Developing Applications Using the IX-API SDK

May 2000, Software Release SDK 3.0
Document Revision 2.3
Part Number 6750002

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

Contents

About This Guide

This guide introduces you to the Intel® IX-API SDK. This software developer's kit (SDK) enables you to develop and deliver wire-speed applications that

- Define policies about network packet traffic
- Enforce these policies
- Manage the networks and policies

To develop these policy-enforcement applications, you need the following:

- The IX-API SDK, a full set of tools for developing applications and run-time libraries for delivering applications
- The Intel Policy Accelerator 100™, a high-speed packet-processing board

This guide helps you get started creating and testing policy-enforcement network applications for Policy Accelerators using the IX-API SDK.

Audience

This guide is intended for software developers who will design, develop, and deliver network applications that must process packets at wire speed. 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 Guide

This guide includes the following chapters and appendixes:

- **Chapter 1, "Introducing the IX-API SDK,"** which describes the basic concepts of policy enforcement and the elements of the IX-API SDK
- **Chapter 2, "Tutorial: Creating a Simple Application,"** which takes you through the steps to create a simple working application and introduces additional key concepts for developing applications using the IX-API SDK
- **Chapter 3, "Elements of an Application,"** which gives an overview of the class and object framework, introduces the IX system software that coordinates objects on the host and in Policy Accelerator memory, and describes the error handling system
- **Chapter 4, "Compiling Applications,"** which explains how to compile and link all of the files for an application
- **Chapter 5, "Controlling Packet Flow,"** which explains how packets flow into and out of a Policy Accelerator within an application and introduces targets as destinations for packets
- **Chapter 6, "Classifying Packets Using NCL,"** which introduces the Network Classification Language and explains how to use it to evaluate packets
- **Chapter 7, "Acting on Packets in Your Action Code,"** which describes some of the actions that you can take on packets to enforce your networking policies and explains how to implement them
- **Chapter 8, "Communication Within an Application,"** which shows how to pass messages between the host and a Policy Accelerator and among applications on the Policy Accelerator using upcalls, downcalls, and crosscalls
- **Chapter 9, "Using Sets of Data to Classify Packets,"** which describes how to create data sets to associate data with packets, and how to perform and act on the results of searches in the sets
- **Chapter 10, "Finding Strings in Packets,"** which describes how to search for strings in the data portion of packet buffers, and act on the search results
- **Chapter 11, "Debugging and Troubleshooting,"** which discusses debugging techniques for different parts of your application code
- **Chapter 12, "Delivering Applications,"** which explains how to build an installer for the IX-API SDK runtimes when delivering an application to your end users
- **Appendix A, "Demonstration Applications,"** which describes the sample applications that come with the IX-API SDK

- **Appendix B, "Packet-Counting Application,"** which provides the complete source code for the application that you created in Chapter 2

Other Sources of Information

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

- *IX-API SDK Reference,* which describes the Intel IX-API SDK (software developer's kit), including both the standard host API (application programming interface) for supported operating systems and the API for the Policy Accelerator, which consists of the Action Services Library (ASL) and the Network Classification Language (NCL™)
- *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 xv.

Typographic 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.

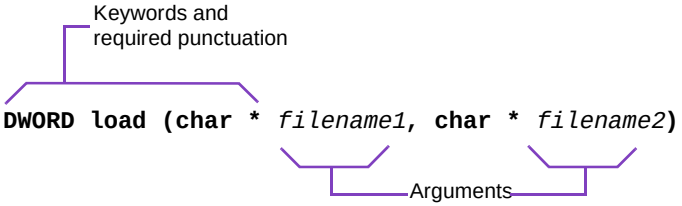
NOTE: Provides extra information, tips, and hints regarding the topic.

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

Customer Support Technicians

- **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`

Chapter 1
Introducing the IX-API SDK

This chapter introduces you to policy-enforcement applications and to the tools that Intel provides to develop and deliver them. It also describes the basic architecture of the applications that you will develop and provides background for Chapter 2, “Tutorial: Creating a Simple Application.”

This chapter contains the following topics:

- Policy-Enforcement Networking
- The Policy Accelerator Solution
- The Application Programming Interface
- Enforcing Policy by Classifying and Acting on Packets

Policy-Enforcement Networking

Networks have become a core component of businesses; their operation is critical to the businesses’ daily operation. Applications have become increasingly complex in order to manage, secure, and optimize these networks.

