65536 consumes nearly 0.5 Mb of RAM. Too small a size could hurt search performance.

The maximum allocation of memory for a set is 64KB. A size hint larger than 65536 results in the maximum allocation of 64KB.

Defining Named Searches

A named search is associated with a specific set. You can define any number of different searches for the same set. You can also define similar searches (using the same key values) on different sets.

You define a named search using the keyword **search**. The syntax is as follows:

```
search set_name.search_name
  (key1, key2, ... keyn) {
    requires { boolean_expression }
  }
```

Argument	Description
<i>set_name</i>	The name of the set to search.
<i>search_name</i>	The name of the new named search being defined.
<i>keyn</i>	The names of the key fields, in the form <i>protocol_name.field_name</i> . The number of key fields must match the number of keys (<i>nkeys</i>) defined for the set.
requires { <i>boolean_expression</i> }	When the Boolean expression is TRUE , perform the search. When it is FALSE , do not perform the search. The requires clause is optional.

A search returns **TRUE** when an element is present in the associated set that matches the specified key values for the current packet. When it succeeds, the search also identifies the matching element.

You can use the **requires** keyword to include an optional Boolean expression in the named search that controls whether the search is performed.

- If the **requires** clause succeeds, the search is performed, and returns a result based on whether a match is found.
- If the **requires** clause fails, the search is not performed. The search returns **FALSE**, regardless of whether there is a matching element in the set.

You can use a named search in subsequent predicates as a Boolean expression. You can use searches on both the left and right sides of rules; they are interpreted differently on each side. For more information, see “Rules and Actions” on page 406.

Executing Searches

For each incoming packet, the Policy Accelerator tries every search defined in the NCL rules file. If the **requires** clause succeeds, it executes the search and stores the result in the corresponding ASL *Search* object.

A search can have one of the following results:

- The search did not run because the requirements were not met.
- The search ran and found a matching element.
- The search ran and did not find a matching element.

Examples

In the following example, the predicate `tcp_sport_in` is defined to be the Boolean result of the named search `tuports.tcp_sport`, which determines whether the `tcp.sport` field (source port) of a TCP segment is in the set `tuports`.

```
predicate tcp_sport_in {tuports.tcp_sport}
```

In the following example, the predicate determines whether a TCP segment is a member of the set `tuports` using both the source and destination port values.

```
predicate tcp_port_in {tuports.tcp_sport && tuports.tcp_dport}
```

In the following example, the predicate determines whether a UDP datagram is a member of the set `tuports` by virtue of either the source or destination port value.

```
predicate udp_sdports_in {tuports.udp_sport || tuports.udp_dport}
```

The following example defines a set of transport-layer protocol ports (TCP or UDP), illustrating how to use one set for multiple searches. The set `tuports` might contain a collection of port numbers of interest for either protocol (TCP/IP or UDP/IP). The four named searches provide checks to determine whether different TCP or UDP source or destination port numbers are present in the set.

```
#define MAX_TCP_UDP_PORTS_SET_SZ 200
/* TUPORTS: a set of TCP or UDP ports */
set tuports<1> {
    size_hint { MAX_TCP_UDP_PORTS_SET_SZ }
}
search tuports.tcp_sport (tcp.sport)
search tuports.tcp_dport (tcp.dport)
search tuports.udp_sport (udp.sport)
search tuports.udp_dport (udp.dport)
```

Rules and Actions

You must create rules to trigger the actions that direct and manipulate packet data. A rule specifies an action to perform based on the protocol classification and predicate or search results when the rule is applied to the current packet. The rules in an NCL rules file can refer only to those action functions defined in the action file that is part of the same ACE.

You specify the actions to which the rules apply using C++ and the Action Services Library (ASL) in an action source file, compiled to a `.nbo` file. For more information on defining action functions, see “Action Functions” on page 115 in Chapter 4, “Action Services Library,” and Chapter 7, “Acting on Packets in Your Action Code,” in *Developing Applications Using the IX-API SDK*.

The final section of your NCL rules file contains the rules for your application, which can use the previously defined protocols, predicates, and searches. Rules are evaluated in the order in which they are specified in the NCL rules file.

Defining Rules Use the `rule` keyword to declare and define a named rule. The syntax is as follows:

```
rule rule_name { predicate } {  
    external_action_fn (arg1, arg2, ...)  
}
```

Argument	Description
<code>rule_name</code>	The name of the rule.
<code>predicate</code>	A Boolean expression consisting of any combination of individual Boolean sub-expressions or predicate names.
<code>external_action_fn</code>	The name of the action function to call when the predicate is <code>TRUE</code> for the current packet. The named action must be defined as an entry point in the <code>.nbo</code> file that is part of the same ACE.
<code>arg1 ...</code>	Arguments for the specified action function. These values are passed to the action code in the <code>.nbo</code> file. You must pass the number of arguments required by the specified action, as specified in its definition.

A rule has a name and two parts, the left side, or predicate part, and the right side, or action part. A rule succeeds if its predicate part evaluates to `TRUE` for the current packet being processed. The action part of the rule indicates which processing function, or action, to apply to the packet when the rule succeeds. Only actions specified by successful rules are executed.

You can use a named search on either side of a rule. How it is used depends on which side of the rule it is on:

- When used on the left side of a rule as part of the predicate, the search acts as a Boolean expression. It succeeds when the `requires` clause is `TRUE` and the search finds a matching record in the set.

- When used on the right side of a rule as an action argument, the search returns a search result object that contains a pointer. When the search has succeeded, the returned search object contains a pointer to the matching record in the set. When it has failed, the search object contains a pointer to a location at which a new record can be inserted in the set. (For more information, see “Set Management Classes” on page 101 and “Search Class” on page 251 in Chapter 4, “Action Services Library.”)

Passing Action Arguments

The arguments that you pass to an action are determined by the definition of the specified action function. The first two arguments of any action function (`buf` and `ace`) are supplied automatically; you do not pass them in the rule. If any additional arguments are defined for the action function, you must pass them.

You are responsible for passing the right number of arguments, and arguments of the right type. You cannot pass expressions with operators as arguments. The following table shows the kinds of arguments you can specify, and how they are passed to the action function.

Argument type	Example	How passed
Protocol field	<code>ip.src</code>	NCL passes the value of the specified field in the current packet. If the value is 32 bits or shorter, it passes the value directly. If the value is longer than 32 bits, it passes a reference to the value.
Protocol name	<code>ip</code>	NCL passes a pointer to where the specified protocol starts in the current packet.
Predicate	<code>ip.bcast</code>	NCL evaluates the predicate and passes the Boolean value (<code>TRUE</code> or <code>FALSE</code>).
Intrinsic	<code>tcp.chksumvalid</code>	NCL evaluates the intrinsic function and passes the Boolean or 32-bit value.
Constant	<code>23</code>	NCL passes the constant value directly.
Search	<code>tuports.tcp_sport</code>	Identifies the corresponding Search object in action code, which contains the result of the search.



NOTE: Due to limitations in the `gcc` compiler, it is recommended that you avoid the use of type `bool` arguments in action functions. Use `unsigned int` instead.

Example

The following example defines two sets and two named searches. The first set contains source and destination IP addresses, plus TCP ports. The other set contains IP addresses and UDP ports. The first search uses the IP source and destination addresses and the TCP destination port number as keys. The second search uses the IP source and destination addresses and UDP destination port as keys.

The predicate `ipValid` checks to make sure the packet is an IP packet with valid checksum, has a header of acceptable size, and is IP version 4. The predicate `newtelnet` determines if the current TCP segment is a SYN packet destined for a telnet port. The predicate `tftp` determines if the UDP destination port corresponds to the TFTP port number and the combination of IP source and destination addresses and destination UDP port number is in the set `ip_udp_ports`.

- The first rule, `telnetNewCon`, determines whether the current segment is a new telnet connection and specifies that the associated external function `start_telnet` is invoked when this rule is `TRUE`. The function takes the search result as an argument.
- The second rule, `tftppkt`, checks whether the packet is a valid TFTP connection. If so, the associated action `is_tftp_pkt` is invoked with `udp.dport` as the argument.
- The third rule, `addnewtelnet`, checks if the current segment is a new telnet connection. If so, the associated action function `add_to_tcp_pkt_count` is invoked with no arguments.

```
set set_ip_tcp_ports <3> {
    size_hint { 100 }
}
set set_ip_udp_ports <3> {
    size_hint { 100 }
}

search set_ip_tcp_ports.tcp_dport ( ip.src, ip.dst, tcp.dport )
{
    requires {ip && tcp}
}
search set_ip_udp_ports.udp_dport ( ip.src, ip.dst, udp.dport )
{
    requires {ip && udp}
}

predicate ipValid {
    ip && ip.chksumvalid && (ip.hlen > 5) && (ip.ver == 4)
}
predicate newtelnet {
    (tcp.flags & 0x02) && (tcp.dport == 23)
}
```

```

predicate tftp {
    (udp.dport == 21) && set_ip_udp_ports.udp_ports
}

rule telnetNewCon {ipValid && newtelnet &&
set_ip_tcp_ports.tcp_dport }
    { start_telnet( set_ip_tcp_ports.tcp_dport) }
rule tftppkt {ipValid && tftp }
    { is_tftp_pkt ( udp.dport ) }
rule addnewtelnet { newtelnet }
    { add_to_tcp_pkt_count() }

```

Conditional Rule Execution

A clause introduced by the keyword **with** provides conditional execution for groups of rules and/or predicates. The syntax is as follows:

```

with boolean_exp {
    predicate pred_name { boolean_exp }
    rule rule_name { predicate } { action_ref }
}

```

The clause can include any number of predicate and/or rule definitions, and can also include nested **with** clauses.

- If the initial boolean expression evaluates to **FALSE**, all of the enclosed predicates and rules also evaluate to **FALSE**.
- If the initial boolean expression evaluates to **TRUE**, the enclosed definitions are each evaluated as if they were defined at the top level.

Example

The following example uses a **with** clause to evaluate the validity of an IP datagram. If the IP datagram is valid, the predicate **newtelnet** is defined, and the rule **telnetNewCon** is evaluated. If the IP datagram is not valid, NCL skips both the predicate definition and rule.

```

predicate tcpValid { tcp && tcp.chksumalid }
with (tcpValid) {
    predicate newtelnet {(tcp.flags & 0x02)
                        && tcp.dport == TELNET }
    rule telnetNewCon { newtelnet && ip_tcp_ports.tcp_dport }
                        { start_telnet( ip_tcp_sport.tcp_dport) }
}

```

The following example uses a **with** clause to evaluate the validity of an IP datagram. If the IP datagram is valid, NCL evaluates a set of predicates and rules which includes a nested **with** clause:

```
predicate tcpValid { tcp && tcp.chksumvalid }
#define TELNET 23 /* port number for telnet */
with ipValid {
    predicate tftp { (udp.dport == 21) && ip_udp_ports.udp_dport }
    with tcpValid { /* Nested with */
        predicate newtelnet {(tcp.flags & 0x02)
                               && tcp.dport == TELNET }
        rule telnetNewCon { newtelnet && ip_tcp_ports.tcp_dport }
        { start_telnet( ip_tcp_port.tcp_dport) }
    }
    rule tftppkt { tftp && ip_udp_ports.udp_dport }
    { is_tftp_pkt ( udp.dport ) }
}
```

Synchronizing NCL with Action Code

Each NCL rules file that you write corresponds to one and only one action file that is part of the same ACE. The action file that corresponds to an NCL rules file must use certain information that is contained in the NCL file:

- **Sets and Searches:** The sets and searches that you define in the NCL rules file must be defined in exactly the same way in C++ for the corresponding action file. To ensure accuracy, you generate the C++ definitions directly from the NCL definitions.
- **Field Accessors:** When the action code has its own access methods for protocol fields, it is not necessary for a rule to pass every field value that an action might need. The field information needed to create accessors is found in the protocol definition in the NCL rules file.

You generate the information needed by the action file directly from the corresponding NCL rules file, using the NCL compiler (`cecomp.exe`) as a standalone tool with command-line options. The compiler generates C++ source code files that you then include as header files in the source for the action code.

Generating Sets and Searches

Use the following command to generate the C++ code for sets and searches:

```
cecomp -Fsfilename sourcefile.ncl
```

The argument *filename* specifies the name of the file to which to write the generated C++ code.

The compiler generates ASL base classes and objects for each of the sets and named searches in the NCL rules file, and also defines a data structure for each set.

Generating Field Accessors

Use the following command to generate the C++ code for field accessors:

```
cecomp -Ffilename sourcefile.ncl
```

The argument *filename* specifies the name of the file to which to write the generated C++ code.

The compiler generates a class for each defined protocol, with accessor methods for each field. These accessors are defined to return results in network byte order. (See “Byte Order Issues” on page 10 in Chapter 2, “System Types and Methods.”)



NOTE: If the NCL code dynamically modifies the protocol definitions, the generated field accessors can become incorrect. In this case, you must pass field values to action functions from rules.

Chapter 7

Command-Line Tools



The IX-API SDK includes command-line tools for compiling and debugging application code, for various utilities, and for starting parts of the system. This chapter describes the syntax and options of the following command-line tools and utilities, listed in alphabetical order.

Command	Description
<code>cecomp</code> Command	Compiles NCL code.
<code>celink</code> Command	Links compiled NCL code.
<code>getaceid</code> Command	Creates a symbol table for the action code debugger.
<code>nbgcc</code> Command	Compiles C++ action code.
<code>nbgdb</code> Command	Starts the command-line debugger for action code.
<code>nbld</code> Command	Links compiled action code.
<code>odxloop</code> Command	Aids in verifying the operation of a customized NIC driver that uses Optimal Data Exchange (ODX) Protocol for PCI.
<code>pa100diag</code> Command	Verifies the hardware installation of the Policy Accelerator 100. (In the <code>diagnostics</code> directory.)
<code>readport</code> Command	On UNIX only, reads and displays output sent to <code>stdout</code> or <code>sysout</code> .
<code>resolver</code> Command	Starts the Resolver, which is the resource manager for the Policy Accelerator and related objects.

Tool Locations

All utilities and system tools are located in the directory `SDKinstallpath/bin`, with the following exceptions:

- The `nbgcc` compiler is located in the directory `/usr/local/bin`.



- Diagnostic tools are located in the directory *SDKinstallpath/diagnostics*.


cecomp Command

Compiles NCL code.

```
cecomp -v -Faprotofilename -Fsetfilename
-Ftsetfilebasename sourcefile.ncl
```

Option	Meaning
-v	Optional. Turn on verbose mode. When on, prints compilation information to <code>stdout</code> .
-Faprotofilename	<p>Optional. Generates the C++ code for field accessors. The argument <i>protofilename</i> specifies the complete name of the file to which to write the generated C++ code; for example, <code>myproto.h</code>.</p> <p>The compiler generates a class for each protocol defined in the NCL file, with accessor methods for each field. These accessors are defined to return results in network byte order. See “Byte Order and Intermodule Communication” in Chapter 2, “System Types and Methods.”</p> <p>NOTE: If the NCL code dynamically modifies the protocol definitions, the generated field accessors can become incorrect. In this case, you must pass field values to action functions from rules.</p>
-Fsetfilename	<p>Optional. Generates the C++ code for sets and searches. The argument <i>setfilename</i> specifies the complete name of the file to which to write generated C++ code; for example, <code>myset.h</code>.</p> <p>The compiler generates a file named <i>setfilename</i>, which defines ASL base classes and objects for each of the sets and named searches in the NCL file, and also defines a data structure for each set.</p>



Option	Meaning
<code>-Ftsetfilename</code>	<p>Optional. Generates the C++ code for sets and searches. The argument <i>setfilename</i> specifies the base name of the files to which to write generated C++ code; for example, <i>myset</i>.</p> <p>The compiler generates two header files. The first file, named <i>setfilename.h</i>, contains the same information as the single file generated by the <code>-Fs</code> option. In addition, it creates a second file named <i>setfilename_def.h</i>, which initializes the static data members of the sets.</p> <p>Use this option rather than <code>-Fs</code> if your accelerator module has more than one action file using the same sets. Before linking, ensure that there is only one reference in the include tree to the <i>setfilename_def.h</i> file. If there is more than one reference, or if you use the <code>-Fs</code> option, the linker generates errors when the static data members are multiply defined.</p>
 NOTE: Do not modify the set header files created by the NCL compiler; any changes you make will be overwritten the next time you generate. Extend the set element definitions in an action source file that includes these headers.	
<code>sourcefile.ncl</code>	Required. The name of the source NCL file to compile, which must have the extension <code>.ncl</code> .

Description

NCL code is normally compiled and linked at run time by the Policy Accelerator system. You use this command independently for the following reasons:

- Generate the C++ code for sets and searches, to be included as a header in your action code.
- Generate the C++ code for protocol field accessors, to be included as a header in your action code.
- Test your NCL code for compilation errors and object size. In this case, use `celink` to link the resulting file.

Because you do not normally compile NCL code for use in an application, the options that control compilation are not documented here. For information on using `cecomp` to compile NCL for an SDK-E application on an embedded system, see *IX-API SDK Host API Reference Supplement for Embedded Systems*.

The Resolver does not need to be running to use this command.

See Also

- “celink Command” on page 417
- Chapter 4, “Compiling Applications,” in *Developing Applications Using the IX-API SDK*
- “Synchronizing NCL with Action Code” on page 411

celink Command

Links compiled NCL code.

```
celink -ce ceId -nomrt -o outfile -v objectfile [objectfiles ]
```

Option	Meaning
-ce ceId	Unsupported. For Intel internal use only.
-nomrt	Unsupported. For Intel internal use only.
-o outfile	Optional. The filename for the generated executable. Default is a.out.
-v	Optional. Turn on verbose mode. When on, prints instruction count to stdout.
objectfile	Required. The name of the compiled NCL file or files to link, as written by the cecomp command.

Description

NCL code is normally compiled and linked at run time by the Policy Accelerator system. You can use this command independently to test your NCL code for compilation errors and object size. In this case, use `celink` to link the file produced by `cecomp` with the startup and runtime code that resides in the Classification Engine within the Policy Accelerator.

The command links NCL object files in the order in which they are specified. Because the compiler generates calls to set search functions, the linker scans the object file for compiler-generated symbols to determine the objects to link in.

For information on using `cecomp` and `celink` to compile NCL for an SDK-E application on an embedded system, see *IX-API SDK Host API Reference Supplement for Embedded Systems*.

The Resolver does not need to be running to use this command.

See Also

- “cecomp Command” on page 414
- Chapter 4, “Compiling Applications,” in *Developing Applications Using the IX-API SDK*

getaceid Command

Creates a symbol table for the action code debugger.

getaceid *ACEpathname actionfilename*

Argument	Meaning
<i>ACEpathname</i>	The complete path to the ACE you want to debug, as it would appear in the bind function of your application code (<i>appname.cpp</i>) and as described in Appendix C, “Policy Accelerator Name Space.”
<i>actionfilename</i>	<p>The file name of the compiled action code for the ACE (without its extension).</p> <p>The action code must have been compiled with the <code>nbgcc</code> debug option (<code>-g</code>) and with no compiler optimizations (<code>-O</code>).</p>

Description

The command prints the ACE’s ID (an integer) to `stdout`, and also creates a compiled and linked executable file *actionfilename.exe* in the current directory. You run the debugger (`nbgdb`) on this file.

The debugger runs on your host system, while an ACE normally runs on the Policy Accelerator. You do not have direct access to the Policy Accelerator from a command shell. To debug a particular ACE, you need a standalone executable for that ACE that you can run on the host from a command shell.

This utility produces the standalone executable in the current directory. You can run the debugger on this file to step through the action functions as if it were really running in the Policy Accelerator.

The Resolver must be running to use this command.



NOTE: To use this command, you must have the debug card installed on your Policy Accelerator. In addition, you must have the entire SDK installed on the computer on which you are running the debugger.

Example

```
getaceid /NBloopAppl/NBloopAceGroup/NBloopAce loopact
```

See Also

- Chapter 11, “Debugging and Troubleshooting,” in *Developing Applications Using the IX-API SDK*
- “nbgdb Command” on page 421

nbgcc Command

Compiles C++ action code.

```
nbgcc -c -g -Ooptlevel -D_byteorder -ISDKinstallpath/include\  
actionsSource.cpp
```

Option	Meaning
-c	Generate object code without linking. Always specify this option. To link multiple action files for a single ACE, compile them using this option, then pass them to the linker (nbld) to produce the single object for the ACE.
-g	Compile with debugging symbol table. Required when you intend to use nbgdb with the resulting object code.
-Ooptlevel	The optimization level to use on the code. A value of 0 means no optimization, which is best for debugging. A value of 2 means maximum optimization, which is best for highest performance.
-D_BIGENDIAN	Specifies that data values on the Policy Accelerator are stored most-significant-byte first. Always specify this option; it ensures that network byte order is correctly used.
-ISDKinstallpath/include	Specifies where to look for header files.
actionsSource.cpp	The source file to be compiled.

Description

To compile action code files, the IX-API SDK provides a compiler (nbgcc) that is a modified version of the GNU C++ compiler (gcc). The nbgcc compiler is located in the directory /usr/local/bin.

The command produces a file named *actionsSource.o*.

- If you have a single source file, rename the .o file with a .nbo extension and pass it to the ACE manager's load method.
- If you have more than one source file for action code for a single ACE, compile each file and pass the resulting objects to the IX-API SDK extension of the GNU linker, nbld.

The Resolver does not need to be running to use this command.

See Also

- *Using and Porting GNU CC* in the `/usr/local/docs` directory for more information on options available during compilation, and on the basic usage of the IX-API SDK versions of these tools
- Chapter 4, “Compiling Applications,” in *Developing Applications Using the IX-API SDK*.
- “nbld Command” on page 423

nbgdb Command

Starts the command-line debugger for action code.

`nbgdb actionfilename`

Argument	Meaning
<i>actionfilename</i>	The file name of the executable file produced by <code>getaceid</code> for the action code of the ACE.

Description

The IX-API SDK provides a command-line debugging tool that you can use to debug your action code. The debugger, `nbgdb`, is a version of the GNU debugger (`gdb`) that has been customized for use with the Policy Accelerator. Use it to debug action code written in C++, C, or StrongARM 110 assembly that runs on the Policy Accelerator.



NOTE: To use `nbgdb`, you must have a debug card attached to your Policy Accelerator. In addition, you must have the entire IX-API SDK installed on the computer on which you are running the debugger.

To use the IX-API SDK debugger, you must compile the action code with the `nbgcc` debug option (`-g`) and with no compiler optimizations (`-O`). You must make the application with the `nmake mode=Debug` option. For additional debugging information, you can start the Resolver in verbose mode using the `-v` switch.



NOTE: With compiler optimizations of any level turned on, the debugging information generated by the compiler might be inaccurate.

The debugger runs on your host system, while an ACE normally runs on the Policy Accelerator. You do not have direct access to the Policy Accelerator from a command shell. To run the debugger on a particular ACE, you need a standalone executable for that ACE that you can run on the host from a command shell.

To produce a standalone executable for an ACE, use the utility `getaceid`. See “`getaceid` Command” on page 418.

After starting the debugger, you must connect to the target ACE, using the ID returned by `getaceid`:

```
(nbgdb) target remote com1:ACEid
```

For information on the debugger commands, see the *GDB User's Guide* in the `/usr/local/docs` directory. The following standard `gdb` commands are not supported in `nbgdb`:

- `run`, `load`, and related commands
- Commands related to watchpoints and tracepoints
- Thread- and task-related commands

If you encounter any unexpected problems while using `nbgdb` to debug your application, restart the Resolver, your application, and `nbgdb`.

Stepping through Action Functions

Because you run the debugger on the host, rather than on the Policy Accelerator itself, debugging information is available only for the action functions themselves. The vast majority of processing time in an application is normally spent in the Policy Accelerator system code, not in these asynchronous action functions.

When single-stepping through action code, you must take care not to single-step past the end of any action function, out of the scope of the action. Instead, always use the `continue` command of `nbgdb` at the end of an action function.



NOTE: Debugging of the action code initialization function, `init_actions`, is not supported. Do not introduce a breakpoint or other pause into this function.

Shutting down the Debugger

When you have finished stepping through action functions, you must use the `delete` command to remove all breakpoints before quitting from the debugger.

See Also

- “`getaceid` Command” on page 418
- Chapter 11, “Debugging and Troubleshooting,” in *Developing Applications Using the IX-API SDK* for information on debugging techniques
- “`resolver` Command” on page 427


nbld Command

Links compiled action code.

```
nbld -r -EB -o outputname actionsSource1.o actionsSource2.o ...
```

Argument	Meaning
-r	Required. Generate relocatable output that can, in turn, be used as input for the IX-API SDK's action code linker/loader.
-EB	Required. Link big-endian objects.
-o outputname	Optional. The name of the output file to generate. You must rename the output file to have the .nbo extension. By default, the command produces a file named a.out.
actionsSource<n>.o	Two or more object files to be linked, as produced by nbgcc using the -c option.

Description Pass the output file (which must have the .nbo extension) to the ACE manager's load method to load it into the Policy Accelerator. If you have more than one ACE, ensure that the .nbo file names are unique for each ACE.

 **NOTE:** Use this command only when you have more than one action code file for a single ACE. Otherwise, simply rename the .o file produced by nbgcc.

The Resolver does not need to be running to use this command.

- See Also**
- GNU ld documentation, available in the /usr/local/docs directory, for more information on options available during linking
 - “nbgcc Command” on page 419
 - Chapter 4, “Compiling Applications,” in *Developing Applications Using the IX-API SDK*

odxloop Command

Aids in verifying the operation of a NIC driver that has been customized to use ODX for communication with a Policy Accelerator.

odxloop

Description After starting your customized NIC driver, execute this command in a command shell. This utility configures the Policy Accelerator to reflect packets back to the NIC. To verify that the NIC driver customization was accomplished correctly and that the NIC and Policy Accelerator communicate properly, send packets to the NIC and verify that they are returned in the same order and condition.

This diagnostic tool is located in the *SDKinstallpath/diagnostics* directory.