Application developers and equipment manufacturers must develop and enhance these applications to meet the rapidly changing needs of their users while ensuring that networks operate with optimal performance.

Policies

A *policy* is a decision made by a company, organization, or individual about how, where, and when various network operations are performed.

For example, a simple policy for packets traveling over a network is to send them on the shortest route through the network. This basic policy can be handled by basic solutions such as routing-table look-ups.

To implement more complex policies—such as dynamically altering routing patterns based on an automated evaluation of the traffic over the network—you must be able to create a specification for handling packets that describes how to enforce the policies you establish.

An application that includes such a specification is known as a *policy-enforcement application*.

Policy-Enforcement Applications

The following are common examples of policy-enforcement applications:

- Firewalls
- Intrusion detection systems
- RMON statistical monitioring applications
- Load-balancing systems
- Quality of service (QoS) applications

A policy-enforcement application must be able to

- Manage multiple policies and devices from anywhere in the network, for example, by changing policy configurations
- Enforce policies by taking action on data

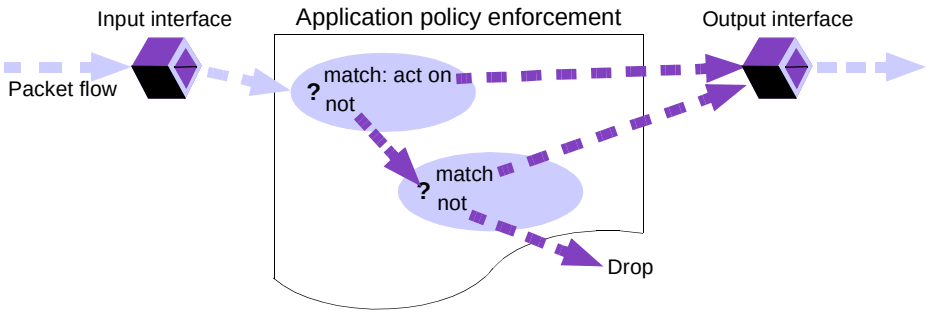
At the highest level, as shown in the following figure, policy-enforcement networking consists of the following:

- Interfaces to and from one or more networks
- Applications that process packets traveling on the networks

Packet Classification

An application's policy-enforcement part *classifies* the packets—that is, it compares the packet contents against some set of selection criteria. It then takes some *action* determined by the result of the classification and *disposes* of the packets, as shown in the following figure. For example, it might:

- View packets and simply decide whether to pass them to another interface or to drop them
- Collect information about the packets and report on the flow of data
- Manipulate the packets in some way



Policy Management

Policy-enforcement networking also requires that policies and network devices can be managed, either directly such as by embedded applications, or indirectly by network administrators using graphical user interfaces.

Performance Issues

Network speeds increase dramatically on a regular basis. Applications that affect the flow of traffic, such as policy-enforcement applications, need to operate at maximum wire speed to keep up with the technology.

Software-only solutions provide the flexibility required to quickly create and enhance applications. However, they typically run on general-purpose hardware, which cannot maintain wire-speed performance under the load typically placed on networks.

Fixed-function hardware can enforce a specific policy efficiently, but when asked to enforce multiple policies of a wide variety, its performance often degrades rapidly.

Ideally, policy-enforcement applications would be implemented using the flexibility of software development environments and run in an environment designed specifically to enforce policies.

The Policy Accelerator Solution

The IX-API SDK and Policy Accelerator 100 board allow you to quickly and easily develop and deliver policy-enforcement applications that operate at wire speed.

Product Components

This solution combines hardware and software that are designed specifically to create and run high-speed policy-enforcement applications:

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

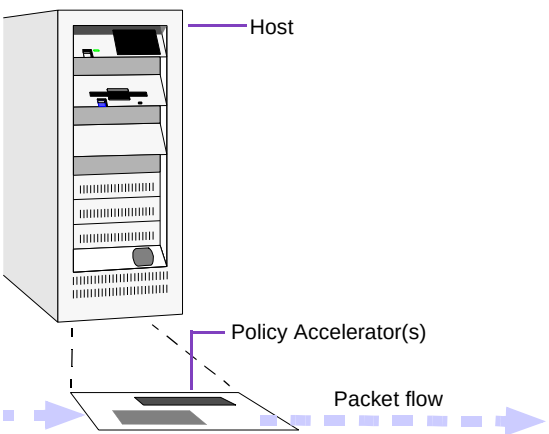
Policy Accelerator Boards

The Policy Accelerator is a wire-speed board designed specifically for enforcing policies. It does this by classifying packets, performing actions upon them, and disposing of them according to applications that you create.