You must have installed both the NIC and the Policy Accelerator device drivers to perform these tests. After performing the tests, reboot the system to return the Policy Accelerator configuration to its default state.



NOTE: This is a diagnostic utility intended to test your customized NIC driver, and is not a usage demonstration. A sample application is included in the installation at the following location:

SDKinstallpath/demo/ODXFilter

See Also *Customizing a NIC Driver Using the ODX Protocol*

pa100diag Command

Verifies the hardware installation of the Policy Accelerator 100.

```
pa100diag -loopback -pa paname
```

Argument	Meaning
-loopback	Optional. When specified, run the hardware loopback tests of the board's MAC interfaces. To run the loopback tests, you must connect the A and B interfaces to one another using a CAT4-compliant ethernet crossover cable.
-pa <i>paname</i>	Optional. When multiple Policy Accelerators are installed, specifies which Policy Accelerator to test. For example, -pa nbhwpe1. The default is nbhwpe0.

Description

In a command shell, execute this command to verify that the hardware installation was accomplished correctly and that the specified board operates properly. This diagnostic tool is located in the *SDKinstallpath/diagnostics* directory.

You must have installed the Policy Accelerator device driver to perform the tests.



NOTE: You cannot use this tool after starting the Resolver, even if you exit from it. If you have ever started the Resolver, you must reboot, then run the tool before starting the Resolver.

This script runs a number of hardware tests. If any test fails, the script terminates. At the end of testing the script displays the final result in the command shell. If all tests pass, it displays the message:

```
***** TEST RESULTS *****
***** serial_number PASSED DIAGNOSTICS TEST *****
```

If any test fails, it displays the message:

```
***** TEST RESULTS *****
***** serial_number FAILED DIAGNOSTICS TEST *****
```

Each time you run the diagnostic tool, it writes a log file of the test results to the current working directory. The text log file, named *serial_number.log*, contains messages about the results of each individual test. In the event of failure, return the board and the log file to the manufacturer.

See Also

- Chapter 2, “Troubleshooting the Policy Accelerator,” of *Installing a Policy Accelerator 100 Board*
- Contact Intel Technical Support for information on testing non-standard boards or special-purpose testing. See “Contacting Intel” on page xxii.

readport Command

On UNIX only, reads and displays output sent from the Policy Accelerator to `stdout` or `sysout`.

readport *port*

Argument	Meaning
<i>port</i>	The port from which to read: 11 or 12. 11 = <code>stdout</code> 12 = <code>sysout</code>

Description

In a UNIX command shell, execute this command to display output in that shell from the specified port.

- When an accelerator module sends output to `stdout` from commands such as `fprintf`, read the results on port 11.
- Read system output from the Policy Accelerator, such as error and warning messages, on port 12.



NOTE: On a Windows NT system, use the utility WinReadPort to display output from both ports in a window. Invoke this utility from the IX-API SDK system menu.

See Also

Chapter 11, “Debugging and Troubleshooting,” in *Developing Applications Using the IX-API SDK*

resolver Command

Starts the Resolver, which is the resource manager for the Policy Accelerator and related objects.

```
resolver -v -k -l -x -c
```

Option	Meaning
-v	Turn on verbose mode for debugging. This prints information on Resolver actions to <code>stdout</code> and <code>stderr</code> .
-k	Do not load and start the Policy Accelerator system software. If you specify this option, you must load and start the Policy Accelerator system software manually.
-l	Use compiler DLLs to compile NCL at run time. This is the default; it is not necessary to specify it.
-x	Unsupported. For Intel internal use only.
-c	Turn on verbose mode for NCL runtime compilation. This prints information on NCL compilation to <code>stdout</code> and <code>stderr</code> .


Description

The Resolver has the following responsibilities:

- Maintains a database of available system resources and allocates them to applications
- Initializes the Policy Accelerator
- Maintains a database of objects created by all IX-API SDK applications and their associations with objects in the Policy Accelerator
- Manages the creation of objects on the Policy Accelerator
- Dynamically invokes the NCL compiler to compile classification code

Starting and Stopping the Resolver

To run an application, the Resolver must be running. When you install the IX-API SDK or a runtime IX application, you normally configure the host computer to start the Resolver process automatically at startup. If you need to start and stop the Resolver, you have the following options:

- Start the Resolver manually in a command shell using the `resolver` command, and stop it using an operating system command, such as Control-C, in that shell.
- Start and stop the Resolver process programmatically, using functions defined in the operating system services layer (OSSL) library. An example of this is provided in the Killer demo. See Appendix A, “Demonstration Applications,” in *Developing Applications Using the IX-API SDK*.
- On a Windows NT system, start and stop the Resolver interactively, using the icon  in the lower right corner of the desktop.
 - Right-click on the icon and choose Start Resolver from the pop-up menu. When the Resolver starts, the icon becomes colored and moving your mouse cursor over the icon displays “Resolver is running.” This starts the Resolver with no options.
 - To stop the Resolver, choose Stop Resolver from the pop-up menu. The icon grays out, and moving your mouse cursor over the icon displays “Resolver is NOT running.”

When the Resolver is running, you run an IX application by executing its host module executable file, as produced by `gcc`. The host module loads the files needed for the accelerator module.

Stopping and Restarting Applications

The Resolver allows you to run multiple IX applications simultaneously. To do so, it maintains a set of application resources. However, when you stop an application, the Resolver does not always free enough resources to restart it (or start other applications) reliably.

If you intend to stop any application, then restart that application or start other IX applications, it is recommended that you stop all running applications, then stop and restart the Resolver before starting or restarting any IX application.

See Also

- Chapter 3, “Elements of an Application,” in *Developing Applications Using the IX-API SDK*

Appendix A

IX-API SDK Host API Error Codes



Alphabetical Listing

The file `nberror.h` contains all IX-API SDK host API error code constants. However, only a few of these codes are visible to applications. The following table presents the visible codes in alphabetical order along with a description of each.

Error codes are encapsulated by the `NBError` class. An `NBError` object is returned by host API methods, and thrown by host API object constructors. You use the `getErrorCode` method to access the error code in the object. For more information, see “NBError Class” on page 81.

Return Codes	Description
NBERROR_ACEGROUP_CANNOTREGISTER	Cannot register this ACE group with the Resolver for one of these reasons: <ul style="list-style-type: none">■ The Resolver is not running.■ The Policy Accelerator driver is not running.■ There is a problem communicating with the Policy Accelerator.■ The name has already been registered.
NBERROR_ACEGROUP_CANNOTUNREGISTER	Resolver cannot unregister this ACE group.
NBERROR_ACEGROUP_NULLAPPL	The <code>argAppl</code> argument is <code>NULL</code> .
NBERROR_ACEMGR_CANNOTCREATEDROP	Cannot create a default drop target.
NBERROR_ACEMGR_CANNOTCREATEPASS	Cannot create a default pass target.



Return Codes	Description
NBERROR_ACEMGR_CANNOTDEVREG	Cannot register this ACE manager with the Policy Accelerator kernel driver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_ACEMGR_CANNOTDEVUNREG	Resolver cannot unregister this ACE from the Policy Accelerator kernel driver.
NBERROR_ACEMGR_CANNOTRECVERRMSG	Cannot retrieve NCL compiler error messages because communication with the Resolver was interrupted during this operation.
NBERROR_ACEMGR_CANNOTRECVFNACK	Cannot receive the message from the Resolver acknowledging that the ACE was loaded.
NBERROR_ACEMGR_CANNOTREGISTER	Cannot register this ACE manager with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_ACEMGR_CANNOTRELOADACTIONS	Cannot reload actions file (.nbo). The actions file must have a valid Win32 filename.
NBERROR_ACEMGR_CANNOTSENDN	Cannot send a message to the Resolver to load this ACE manager.
NBERROR_ACEMGR_CANNOTUNREGISTER	Resolver cannot unregister this ACE manager.
NBERROR_ACEMGR_FNTOOLONG	Filename is too long. The maximum length of the filename is MAX_FILENAME_LENGTH, which is defined in NBapi\nbparam.h.
NBERROR_ACEMGR_INVACEMODE	Invalid ACE mode specified for argAceMode. Valid ACE modes are ACE_READER and ACE_WRITER. Check your spelling.

Return Codes	Description
NBERROR_ACEMGR_INVALIDCMPLRERRMSG	Cannot retrieve valid NCL compiler error messages. No error messages are currently available.
NBERROR_ACEMGR_INVFN	Invalid filename.
NBERROR_ACEMGR_NULLACEGROUP	The <code>argAceGroup</code> argument is NULL.
NBERROR_ACEMGR_NULLAPPL	The <code>argAppl</code> argument is NULL.
NBERROR_ACEMGR_NULLFILENAMES	One of the filename pointers is NULL. You must specify filenames when loading an ACE.
NBERROR_ACEMGR_NULLRULES	One or both rule name pointers is NULL.
NBERROR_ACEMGR_OUTOFMEM	Cannot allocate memory to load code.
NBERROR_ACEMGR_OUTOFMEMORY	Cannot allocate memory to load code.
NBERROR_BOOT_CFGREADFAILED	Cannot read the area of the Policy Accelerator containing the configuration.
NBERROR_BOOT_FILENOTFOUND	One of the boot image files was not in the <i>SDKinstallpath\hpex</i> directory.
NBERROR_BOOT_PEINITFAILED	Unable to successfully initialize the Policy Accelerator.
NBERROR_BOOT_UNABLETOCREATEASYNCSTRUCT	Could not create needed resource for booting; most likely out of memory.
NBERROR_BOOT_UNABLETOCREATEBOOTPE	Out of memory instantiating the <code>CBootPE</code> class. <code>CBootPE</code> loads all the appropriate software on the Policy Accelerator and starts the execution of the ARM processor.
NBERROR_BOOT_UNABLETOOPENDRIVER	Driver is either not loaded or not communicating correctly.
NBERROR_BOOT_UNABLETOSETDRIVERSECURITY	Current user does not have access to communicate with the driver.
NBERROR_BOOT_UNABLETOWRITEPEBYTES	Boot process was unable to write data into the memory of the Policy Accelerator.
NBERROR_BOOT_UNABLETOWRITEPEREG	Boot process was unable to write data into the registers of the Policy Accelerator.

Return Codes	Description
NBERROR_CROSSCALL_CANNOTREGISTER	Cannot register this crosscall manager with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_CROSSCALL_CANNOTUNREGISTER	Resolver cannot unregister this crosscall manager.
NBERROR_CROSSCALL_NULLACEMGR	The <code>argAceMgr</code> argument is NULL.
NBERROR_CROSSCALL_NULLACEGROUP	The <code>argAceGroup</code> argument is NULL.
NBERROR_CROSSCALL_NULLAPPL	The <code>argAppl</code> argument is NULL.
NBERROR_CROSSCALLHANDLER_CANNOTREGISTER	Cannot register this crosscall handler with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_CROSSCALLHANDLER_CANNOTUNREGISTER	Resolver cannot unregister this crosscall handler.
NBERROR_CROSSCALLHANDLER_NULLACEGROUP	The <code>argAceGroup</code> argument is NULL.
NBERROR_CROSSCALLHANDLER_NULLACEMGR	The <code>argAceMgr</code> argument is NULL.
NBERROR_CROSSCALLHANDLER_NULLAPPL	The <code>argAppl</code> argument is NULL.
NBERROR_DOWNCALL_CANNOTOBTAINPECONTEXT	Cannot find downcall handler in the Policy Accelerator. Every downcall must have a corresponding downcall handler. Cannot issue downcall.

Return Codes	Description
NBERROR_DOWNCALL_CANNOTREGISTER	Cannot register this downcall with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_DOWNCALL_CANNOTSENDDOWNCALL	Kernel driver refused to send this downcall.
NBERROR_DOWNCALL_CANNOTUNREGISTER	Cannot unregister this downcall handler.
NBERROR_DOWNCALL_NULLACEGROUP	The <code>argAceGroup</code> argument is NULL.
NBERROR_DOWNCALL_NULLACEMGR	The <code>argAceMgr</code> argument is NULL.
NBERROR_DOWNCALL_NULLAPPL	The <code>argAppl</code> argument is NULL.
NBERROR_NBAPPL_CANNOTACCESSDEV	Application cannot connect with the Policy Accelerator device or driver. The device or driver is not properly initialized or installed or is not installed.
NBERROR_NBAPPL_CANNOTCREATEPIPE	Application cannot create a communication channel through which to talk to the Resolver.
NBERROR_NBAPPL_CANNOTCREATERSLVREQTHREAD	Application cannot create a thread dedicated to handle requests issued by the Resolver.
NBERROR_NBAPPL_CANNOTCREATEUPCALLTHREAD	Application cannot create a thread dedicated to handle upcalls from the Policy Accelerator.
NBERROR_NBAPPL_CANNOTRECVBINDREQ	Application cannot receive message from the Resolver acknowledging the bind request because it lost communication with the Resolver while executing this request.
NBERROR_NBAPPL_CANNOTRECVLINKREQ	Application cannot receive message from the Resolver acknowledging the link request because it lost communication with the Resolver while executing this request.
NBERROR_NBAPPL_CANNOTRECVOBJREGACK	Cannot receive message from the Resolver acknowledging registration of this object.

Return Codes	Description
NBERROR_NBAPPL_CANNOTRECVOBJUNREGACK	Cannot receive message from the Resolver acknowledging unregistration of this object.
NBERROR_NBAPPL_CANNOTRECVPECONTEXT	Cannot receive message from the Resolver to obtain the context data for a specific object in the Policy Accelerator.
NBERROR_NBAPPL_CANNOTRECVREGACK	Cannot receive message from the Resolver acknowledging registration of this application.
NBERROR_NBAPPL_CANNOTRECVUNBINDREQ	Cannot receive message from the Resolver acknowledging an unbind request.
NBERROR_NBAPPL_CANNOTRECVUNLINKREQ	Cannot receive message from the Resolver acknowledging an unlink request.
NBERROR_NBAPPL_CANNOTRECVUNREGACK	Cannot receive message from the Resolver to acknowledge unregistration of this application.
NBERROR_NBAPPL_CANNOTREGISTER	Cannot register this application with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_NBAPPL_CANNOTSENDBINDREQ	Application cannot send a message to the Resolver to request a binding.
NBERROR_NBAPPL_CANNOTSENDLINKREQ	Application cannot send a message to the Resolver to link the crosscall. The application cannot communicate with the Resolver.
NBERROR_NBAPPL_CANNOTSENDOBJREG	Cannot send message to the Resolver to register this object.
NBERROR_NBAPPL_CANNOTSENDOBJUNREG	Cannot send message to the Resolver to unregister this object.
NBERROR_NBAPPL_CANNOTSENDPECONTEXTREQ	Cannot send message to the Resolver to get the context data (PEContext) for a specific object in the Policy Accelerator.
NBERROR_NBAPPL_CANNOTSENDREG	Cannot send message to the Resolver acknowledging registration of the new application.

Return Codes	Description
NBERROR_NBAPPL_CANNOTSENDUNBINDREQ	Cannot send message to the Resolver to unbind the ACE.
NBERROR_NBAPPL_CANNOTSENDUNLINKREQ	Cannot send message to the Resolver to unlink the crosscall.
NBERROR_NBAPPL_CANNOTSENDUNREG	Cannot send message to the Resolver to unregister the application.
NBERROR_NBAPPL_CANNOTSERIALIZECBACKARGS	Resolver cannot process NBFactory callback arguments.
NBERROR_NBAPPL_CANNOTUNREGISTER	Resolver cannot unregister this application.
NBERROR_NBAPPL_DOUBLEERRUNREG	Not enough available memory to unregister this object.
NBERROR_NBAPPL_ERRBINDREQ	Resolver cannot execute the bind operation. Either the target or the destination ACE does not exist.
NBERROR_NBAPPL_ERRLINKREQ	Resolver cannot execute the link operation. Either the target or the destination ACE does not exist. If using the <code>unlink</code> command, this error means that the crosscall does not exist.
NBERROR_NBAPPL_ERRNBPIPE	Cannot allocate memory to create an object used to manage the named pipe used to communicate with the Resolver.
NBERROR_NBAPPL_ERRORDEVREG	Application cannot register with the Policy Accelerator device driver.
NBERROR_NBAPPL_ERRPECONTEXT	Resolver cannot get Policy Accelerator context data for the object.
NBERROR_NBAPPL_ERRUNBINDREQ	Resolver cannot execute the unbind operation.
NBERROR_NBAPPL_ERRUNLINKREQ	Resolver cannot execute the unlink operation.
NBERROR_NBAPPL_ETERMSLVREQTHREAD	Cannot terminate thread accepting Resolver messages.
NBERROR_NBAPPL_ETERMUPCALLTHREAD	Resolver cannot terminate the thread currently being used to handle upcalls for this application.
NBERROR_NBAPPL_INVHANDLE	Invalid Policy Accelerator context data sent from the Resolver.

Return Codes	Description
NBERROR_NBAPPL_INVTYPE	Application cannot operate on this type of object.
NBERROR_NBAPPL_NAMETOOLONG	Filename is too long. The maximum length of the filename is MAX_FILENAME_LENGTH, which is defined in NBapi\nbparam.h.
NBERROR_NBAPPL_NULLCMDLINE	Command line specified in constructor is NULL.
NBERROR_NBAPPL_NULLNAMES	Pointer that should contain the object name is NULL.
NBERROR_NBAPPL_OBJREGERROR	Resolver cannot register this object.
NBERROR_NBAPPL_OBJUNREGERROR	Resolver cannot unregister this object.
NBERROR_NBAPPL_OUTOFMEM	Out of memory.
NBERROR_NBAPPL_PATHTOOLONG	Object path name is too long. The maximum length of the object path name is OBJPATH_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_NBAPPL_REGERROR	Resolver could not register the new application. Either the application name already exists or the Resolver lacks resources to register it.
NBERROR_NBAPPL_UNREGERROR	Resolver cannot unregister this application.
NBERROR_NBAPPL_UPCALLHANDLERERROR	Upcall handler thread is exiting due to operation error.
NBERROR_NBOBJ_INVOBJTYPE	System is trying to create an invalid object.
NBERROR_NBOBJ_NAMETOOLONG	Object name is too long. The maximum length of the object name is OBJNAME_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_NBOBJ_NOTSUPPORTED	Method is not supported for this type of object.
NBERROR_NBOBJ_NULLNAME	Pointer used to specify the object name is NULL.
NBERROR_NBOBJ_OUTOFMEMORY	System cannot allocate memory to create this object.

Return Codes	Description
NBERROR_NBOBJ_TITLETOOLONG	Title specified for the object is too long. The maximum length for a title is OBJTITLE_MAXLEN, which is defined in NBapi\nbparam.h.
NBERROR_TARGETMGR_CANNOTREGISTER	Cannot register this target manager with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_TARGETMGR_CANNOTUNREGISTER	Resolver cannot unregister this target manager.
NBERROR_TARGETMGR_NULLACEGROUP	The argAceGroup argument is NULL.
NBERROR_TARGETMGR_NULLACEMGR	The argAceMgr argument is NULL.
NBERROR_TARGETMGR_NULLAPPL	The argAppl argument is NULL.
NBERROR_UPCALL_CANNOTREGISTER	Cannot register this upcall handler with the Resolver for one of these reasons: <ul style="list-style-type: none"> ■ The Resolver is not running. ■ The Policy Accelerator driver is not running. ■ There is a problem communicating with the Policy Accelerator. ■ The name has already been registered.
NBERROR_UPCALL_CANNOTUNREGISTER	Resolver cannot unregister this upcall handler.
NBERROR_UPCALL_NULLACEGROUP	The argAceGroup argument is NULL.
NBERROR_UPCALL_NULLACEMGR	The argAceMgr argument is NULL.
NBERROR_UPCALL_NULLAPPL	The argAppl argument is NULL.
NBERROR_UPCTHREAD_CANNOTCREATEEVENT	Cannot create synchronization object used by upcall handler thread.
NBERROR_UPCTHREAD_ERRDEVUPCALL	Error while waiting for the driver to return upcall.
NBERROR_UPCTHREAD_ERRIDTOACE	Error converting ACE ID into ACE reference to direct upcall.

Alphabetical Listing

Return Codes	Description
NBERROR_UPCTHREAD_ERRIDTOUPCALL	Error converting upcall ID into upcall reference to issue upcall.
NBERROR_UPCTHREAD_OUTOFMEMORY	Out of memory.

Appendix B

IX-API SDK File Types



File Types and Extensions

This appendix lists the source and object code file types used by the IX-API SDK, as identified by their filename extensions.

For more information on the compilers and linkers that use and produce these files, see Chapter 7, “Command-Line Tools.”

File Extension	Description
.a	A static library file for UNIX. In the IX-API SDK installation directory, an IX-API SDK system file.
.cpp	A C++ source code file. The source file for the host module or action code part of the accelerator module of a network policy application. <ul style="list-style-type: none">■ On Windows NT, you must use Microsoft Visual C++ to produce and compile C++ source files.■ On UNIX, you can use any editing tool (such as <code>emacs</code> or <code>vi</code>) and save C++ source files with other extensions (such as <code>.C</code>, <code>.c++</code>, or <code>.cxx</code>) that are accepted by your ANSI C++ compiler.
.dll	A shared library file for Windows NT. In the IX-API SDK installation directory, a IX-API SDK system file.
.exe	An executable file for Windows NT. In the IX-API SDK installation directory, a IX-API SDK utility application or tool.
.h	A C++ header file to be included in a C++ host module or action code file, or an NCL header file to be included in a classification code file. Generally contains definitions to be used by multiple files.

File Extension	Description
.nbo	<p>A compiled action code file, written using C++ and the IX-API SDK action services library (ASL), and compiled with the ASL compiler (nbgcc).</p> <p>Together with a .ncl file, makes up the accelerator module for an ACE.</p>
.ncl	<p>A classification code source file, written in the IX-API SDK classification language (NCL). Compiled at run time in the Policy Accelerator.</p> <p>Together with a .nbo file, makes up the accelerator module for an ACE.</p>
.o	<p>A compiled C++ file. The compiled host module file for a network policy application.</p>
.so	<p>A shared library file for UNIX. In the IX-API SDK installation directory, an IX-API SDK system file.</p>

Appendix C

Policy Accelerator Name Space



This appendix explains how the Resolver manages relations between objects and entities on the host and on the Policy Accelerator using naming and reference conventions. It includes the following topics:

- Overview
- Object Name Syntax
- System Names for Policy Accelerator Interfaces
- Example

Overview

An IX application contains logical entities that comprise objects on both the host and the Policy Accelerator. ACEs, targets, and crosscalls, for example, each have a manager object on the host, and a managed object on the Policy Accelerator. The Resolver manages the relationships between these paired objects using the dictionary names—that is, the value of the name argument that you give the objects on creation. The dictionary name is the same for both objects in a pair.

Dictionary names are unique only in the context of the containing entity—a target, for example, within an ACE, an ACE within an ACE group, and an ACE group within an application. You must use a *full name* to uniquely identify an object. The full name consists of the object's individual dictionary name along with a path that identifies the objects that contain it. Full object names are derived from the hierarchy of named objects as shown in the following figure:

Object classes

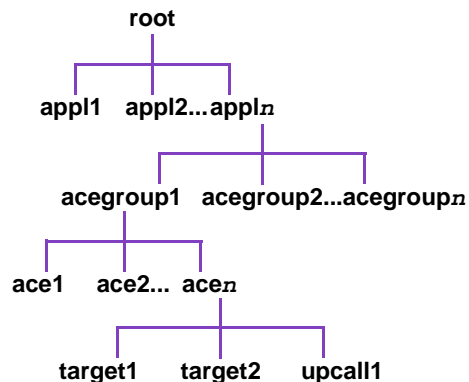
Example object names

NBAppI

AceGroup

Ace/AceManager

Target, Upcall, Downcall, Crosscall



Names must be unique at each level within the object above it. For example, you cannot have an upcall and a target using the same name within the same ACE block.

In most cases you can use an individual object's name, because the application or ACE context is clear. You must specify an object's full name to provide an explicit context in the following cases:

- To bind targets (using the application object's `bind` method)
- To link crosscalls and their handlers (using the application object's `link` method)
- To retrieve an ACE identifier for use with the `nbgdb` debugger (using the `getaceid` command)

Object Name Syntax

The full name of an object has the following possible formats:

```

/appname/acegroupname/acename
/appname/acegroupname/acename/objectname

```

The names come from the following definitions:

<i>appname</i>	<p>The application's dictionary name, which you specified when creating your subclass of <code>NBApp1</code>. For example, the name <code>Firewall</code> in the following code:</p> <pre>NBMyApp1::NBMyApp1 (void): NBApp1 ("Firewall", "myapp.exe")</pre> <p>This name must be unique in a host system. For example, attempting to run a second application with the dictionary name <code>Firewall</code> would fail.</p> <p>Use the Policy Accelerator's name for <i>appname</i> to refer to the Policy Accelerator's interfaces, as described in "System Names for Policy Accelerator Interfaces" on page 444.</p>
<i>acegroupname</i>	<p>The dictionary name you specified when creating your subclass of <code>AceGroup</code>; for example, the name <code>MyGroupName</code> in the following code:</p> <pre>myGroup = new NBMyGroup (appl, this, "MyGroupName")</pre> <p>This name must be unique within an instance of <code>NBApp1</code>.</p> <p>The IX-API SDK provides system-defined ACE group names that refer to the Policy Accelerator's interfaces, as described in "System Names for Policy Accelerator Interfaces" on page 444.</p>
<i>acename</i>	<p>The dictionary name for your ACE block (consisting of an ACE manager and an ACE). Use the name you specified when creating your subclass of <code>AceManager</code> or of <code>Ace</code>. It is the same for both the ACE and the ACE manager, and it is the means by which they are associated; for example, the name <code>MyAceName</code> in the following code:</p> <pre>myMgr = new NBMyMgr (appl, this, "MyAceName");</pre> <p>This name must be unique within an ACE group.</p> <p>The IX-API SDK provides system-defined ACE names that refer to the Policy Accelerator's interfaces, as described in "System Names for Policy Accelerator Interfaces" on page 444.</p>
<i>objectname</i>	<p>The dictionary name you specified when creating your subclass of an object within the ACE block; for example, the <code>Target</code> subclass with the name <code>MyTgtName</code> in the following code:</p> <pre>target1 = new NBMyTgt (id, ace, "MyTgtName");</pre> <p>This name must be unique within an ACE block.</p> <p>Every ACE, by default, also comes with two predefined target names: <code>pass</code> and <code>drop</code>.</p>

For the examples shown in this table, the following are among the list of valid names:

```
/Firewall/MyGroupName/MyAceName
/Firewall/MyGroupName/MyAceName/MyTgtName
/Firewall/MyGroupName/MyAceName/pass
```

System Names for Policy Accelerator Interfaces

For packets to flow through the Policy Accelerator, you must bind ACEs to targets. To allow you to bind the hardware interfaces on each Policy Accelerator, the IX-API SDK provides system-defined ACEs representing the interfaces. These ACEs contain system-defined targets for use in binding. This section describes the names of these objects.

System ACE Names

The system-defined ACEs are named using the following formats:

```
/PAname/sysacegroupname/sysacename
/PAname/sysacegroupname/sysacename/objectname
```

<i>PAname</i>	The Policy Accelerator name as described in “Policy Accelerator Names” on page 445.	
<i>sysacegroupname</i> <i>/sysacename</i>	These specify whether to use the interface as a network interface or as a stack interface. They take one of these formats: From <i>type</i> : <i>PNameInterface/type</i> To <i>type</i> : <i>PNameInterface/type</i>	
	From To	Specifies how the interface is used.
	<i>type</i>	Interface (for the network) or stack (for the host stack).
	<i>PAname</i>	Same as the preceding <i>PAname</i> ..
	<i>Interface</i>	A or B , as described in “Policy Accelerator Interface Names” on page 445.
<i>objectname</i>	Only the default <i>pass</i> target is a valid object name. Every ACE, by default, comes with two predefined target names: <i>pass</i> and <i>drop</i> .	

For example:

- From packet interface on the first installed Policy Accelerator:

```
/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass
```

- To packet interface on the second installed Policy Accelerator:

```
/nbhwpe1/ToInterface:nbhwpe1B/Interface
```

- From host stack on the first installed Policy Accelerator:

```
/nbhwpe0/FromStack:nbhwpe0B/Stack/pass
```

- To host stack on the third installed Policy Accelerator:

```
/nbhwpe2/ToStack:nbhwpe2A/Stack
```

Policy Accelerator Names

You can install more than one Policy Accelerator in a single host.

Each installed Policy Accelerator has a default name that is used in constructing the default system names. The Policy Accelerator names have the following format:

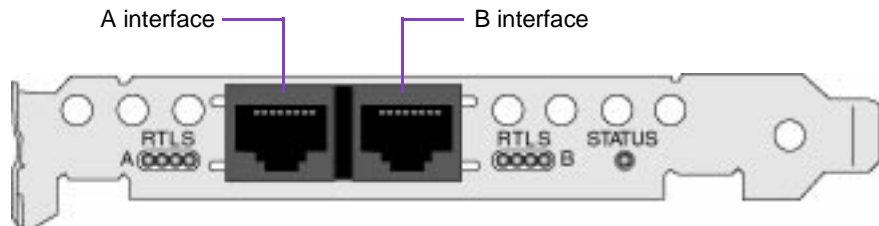
nbhwpe*number*

where *number* indicates which Policy Accelerator. The first Policy Accelerator installed in a system is number 0, the next is 1, and so on; for example, nbhwpe0 and nbhwpe1.

You can use these names in the `AceManager` class constructor to specify on which Policy Accelerator you want the ACE to be executed.

Policy Accelerator Interface Names

The two interfaces on a Policy Accelerator are named A and B, as shown in the following diagram for the PCI-format Policy Accelerator:

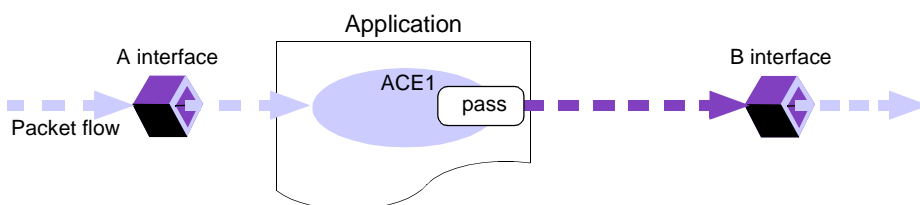


Each interface can be used as a packet interface or as a connection to the host protocol stack, or both. For more information on ways to connect the interfaces for different purposes, see “Physical Packet Flow” on page 38 of *Developing Applications Using the IX-API SDK*.

If your site has customized the drivers for a standard network interface card (NIC) for communication with the Policy Accelerator using the ODX protocol, you can address the NIC connection directly as interface C. The syntax for using interface C is the same as that for the built-in interfaces A and B. For more information, see *Customizing a NIC Driver Using the ODX Protocol*.

Example

For example, in the following diagram, the ACE named ACE1 is designed to take input from network interface A, then send packets through its own pass target to network interface B:



The code to bind this packet flow is:

```
DWORD rval;
rval = bind
    ("/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass",
     "/MyAppl/MyAceGroup/ACE1");
if (rval != NB_SUCCESS) {
    NB_ABORT(rval);
}
rval = bind ("/MyAppl/MyAceGroup/ACE1/pass",
             "/nbhwpe0/ToInterface:nbhwpe0B/Interface");
if (rval != NB_SUCCESS) {
    NB_ABORT(rval);
}
```



Glossary

A

accelerator module	The portion of an application that runs on the <i>Policy Accelerator</i> . Consists of two parts: C++ code that uses the <i>Action Services Library (ASL)</i> , and <i>Network Classification Language (NCL)</i> code.
ACE	Action/Classification Engine. An ACE contains a set of packet classification <i>rules</i> and associated <i>actions</i> , <i>upcall</i> and <i>downcall</i> entry points, and <i>targets</i> . Applications use ACEs to process <i>packet buffers</i> .
ACE block	A collective term of an <i>ACE manager</i> object in the <i>host module</i> and its corresponding <i>ACE</i> object in the <i>accelerator module</i> .
ACE group	A container object in the <i>host module</i> that groups one or more <i>ACE blocks</i> .
ACE manager	A manager object in the <i>host module</i> that manages and controls an <i>ACE</i> in the <i>accelerator module</i> .
action code	A part of the <i>accelerator module</i> written using the <i>Action Services Library (ASL)</i> . This part of the module defines and creates the objects that reside on the <i>Policy Accelerator</i> , as well as defining the <i>actions</i> for an ACE. The run-time environment for action code includes a C and C++ run-time environment with restricted standard libraries appropriate to the <i>Policy Accelerator</i> .
Action Services Library (ASL)	A set of C++ library functions and methods supplied with the IX-API SDK. It consists of classes, methods, functions, and macros that you can use when writing C++ code to implement <i>actions</i> that take place after packets have been classified by <i>rules</i> .
action	A function entry point in the <i>accelerator module</i> implemented according to the calling conventions of the C++ programming language. An action function performs an operation on a <i>packet buffer</i> that is sent to it as the result of the <i>classification</i> performed by a <i>rule</i> .

B

big endian	A compiler term specifying that, for multibyte values, the most significant byte is sent first. See also <i>little endian</i> .
bind	A method of the <i>NBAppl</i> class that associates a <i>target</i> with an <i>ACE</i> . The bindings of the <i>ACEs</i> determine how a <i>packet buffer</i> flows through the <i>Policy Accelerator</i> .
buffer	See <i>packet buffer</i> .
byte order	The way a system stores numeric data, with the most or least significant byte first. See also <i>endianness</i> , <i>network byte order</i> , <i>marshalling</i> .

C

callback	An application-defined function that handles the results of an asynchronous operation, such as passing a <i>message</i> between the <i>host</i> and <i>Policy Accelerator</i> . You specify a callback function when initiating the operation, and the function is called when the operation is completed.
classification	The process by which <i>rules</i> you write using <i>Network Classification Language (NCL)</i> evaluate the content of packets.
crosscall	An object in the <i>accelerator module</i> that enables an <i>ACE</i> to send a <i>message</i> to another <i>ACE</i> . The other <i>ACE</i> must contain a <i>crosscall handler</i> to receive or handle this call.
crosscall handler	An object in the <i>accelerator module</i> that receives and handles a <i>crosscall</i> message sent from another <i>ACE</i> .
crosscall handler manager	A manager object in the <i>host module</i> that manages and controls a <i>crosscall handler</i> object.
crosscall manager	A manager object in the <i>host module</i> that manages and controls a <i>crosscall</i> object.
cryptographic card	A daughter card for the <i>Policy Accelerator</i> that allows you to perform cryptographic operations, such as encryption and authentication, in your network application.
cryptographic extensions	Additional classes for the <i>Action Services Library (ASL)</i> that allow you to perform cryptographic operations using the <i>cryptographic card</i> .

D

datagram	An IP packet. Datagrams contain useful information (the <i>payload</i>) combined with network control information and an IP header. Network routers use the IP header to direct the packet to the proper network node.
data set	See <i>set</i> .
dictionary name	The value you pass as the <i>name</i> argument when creating an <i>object pair</i> .
downcall	An object in the <i>host module</i> that enables it to send a <i>message</i> to an ACE in the <i>accelerator module</i> .
downcall handler	An object in the <i>accelerator module</i> that receives and handles a <i>message</i> sent from the <i>host</i> .

E

element	A member of an application-defined data <i>set</i> . You define the data to be included in elements as part of the <i>action code</i> . You create and delete elements in an <i>action</i> .
endianness	A compiler term for the byte order of multibyte values. See <i>big endian</i> and <i>little endian</i> .

F

fragment	An <i>IP4 datagram</i> that represents a portion of a higher layer's packet that was too large to be sent in its entirety over the output network. See also <i>reassembly</i> .
-----------------	---

G

global function	A function defined in the <i>Action Services Library (ASL)</i> that you call independently of the C++ objects in the <i>accelerator module</i> . Global functions include an initialization function, memory management functions, and predefined action functions.
------------------------	---

H

header file	A system file that contains definitions for an application. The <i>host API</i> and <i>Action Services Library (ASL)</i> both have header files that you must include to use the defined classes.
host	The computer containing the <i>Policy Accelerator</i> . The <i>host module</i> part of a <i>policy-enforcement application</i> runs on the host in C++.

host API	A set of C++ classes that you use to write the <i>host module</i> , in which you create and configure <i>ACE managers</i> , <i>bind targets</i> , and pass <i>messages</i> to and from the <i>Policy Accelerator</i> .
host byte order	The byte order of multibyte values used by the sender of data, as opposed to the byte order used by the network. See also <i>network byte order</i> and <i>marshalling</i> .
host module	The part of an application that runs on the <i>host</i> . The host module uses the <i>host API</i> to manage the <i>accelerator module</i> part of the application, which runs on the <i>Policy Accelerator</i> .
I	
include file	See <i>header file</i> .
Intel® Internet Exchange™ architecture	A new approach to designing networking and telecommunications equipment based on reprogrammable silicon and open interfaces. Manufacturers of networking and communications equipment can use components from the IX architecture-based product portfolio for designing new, more intelligent network systems.
interface	A physical connection to the <i>Policy Accelerator</i> through which packets flow. Each interface is associated with a <i>system ACE</i> , which can be bound to a <i>target</i> .
intrinsics	Built-in functions in <i>Network Classification Language (NCL)</i> that deal with checksums in TCP/IP packets. These are routines that you cannot create with NCL. The intrinsics are included as part of the <i>IX API</i> (for example, <code>chksum</code> , <code>gencksum</code>).
IP4	Internet protocol, version 4. A standard network protocol, part of TCP/IP.
IX	<i>Intel® Internet Exchange™ architecture</i> . A set of hardware and software building blocks for networking infrastructure applications.
IX API	The application programming interface to <i>IX</i> . Includes the <i>host API</i> , <i>Action Services Library (ASL)</i> , and <i>Network Classification Language (NCL)</i> .
IX-API SDK	The software developer's kit for <i>IX</i> . Includes the <i>IX API</i> , system software, runtime libraries, and software development tools such as compilers and debuggers.

J

K

key A field to use when searching for an *element* in a data *set*. You define the number of keys when you define the set in *Network Classification Language (NCL)*, then specify which protocol field to use for each key when you define a *search* for the set.

L

link A method of the *NBAppl* class that associates a *crosscall* with the *crosscall handler* that will handle the *message*.

little endian A compiler term specifying that, for multibyte values, the least significant byte is sent first. See also *big endian*.

load A method of the *ACE manager* object that loads the *rule* and *action* for the associated *ACE* into the *Policy Accelerator*.

M

MAC Media access control. The *interface* on the *Policy Accelerator* are MAC interfaces. Each has a MAC address (in this case, an Ethernet address).

In cryptography, message authentication code. A one-way hash that is computed from a message and some secret data, which allows you to detect whether the message has been altered.

marshalling The process of converting the numeric arguments of a *message* to a known *byte order* before transporting them over the network. See also *unmarshalling*.

message An encapsulation of data that you can send from the *host module* to the *accelerator module* or vice versa, or pass between *ACEs*.

module A portion of application code destined to execute in a particular hardware location. An IX application requires two modules: the *host module* and the *accelerator module*.

N

NBAppl class	The main <i>host API</i> class that represents the <i>IX API</i> portion of an application. This class provides management services (such as <i>bind</i> and <i>link</i>) to set up the relationships of <i>ACE</i> classes.
network address translation (NAT)	The ability to modify various fields of different protocols so that the effective source, destination, or source and destination entities are replaced by an alternative.
network byte order	The byte order of multibyte values used by the network, as opposed to the byte order used by the host (in this context, the sender of data). See also <i>host byte order</i> and <i>marshalling</i> .
Network Classification Language (NCL)	A declarative language consisting of Boolean operators, packet header field descriptors, constants, set-membership queries, and other operations, which you use to create <i>rules</i> for classifying packets.
NIC	Network interface card. A hardware device that connects a computer with the network. Packets flow through a NIC to and from the host and the network. The <i>Policy Accelerator</i> can be used as a simple NIC, although that is not its primary purpose. A computer that contains a Policy Accelerator can also contain a conventional NIC.

O

object pair	Associated objects in the <i>accelerator module</i> and <i>host module</i> of an IX application. You associate an object with its manager or handler by giving both objects the same <i>dictionary name</i> .
--------------------	---

P

packet buffer	A data buffer that the <i>Policy Accelerator</i> uses to store packet information for processing by <i>rules</i> and <i>actions</i> . After an <i>ACE</i> completes processing, it can drop the packet buffer or pass it to a <i>target</i> .
payload	The part of a packet that carries data, as opposed to those parts that carry information about the packet.
policy	A set of <i>rules</i> that determine the disposition of network packets.
Policy Accelerator	An Intel board that you install into a computer that runs <i>policy-enforcement applications</i> using the <i>IX API</i> .

policy-enforcement application	An application that uses the <i>Policy Accelerator</i> and <i>IX API</i> to describe and implement <i>policy</i> regarding the manipulation or disposition of packets in network traffic flow.
predicate	A Boolean function on protocol fields in NCL. You use predicates in rules to determine whether a packet contains certain information or structure.
protocol	The definition of the structure of a network packet. There are standard protocols, such as <i>TCP/IP</i> and <i>UDP</i> , for which the <i>IX API</i> contains built-in descriptions. You can also describe the structure of your preferred protocols in <i>Network Classification Language (NCL)</i> .

Q

R

reassemble	The process of collecting related IP <i>fragments</i> into a single <i>datagram</i> .
Resolver	<p>A process started at boot time that is responsible for managing the status of all applications that use the Intel <i>Policy Accelerator</i>.</p> <p>The Resolver includes the NCL compiler and Policy Accelerator linker/loader. The process responds to requests from applications to set up ACEs, bind targets, and perform other maintenance operations on the Intel Policy Accelerator.</p>
rule	A specification of what to do with a network packet. You write rules in <i>Network Classification Language (NCL)</i> . A rule associates a <i>predicate</i> with an <i>action</i> to classify packets and dispose of them according to the <i>classification</i> .

S

search	An application-defined method of finding <i>element</i> of data <i>set</i> . You define searches with the set definition in <i>Network Classification Language (NCL)</i> , and use the results of searches in an <i>action</i> function.
set	A data structure specified for an application, with specific named searches that you can execute in the <i>accelerator module</i> . You use sets to store and retrieve application-specific data. You declare and name sets in <i>Network Classification Language (NCL)</i> , and populate them in <i>action</i> functions.
string search	A facility in the <i>Action Services Library (ASL)</i> that allows you to find specific strings or strings matching a specific pattern in the <i>payload</i> of one or more <i>packet buffers</i> .

system ACE A system-defined *ACE* with a reserved name. System ACEs represent the network *interfaces*. You *bind* a system ACE to an application-defined ACE's *target* to send or receive packets from the corresponding network interface.

T

target A part of an *ACE* representing the source or destination of a *packet buffer*. You *bind* targets to other application-defined ACEs or *system ACE* representing network interfaces.

TCP Transmission control protocol. A standard network protocol in which transmission status can be confirmed. Part of *TCP/IP*.

TCP/IP A standard network protocol, using *TCP* over *IP4*.

TCP/IP extensions Additional classes for the *Action Services Library (ASL)* that allow you to access *TCP/IP* and *UDP* packets and perform common operations such as *network address translation (NAT)* on them.

U

UDP User datagram protocol. A standard network protocol in which transmission status cannot be confirmed. A peer of *TCP*, used over *IP4*.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. It is transaction oriented; delivery and duplicate protection are not guaranteed. Applications requiring ordered, reliable delivery of data streams should use *TCP*.

unmarshalling The process of converting the numeric arguments of a *message* from *network byte order* to *host byte order* after transporting them over the network. See also *marshalling*.

upcall An object in the *accelerator module* that enables it to send a *message* to the *host module*.

upcall handler An object in the *host module* that receives and handles a *message* sent from the *accelerator module*.

V

W

X

Y

Z

Index



A

- .a files 439
- accelerator module 3
 - associated objects in host module 27
 - communicating with host 99
 - creating message in 180
 - data types 9
 - files 439
 - initializing 45
- ACE 4
 - full name 441
 - see also* ACEs
 - system-defined, names of 444
- ACE blocks 4, 33
- Ace class 110
 - constructor 113
- ACE managers 4
 - associating with ASL ACEs 27, 45
 - communicating with ACEs 5
 - grouping 33
 - see also* AceManager class
- ace method 156
- AceGroup class 33
 - constructor 35
- AceManager class 37
 - constructor 40
 - see also* ACE managers
- ACEs 4
 - accelerator module classes 97, 110
 - assigning Policy Accelerator for 40
 - associating ACE and ACE manager objects 27, 45
 - blocks 33
 - communicating with ACE managers 5
 - connecting 265
 - creating object in init function 7, 46, 172
 - finding for object 156
 - groups 33
 - initializing 45, 97, 172
 - initializing string searches in 98
 - management classes 28
 - management objects 33, 37
 - packet modification by 40
- ack method 351
- action code 2
 - loading 45
 - see also* ASL
- action functions 97, 115
 - allowed return values 117
 - associating with rules 118
 - calling from rules 406
 - debugging 422
 - defining 117
 - passing arguments to 408
- Action Services Library, *see* ASL
- Action/Classification Engines, *see* ACEs
- action_drop function 116
- action_pass function 116
- ActiveStrings method 225
- add method (Rate) 250
- add method (ReassemblyQueue) 340
- address translation, *see* network address translation
- addresses, IP4 308
- AddString method 236
- apasum method 303
- API 2
 - classes in ASL 107
 - classes in ASL extension 300

- classes in host API 31
 - functions in ASL 107
 - see also* host API, ASL, NCL
- append method 126
- application management classes 28
 - ACE managers 37
 - groups 33
 - main class 68
 - targets 85
- application programming interface, *see* host API, ASL, NCL
- applications 3
 - creating 3
 - for networks with multibyte values 10
 - main class 68
 - modules in 3
- apsasum method 303
- apssum method 304
- apsum method 304
- architecture of applications 3
- ASL 2
 - classes and functions in 107
 - extension classes for TCP/IP 279
 - files 439
 - global functions 107
 - header files to include 96, 107
 - overview 95
 - part of SDK API 2
- ASL objects 105
 - finding owning ACE 156
- assembling TCP fragments 287, 339
- associating objects with their managers 27
- asum method 305
- asynchronous processing, *see* communication, string searches

B

- Backlog class 119
- base classes 83, 105
- bcast method 309
- big endian byte order 10
- bind method (NBAppI) 72
- bits method 332
- Boolean expressions 392
 - and named predicates 402
- buf method 320
- Buffer class 123
 - new operator 131

- buffer memory, limiting 104
- buffers 97, 123
 - dropping 113, 116
 - passing 114, 116
 - searching for strings in 98
 - see also* packets
- busy method 126
- byte order 10
 - and message passing 13
 - and operators 14
 - and search keys 102
 - changing 12
 - classes 10, 15
 - in messages 13
 - little endian and big endian 10
 - marshalling arguments 13
 - Policy Accelerator 12

C

- C interface 73, 446
 - testing 424
- C language, data type definitions for 10
- call method (Crosscall) 143
- call method (Downcall) 61
- call method (Uppcall) 276
- callbacks 6
 - connection state 206
 - crosscall 148
 - downcall 154
 - event 167
 - expiration 164
 - see also* communication, string searches
 - string search 222, 228, 230, 232, 233, 237, 239, 241, 242
 - upcall 92
- cancel method (Elt_setname) 163
- cancel method (Event) 169
- cecomp command, syntax 414
- cecomp.exe, *see* NCL compiler
- celink command, syntax 417
- ChangeOpMode method 239
- checkcksum method (IP4Datagram) 312
- checkLinks method 207
- checksums 302
 - validating and generating in NCL 397
- cksum method (Internet) 305
- cksum method (IP4Header) 325
- cksum method (TCPHeader) 351

- cksum method (UDPHeader) 376
- classes 26, 95
 - ACE (Accelerator side) 97
 - ACE management 28
 - ASL 95, 107
 - base (host side) 30
 - base accelerator side) 105
 - byte order 11, 15
 - data sets and searches 101
 - error handling 30
 - host API 26, 31
 - interface management 105
 - memory management 103
 - message passing (accelerator side) 99
 - message passing (host side) 29
 - network address translation 281
 - packet manipulation 97
 - statistical 101
 - TCP/IP 279, 300
 - time and event scheduling 100
- classification rules, *see* rules
- classification with sets 101
- clear method (Rate) 250
- clear method (ReassemblyQueue) 341
- client method 370
- code 439
 - accelerator module, action 95
 - file types 439
 - host module 25
- codeaccelerator module, classification 387
- commands 413
- comments in NCL 390
- communication 5
 - between host and Policy Accelerator 29, 99
 - byte order during 12
 - message classes in API 29
 - message classes in ASL 99
 - see also* message passing
 - with a NIC 73, 446
- compiled files 439
- compiler syntax 413
 - action code 419
 - ASL file linker 423
 - NCL compiler 414
- complete method (IP4Datagram) 312
- complete method (IP4Fragment) 320
- configuration files, memory management directive 104
- configuration information 76

- connections 197
 - monitoring 205
 - properties of 197
- contacting Intel xxii
- conventions, typographical xxi
- converting byte order, *see* byte order
- count method 250
- .cpp files 439
- Crosscall class 139
 - constructor 142
- CrosscallHandler class 144
 - constructor 147
- CrosscallHandlerManager class 48
 - constructor 50
- CrosscallManager class 52
 - constructor 54
- crosscalls 29, 99
 - associating sender and receiver 77
 - dissociating sender and receiver 80
 - managing 48, 52
 - receiving 144
 - responding to 148
 - sending 139
- curr method (Event) 169
- curr method (Time) 270
- customer support xxii

D

- data method 362
- data sets 101
 - ASL classes 103
 - see also* sets
- data storage consistency 10
- data types 10
 - and search keys 102
 - including in code 9
- datagrams, IP4 310
 - example 287
- datalen method (IP4Fragment) 321
- datalen method (IP4Header) 325
- debugging tools 413
 - syntax 421
- decref method 127
- demultiplexing 399
- demux statement 399
- destinations for packets 265
- diagnostic tests 424, 425
- dictionary names of objects 27

- direct method (CrosscallHandler) 148
- direct method (Downcall) 154
- direct method (Event) 170
- displaying output from PolicyAccelerator 426
- distributed objects 27
- .dll files 439
- DOS commands 413
- Downcall class 56
 - constructor 59
- DowncallHandler class 150
 - constructor 153
- downcalls 29, 99
 - and byte order 13
 - and handlers 27
 - receiving 150
 - responding to 154
 - sending 56
- dport method (TCPHeader) 351
- dport method (UDPHeader) 377
- drop method 113
- drop target 5
- dst method 325
- dual objects 27
- Dualobj class 155
 - constructor 156
- duplex interface property 197
- durations 268
- Dynamic class 157
 - delete operator 158
 - new operator 158

E

- elapsed times 268
- Element class 159
- Elt_setname class 160
 - constructor 162
 - delete operator 163
 - destructor 162
 - new operator 165
- empty method 341
- end method 218
- endianness, *see* byte order
- endpoints, TCP 346
- endseq method 363
- error codes 429
- error handling 30, 81
- error messages, compiler
 - freeing local memory 47

- retrieving 43
- est method 120
- Event class 166
 - constructor 168
 - destructor 169
- events, scheduling 100
- .exe files 439
- expire method 164
- extensions, file name 439

F

- file system access from applications 96
- file types 439
- find method (Name) 192
- find method (Named) 196
- first method (IP4Fragment) 321
- first method (Set) 260
- flags method (TCPHeader) 351
- flags method (TCPSegInfo) 363
- floating-point math in applications 96
- fragment method (IP4Datagram) 313
- fragment method (IP4Fragment) 321
- fragmented method 313
- fragments 318
 - assembling 287, 339
- free method (Pool) 247
- free method (Tagged) 263
- freeing local message memory 47
- full object names 441
- functions
 - callback, *see* callbacks
 - defining action functions 117
 - in ASL 107
 - initialization for ACEs 172
 - intrinsic in NCL 397
 - memory monitoring 104, 178
 - predefined action functions 115

G

- generating header files from NCL 411
- getaceid command, syntax 418
- getBuffer1 method 64
- getBuffer2 method 64
- getCompilerErrorMessages method 43
- getDropTarget method 44
- getErrorcode method 82
- getId method 83
- getLen1 method 64

- getLen2 method 65
- getmemstatvalues function 178
- getPassTarget method 44
- GetProperty method 200
- GetPropertyList method 201
- GetQueryRate method 215
- GetRmonCounters method 213
- GetRXTXStats method 214
- getStackDriverName method (NBAppI) 76
- getTag method (AceManager) 45
- getTag method (NBAppI) 74
- getType method 83
- getUpcallFunction method 93
- global functions in ASL 107
- glossary 447

H

- .h files 439
- handlers, *see* communication
- hardware resources, specifying 40
- hardware tests 424, 425
- hdr method 320
- head method 314
- header files 9
 - for ASL classes 96, 107
 - for host API classes 25
 - including in NCL 389
 - using to synchronize NCL with ASL 411
- headerBase method 128
- headers, packet 283
 - IP4 324
 - TCP 350
 - UDP 376
- headerType method 128
- here method 193
- hierarchy of named objects 441
- hit method 253
- hl method 326
- hlen method (IP4Header) 326
- hlen method (TCPHeader) 352
- host 1
 - resource manager 2
- host API 25
 - classes 31
 - header files for inclusion 25
 - part of SDK API 2
- host module 3
 - associated objects in accelerator module 27
 - communicating with accelerator module 29
 - creating messages in 62
 - data types 9
 - files 439
 - managing ACEs 28
- htohs macro 12
- htonl macro 12
- htonl method 22
 - example 13
 - using 12
- htons method 18
 - using 12

I

- id method 326
- image method 366
- include files 9
 - for ASL classes 96, 107
 - for host API classes 25
 - in NCL 389
- incrcksum method 306
- incrcf method 129
- init method (NBRmon) 212
- init method (TCPEndpoint) 347
- init_actions function 172
- initializing ACEs 97, 172
- insert method (IP4Datagram) 314
- insert method (Search) 253
- integer types 10
 - byte order sometimes not important 14
 - converting byte order 11
 - network byte order 16
- Intel, contacting xxii
- interaces
 - built into Policy Accelerator 445
- interface configuration 76
- interface properties 105, 197
 - duplex 197
 - MAC address 197
 - speed 197
- interfaceNum method 129
- interfaces
 - C interface for NIC connection 73, 446
 - system ACEs for 444
- interfaceType method 131
- Internet class 302
- intervals, specifying 268
- intrinsic functions in NCL 397

IP addresses, finding for Policy Accelerator 76

IP4 324

- constant definitions 293
- datagrams 310
- datagrams, example 287
- fragments 318
- fragments, assembling 339
- headers 324
- masks 331
- network address translation 333

IP4Addr class 308

- constructor 308

IP4Datagram class 310

- constructor 311
- destructor 311

IP4DNat class 316

- constructor 316

IP4Fragment class 318

- constructor 319
- destructor 320

IP4Header class 324

IP4Mask class 331

- constructor 331

IP4Nat class 333

IP4SDNat class 335

- constructor 335

IP4SNat class 337

- constructor 337

K

key point, explanation of xxi

keys for sets and searches 102

- specifying number 403

keywords in NCL 392

L

leftcontig method 332

len method (IP4Datagram) 315

len method (IP4Header) 327

len method (NBStringMatchReport) 218

len method (TCPSegInfo) 363

len method (UDPHeader) 377

library files 439

link method (Linked) 175

link method (NBAppI) 77

Linked class 174

- constructor 175
- destructor 175

linking 423

- compiled ASL files 423
- crosscall senders and receivers 77
- objects in a ring 174

little endian byte order 10

load method 45

locate method 260

M

MAC address interface property 197

MAC interfaces 205

- RMON information 208

macros, byte swapping 12

main application class 68

managers and managed objects 27

marshalling arguments to control byte order 13

masks, IP4 331

matches method 219

maxbuf configuration directive 104

mcast method 309

members of data sets 160

- creating 165, 253
- expiration 163, 164
- removing 254

memory management 47, 103, 104

- allocating packet buffer memory 104
- allocation in ASL 105
- for messages (ASL) 180
- for messages (host API) 62
- freeing local error message memory 47
- global functions 178
- monitoring packet buffer memory 119, 135
- usage statistics 104

Message class (ASL) 180

- access methods 183
- acknowledgement methods 183
- constructor 182
- example 13

Message class (host API) 62

- constructor 63

message passing 29

- classes in API 29
- classes in ASL 99
- crosscalls 48, 52, 139, 144
- downcalls 56, 150
- upcalls 88, 273

MessageBlock class (ASL) 184

- constructor 186

- MessageBlock class (host API) 66
 - constructor 67
 - messages 29
 - and byte order 13
 - classes in API 29
 - classes in ASL 99
 - constructing in ASL 180, 184
 - constructing in host API 62, 66
 - freeing local host memory 47
 - maximum size (ASL) 180
 - maximum size (host API) 66
 - packing and unpacking to control byte order 13
 - MIB creation 208
 - miss method 254
 - modifying packets with ACEs, allowing 40
 - modules in an application, *see* host module, accelerator module
 - monitoring 208
 - connection state 205
 - memory usage 104
 - network connection 105
 - packet flow 119, 248
 - mstats function 179
 - multibyte values in network applications, *see* byte order
 - multiple operations on packet buffers 126, 127, 129
 - multiple Policy Accelerators 40
 - multiplexed protocols 399
 - multithreading in applications 96
- N**
- Name class 191
 - constructor 192
 - name method 196
 - Named class 194
 - constructor 195
 - destructor 195
 - names 441
 - full names of objects 441
 - uniqueness 442
 - names and paired objects 27
 - names method 120
 - namespace for managed objects 27, 106
 - NAT, *see* network address translation
 - NBAppl class 68
 - constructor 70
 - NBError class 81
 - NBFIF_GET_SET_PROP_ITEM structure 201
 - NBFIF_PROP_CAP structure 202
 - NBFIF_PROP_CAP_ITEM structure 202
 - NBFIF_PROP_ITEM structure 203
 - nbgcc command, syntax 419
 - nbgdb command, syntax 421
 - NBInterfaceProp class 197
 - constructor 200, 206
 - nbld command, syntax 423
 - NBLinkwatch class 205
 - .nbo files 440
 - NBObject class 83
 - NBRmon class 208
 - constructor 211
 - destructor 212
 - NBSearchContext class 222
 - constructor 225
 - NBStringMatchReport class 216
 - constructor 217
 - NBStringSearchEngine class 233
 - constructor 235
 - NCL 2, 387
 - comments in 390
 - defining protocols in 395
 - defining sets in 403
 - file structure 388
 - files 439
 - including header files 389
 - intrinsic functions 397
 - keyword overview 392
 - loading code 45
 - named predicates 402
 - operators 392
 - part of SDK API 2
 - rules 406
 - runtime compilation of 387
 - symbolic constants in 389
 - NCL compiler 414
 - retrieving error messages 43
 - using to generate action header files 411
 - .ncl files 440
 - nested protocols 399
 - network address translation 281
 - example 284
 - IP4 316, 333, 335, 337
 - TCP 343, 355, 359, 371
 - UDP 374, 378, 381, 384
 - network applications 3
 - and multibyte values 10
 - and numbers 10

- network byte order 10
 - on host and Policy Accelerator 12
- Network Classification Language, *see* NCL
- network connection, monitoring 105
- next method (Buffer) 132
- next method (IP4Fragment) 322
- next method (Linked) 176
- next method (Set) 261
- next method (TCPSegInfo) 363
- nfrags method 315
- NIC driver
 - testing 424
- note, explanation of xxi
- now method 121
- ntohl macro 12
- ntohl method 22
 - example 13
 - using 12
- ntohs macro 12
- ntohs method 18
 - using 12
- nuint16 class 16
 - constructor 17
 - converting byte order 11
- nuint32 class 20
 - constructor 21
 - converting byte order 11
- numbers in network applications, *see* data types

O

- .o files 440
- objects 27
 - ACE, creating in init function 7, 46, 172
 - associating with manager objects 27
 - calls and call handlers 27
 - creating in accelerator module 7
 - creating in host module 6
 - full names of 441
 - hierarchy of named objects 441
 - object name syntax 442–444
 - packet buffer 123
 - uniqueness of names 442
- ODX protocol 73, 446
- odxloop command, syntax 424
- off method 352
- offset method 327
- operating statistics 100
 - memory usage 104

- RMON 101
- operators 392
 - and byte order 14
 - in NCL 392
- OpMode method 240
- optbase method (IP4Header) 327
- optbase method (TCPHeader) 352
- optcopy method 322
- orphan method 176
- output from Policy Accelerator, displaying 426

P

- pa100diag command, syntax 425
- packet buffer memory 104
 - limiting 104
- packetPadHeadSize method 133
- packetPadTailSize method 133
- packets 123
 - allowing modification by ACE 40
 - classifying with rules 406
 - classifying with sets 101
 - constructing 123
 - constructing manually 124
 - destinations for 265
 - dropping 113, 116
 - finding immediate source 129
 - finding time received 134
 - finding time transmitted 138
 - headers, accessing 283
 - IP4 280, 324
 - modifying with ACEs 40
 - operating on contents 123
 - passing 114, 116
 - searching for strings in 98
 - TCP 280, 350
 - TCP/IP, handling 283
 - traffic statistics 208, 248
 - UDP 280, 376
 - working with 97
- packetSize method 133
- paired objects 27
- pass method 114
- pass target 5
- payload method (IP4Fragment) 323
- payload method (IP4Header) 328
- payload method (TCPHeader) 353
- payload method (UDPHeader) 377
- pointers for network data 16

Policy Accelerator
 testing 425
 Policy Accelerator API, *see* ASL, NCL
 Policy Accelerators 1
 and packet classification 2
 assigning to ACEs 40
 byte order in 12
 communicating with host 5
 displaying output from 426
 downloading code into 45
 interface names 445
 names for addressing multiple 445
 see also accelerator module
 Pool class 245
 constructor 245
 destructor 246
 ports method (TCPNat) 357
 ports method (UDPNat) 380
 predicates 407
 in rules 407
 naming 402
 prepend method 134
 prev method (IP4Fragment) 323
 prev method (Linked) 176
 prev method (TCPSEgInfo) 364
 process method (TCPEndpoint) 348
 process method (TCPSession) 369
 proto method 328
 protocol fields 398
 accessing in action code 411
 accessing in NCL 398
 defining 398
 protocols 279, 395
 adding fields to 401
 adding predicates to 401
 defining in NCL 395
 extending definitions 401
 extracting nested 399
 IP4 280, 324
 TCP 280, 368
 UDP 280, 376
 psum method (Internet) 306
 psum method (IP4Header) 328

R

ran method 254
 Rate class 248
 constructor 249

raw_ member (nuint16) 19
 raw_ member (nuint32) 23
 read method 341
 readport command, syntax 426
 ReassemblyQueue class 339
 constructor 340
 reference counting for packet buffers 126, 127, 129
 reference, explanation of xxi
 register data, byte order 12
 regular expressions for string searches 236
 releaseCompilerErrorMessage method 47
 releaseMessage method 47
 RemoveString method 241
 reports method 219
 reset method 348
 Resolver 2
 starting 427
 resolver command, syntax 427
 resource manager 2
 starting 427
 rewrite method (IP4DNat) 317
 rewrite method (IP4NAT) 334
 rewrite method (IP4SDNat) 336
 rewrite method (IP4SNat) 338
 rewrite method (TCPDNat) 344
 rewrite method (TCPNat) 356
 rewrite method (TCPSDNat) 361
 rewrite method (TCPSNat) 373
 rewrite method (UDPDNat) 375
 rewrite method (UDPNat) 379
 rewrite method (UDPSDNat) 382
 rewrite method (UDPSNat) 385
 RMON information 101, 208
 rules 406
 associating with action functions 118
 defining 407
 defining results of 117
 loading 45
 passing arguments to actions from 408
 rxTime method 134

S

SchedDelete method (NBStingSearchEngine) 242
 SchedDelete method (NBStringSearchContext) 226
 SchedReset method 226
 schedule method 170
 scheduling events 100, 268
 Search class 251

- SearchBuffer method 243
 - searches for strings in packets, *see* string searches
 - searches in sets 251, 403
 - defining 404
 - execution of 405
 - in action code 101
 - key values 102
 - results of (ASL) 101, 253, 254
 - results of (NCL) 405
 - see also* members of data sets
 - segment method 364
 - segments, TCP 362
 - seq method (TCPHeader) 353
 - seqs method (TCPNat) 357
 - sequences, TCP 365
 - server method 370
 - sessions, TCP 368
 - Set class 255
 - Set_setname class 256
 - constructor 259
 - SetOpt method 227
 - SetPerBufferCallback method 228
 - SetPerMatchCallback method 230
 - SetPerResetCallback method 232
 - SetProperty method 204
 - SetQueryRate method 215
 - sets 101, 256, 403
 - defining 403
 - defining members 162
 - elements 102, 159, 160
 - generating action header files for 411
 - naming 256, 403
 - populating 165, 253
 - removing elements 254
 - searching 101, 251, 260
 - see also* searches in sets
 - sid method 219
 - size method 122
 - .so files 440
 - source files 439
 - sources of packets 129
 - speed interface property 197
 - sport method (TCPHeader) 353
 - sport method (UDPHeader) 377
 - src method 329
 - stack ACEs 444
 - start method 220
 - startseq method 364
 - state method 349
 - statistics 101
 - flow rates 100, 248
 - memory usage 104
 - RMON 101, 208
 - string searches 98
 - acting on results of 222, 228, 230
 - in multiple packets 98, 222, 232
 - initializing 98
 - regular expressions in 236
 - reinitializing 232
 - reporting on results of 216
 - specifying string to search for 233, 236, 241
 - starting 243
 - support for Intel xxii
 - swapl macro 12
 - swapl method 23
 - using 12
 - swaps macro 12
 - swaps method 19
 - using 12
 - symbols xxi
 - system ACEs 444
 - system management classes 30
 - system tools 413
- T**
- tag method 221
 - Tagged class 262
 - takable method 135
 - takable_cir method 135
 - takable_max method 136
 - takable_min method 136
 - takable_set method 137
 - take method (Pool) 247
 - take method (Tagged) 264
 - take method (Target) 267
 - Target class 265
 - constructor 267
 - TargetManager class 85
 - constructor 86
 - targets 265
 - binding 72
 - identifying 44
 - retrieving tag value 74
 - system-defined 5
 - unbinding 79
 - TCP 350
 - constant definitions 295

- control bits 296
- datagrams 310
- datagrams, example 287
- fragments 318
- fragments, assembling 339
- headers 350
- network address translation 355
- options 296
- return codes 298
- session state 296
- sessions 368
- TCP/IP protocol 279
 - built-in NCL definitions 396
 - handling packets 283
- TCPDNat class 343
 - constructor 343
- TCPEndpoint class 346
 - constructor 347
 - destructor 347
- TCPHeader class 350
- TCPNat class 355
 - constructor 356
- TCPSDNat class 359
 - constructor 360
- TCPSegInfo class 362
- TCPSeq class 365
 - constructor 366
 - operators 367
- TCPSession class 368
 - constructor 368
 - destructor 369
- TCPSPNat class 371
 - constructor 371
- terminology 447
- testing a customized NIC driver 424
- testing Policy Accelerator 425
- Time class 268
 - access methods 270
 - assignment operators 272
 - builder methods 271
 - constructor 269
 - conversion operator 272
- toElement method 254
- tools 413
- tos method 329
- traffic flow statistics 208, 248
- trim_head method 137
- trim_tail method 138
- ttl method 329

- txTime method 138
- types, *see* data types
- typographical conventions xxi

U

- u_char 10
- u_int 10
- u_long 10
- u_short 10
- UDP 376
 - headers 376
 - network address translation 378
- udp method 354
- UDPDNat class 374
 - constructor 374
- UDPHeader class 376
- UDPNat class 378
 - constructor 379
- UDPSDNat class
 - constructor 381
 - description 381
- UDPSNat class 384
 - constructor 384
- unbind method (NBAppI) 79
- UNIX commands 413
 - displaying output from Policy Accelerator 426
- unlink method (Linked) 177
- unlink method (NBAppI) 80
- Uppcall class 273
 - constructor 276
- UppcallHandler class 88
 - constructor 91
- upcalls 29, 99
 - and byte order 13
 - and handlers 27
 - example 13
 - receiving 88
 - sending 273

V

- val method 367
- variables, ensuring width compatibility 10
- ver method 330
- vhl method 330

WXYZ

- warning, explanation of xxi

win method 354	Windows NTcommands 413
window method 354	displaying output from Policy Accelerator 426

IX SDK Software Developer's Kit License Agreement

IMPORTANT: You (the "licensee") are consenting to be bound by this agreement if you do any of the following:

- Click on the "accept" button
- Install or use the software
- Otherwise exercise any rights provided below to use the accompanying Intel™ IX API Software Developer's Kit (the "Intel SDK")

Or, if applicable, you are bound by a currently effective written agreement regarding the use of the Intel SDK and signed by an authorized agent of you and by an officer of Intel.

If you do not agree to the terms of this agreement or such signed agreement, as applicable, then do not use or copy the Intel SDK, and contact the place from which you obtained it, if any of these terms are considered an offer, acceptance is expressly limited to these terms.

This Agreement sets forth the terms and conditions of your use of the accompanying Intel SDK, together with documentation provided to you by Intel. Any third party software that is provided with the Intel SDK with such third party's license agreement (in either electronic or printed form), and your use of such third party software, shall be governed by such third party's license agreement in addition to this Agreement. As used in this Agreement, Intel shall mean Intel Corporation, its affiliates, or its subsidiaries.

Users of the Intel SDK pursuant to this Agreement must either be individuals using the license on their own behalf or be employees or contractors of a corporation or other entity which has accepted the terms of this Agreement and on behalf of which the Intel SDK is being used, in which case the term "Licensee" in this Agreement refers to you and such entity.

1. Grant of License.

a. Subject to the terms of this Agreement, Intel grants to Licensee a worldwide, nonexclusive, nontransferable, nonassignable, nonsublicensable license (the "License") under Intel's copyrights to (i) copy the Intel SDK and associated documentation for internal use to integrate Applications for use with Intel Products, and (ii) to make and distribute as many copies of the integrated applications containing the Intel SDK as necessary. "Intel Products" means approved Intel Hardware listed in the datasheet provided with this Intel SDK. "Applications" means Licensee's current and future expected applications that will use the Intel SDK.

b. Accompanying the Intel SDK is specific source code ("Intel Source") such as ARM Compiler, ARM Debugger, Include Files, and reference applications that Licensee may incorporate into Applications during the integration process using the Intel SDK. Subject to the terms of this Agreement, Intel grants to Licensee a worldwide, nonexclusive copyright license to reproduce, distribute, and sublicense to third parties the Intel Source in Licensee's Applications for use with Intel Products. Licensee recognizes that when it uses the Intel SDK to create or compile Applications, a portion of the Intel SDK, the Intel Source, will be compiled and linked into or with the Applications.

2. Ownership of the Intel SDK. As between the parties, Intel retains title to and ownership of, and all proprietary rights with respect to, the Intel SDK, the Intel Source, and all copies and portions thereof, whether or not incorporated into or with other Software. The License does not constitute a sale of the Intel SDK, the Intel Source, or any portion or copy of it.

3. Restrictions; Licensee Obligations.

a. Any redistribution or duplication of any software, code, and or application derived from the Intel SDK shall require that the Intel InstallShield installation program be used for installation or Licensee agrees to incorporate the Intel file license.txt in its entirety into Licensee's install program. The Intel-provided file license.txt includes all relevant copyright notices, trademark notices, and any other notices. Except as specified in the applicable user documentation provided by Intel, Licensee shall not (and shall not allow any third party to) (i) decompile, disassemble, or otherwise reverse engineer or attempt to reconstruct or discover any source code or underlying ideas or algorithms of the Intel SDK by any means whatsoever, (ii) remove any product identification, copyright or other notices, (iii) retarget any Intel SDK to interoperate with products other than Intel Products, or (iv) provide, lease, lend, use for timesharing or service bureau purposes, or otherwise use or allow others to use the Intel SDK to or for the benefit of third parties.

Confidential Information disclosed under this license agreement, including the existence and content of this Agreement, shall be considered "Confidential Information." Use and disclosure of such Confidential Information shall be governed by the terms of the Corporate Single Use Nondisclosure Agreement or other Nondisclosure Agreement, signed between the parties and incorporated into this Agreement by reference.

4. Termination of License for Cause. This agreement will remain in effect unless Intel terminates it due to a breach of its terms. Upon termination, Licensee will cease all use of the Intel SDK and promptly destroy or return to Intel all printed materials and copies of the Intel SDK and all portions thereof (whether or not modified or incorporated with or into other software) and so certify to Intel. Except for the License and except as otherwise expressly provided herein, the terms of this Agreement shall survive termination. Termination is not an exclusive remedy, and all other remedies will be available whether or not the License of the Agreement is terminated.

5. Limited Warranty and Disclaimer. **The Intel SDK is provided "as is" without warranty of any kind including, without limitation, any warranty of merchantability or fitness for a particular purpose or noninfringement. Further, Intel does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the Intel SDK or written materials in terms of correctness, accuracy, reliability, or otherwise.** Licensee understands that Intel is not responsible for and will have no liability for hardware, software, or other items or any services provided by any person or entity other than Intel.

6. Export Restrictions. Licensee agrees to fully comply with all applicable United States and EEC or other countries regulations and laws in effect now and hereinafter, including compliance with the U.S. Foreign Corrupt Practices Act and all export laws, restrictions, national security controls and regulations on the distribution or dissemination of Applications or Intel Products, technology, and information related to and/or exchanged under this Agreement. Licensee agrees not to export or reexport, or allow the export or reexport of the Intel SDK or any Intel Product, Intel Proprietary Information, or any direct product thereof in violation of any such restrictions, laws or regulations, or without all required licenses and proper authorizations, to Cuba, Libya, North Korea, Iran, Iraq, or Rwanda or to any Group D:1 or E:2 country (or national of such country) specified in the then current Supplement No. 1 to Part 740 of the U.S. Export Administration Regulations (or any successor supplement or regulations).

7. Government Contracts. The Intel SDK is provided with RESTRICTED RIGHTS. If Licensee is the Government or a Government contractor, use, duplication or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) or the Rights in Technical Data and Computer Software Clause as DFARS 252.227-7013 and FAR 52.227-19, as applicable. Manufacturer is Intel Corporation, 1350 Villa Street, Mountain View, California 94041-1126.

8. Limitation of Remedies and Damages. **To the maximum extent allowed by law, Intel shall not be responsible or liable with respect to any subject matter of this agreement under any contract, negligence, strict liability, or other theory: (a) for loss or inaccuracy of data or cost of procurement of substitute goods, services or technology; (b) for any special, indirect, incidental, or consequential damages including, but not limited to, loss of profits; or (c) for any matter beyond its reasonable control.**



Distribution of the Intel SDK is also subject to the following limitations: Licensees (i) are solely responsible to your customers for any update or support obligation or other liability that may arise from the distribution, (ii) do not make any statement that your product is "certified," or that its performance is guaranteed, by Intel, (iii) do not use Intel's name or trademarks to market your product without written permission, (iv) shall prohibit disassembly and reverse engineering, and (v) shall indemnify, hold harmless, and defend Intel and its suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from your distribution of any product.

9. Transfer; Successors. Licensee shall not assign this agreement or any part of it except with Intel's prior written consent.

10. General. This Agreement shall be governed by and construed under the laws of the State of Delaware and the United States without regard to conflicts of laws provisions thereof and without regard to the United Nations Convention on Contracts for the International Sale of Goods. In any action or proceeding to enforce rights under this Agreement, the prevailing party shall be entitled to recover costs and attorneys' fees. If any provision of this Agreement is held by a court of competent jurisdiction to be illegal, invalid or unenforceable, that provision shall be limited or eliminated to the minimum extent necessary so that this Agreement shall otherwise remain in full force and effect and enforceable. No rights or licenses with respect to the Intel SDK or Intel Products are granted, other than those rights expressly and unambiguously granted in this Agreement. This Agreement constitutes the entire agreement between the parties relating to the subject matter hereof.

Copyrights and Trademark Notification

Intel: Copyright ©1998-2000 Intel Corporation. All Rights reserved. **Trademark.** Intel is a trademark of Intel Corporation.

*Other products and company names mentioned herein may be the trademarks of their respective owners.

UCB: Contains Software from The Regents of the University of California. Copyright ©1982, 1986, 1993, 1997-2000 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistribution of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by the University of California, Berkeley, Network Research Group at Lawrence Berkeley National Laboratory and its contributors.
4. Neither the name of the University nor the Laboratory nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the Regents and contributors "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, are disclaimed. In no event shall the Regents or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

LCC: LCC Source Code from Addison Wesley Longman ("Licensor") from Christopher W. Fraser and David R. Hanson ("Authors"). LCC Source Code Copyright © 1995-2000 by David R. Hanson and AT&T. Reproduced by permission.

No warranty is made by Intel, the Licensor or the Authors of the LCC source code software, either express or implied, regarding the absence of defects in the LCC software, or its merchantability or fitness for a particular purpose. Intel, Licensor, and the Authors shall have no liability for damages of any nature arising out of any use, distribution, or modification of the LCC software, even if Intel, Licensor, or the Authors have been advised of the possibility of such damages.

GNU: Software coded using ARM Debugger and Compiler. Copyright ©1998-2000 Intel Corporation. All Rights reserved.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA 02139, USA.



Intel Corporation

1350 Villa Street

Mountain View, CA 94041-1126

Tel: 650.567.9800

Fax: 650.567.9810

www.netboost.com