You can equip a host system with one or more Policy Accelerators; each Policy Accelerator can support multiple applications. Policy Accelerators are available with the following interface formats:

- PCI card
- PMC card

For more information about Policy Accelerators, see the document *Installing a Policy Accelerator 100 Board.*



Software Developer's Kit (IX-API SDK)

The IX-API SDK is designed specifically to develop policy-enforcement applications. It contains the tools, libraries, drivers, and other run-time elements that you use to develop, debug, and deliver applications that work with the Policy Accelerator. It also contains an efficient language for describing and classifying packets. With the IX-API SDK, you can develop and deliver:

- The policy-enforcement part of your application, which runs on the Policy Accelerator
- The management portion of your application, which runs on the host

The IX-API SDK includes:

- Application programming interface (API)

  Functions, classes, and language support for the part of your application that runs on the host and for the part that runs on the Policy Accelerator, as described in "The Application Programming Interface" on page 6.

- System software

  Simple system software for the Policy Accelerator. This software can manipulate network packets with minimal overhead, which makes the Policy Accelerator ideal for running an application that does fast and simple manipulation of packet data.

- Run-time libraries

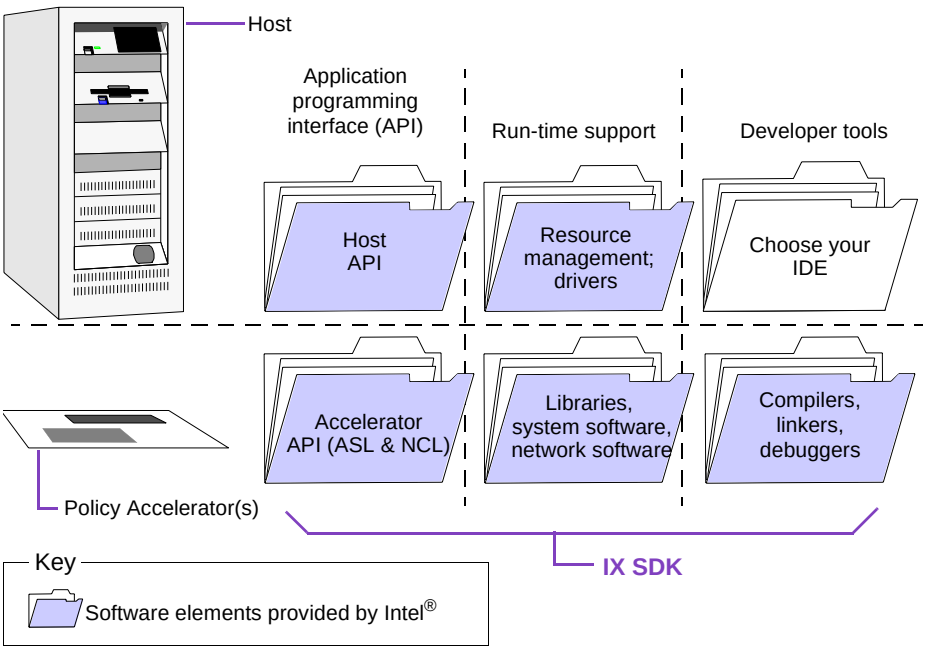
  Libraries that allow you to develop an efficient policy-enforcement application and deliver a completed application. On the host, the run-time libraries include Policy Accelerator device drivers and resource management tools. On the Policy Accelerator, the run-time libraries include support for C++ code and other services to support wire-speed policy enforcement.

- Software development tools

  Compilers, assemblers, linkers, loaders, and debuggers for code that runs on the Policy Accelerator.

- Predefined building blocks

  Examples of code for the host and for the Policy Accelerator, predefined protocol descriptions, and other elements to make application development rapid.



Your Host System

The host is the system on which the management portion of your policy-based application runs and in which one or more Policy Accelerators are installed.

The host provides basic nonnetworking services to your application, such as setting up the Policy Accelerator and communicating information about packet handling to the Policy Accelerator.

You can manage the application from anywhere on the network, not only from the host.

The Application Programming Interface

When you write a policy-enforcement application, the Policy Accelerator application programming interface (API) gives you the classes, objects, and languages needed to define and enforce policies, to manage packets, to communicate between the host and the Policy Accelerator, and to perform other related tasks.

API Components

The Policy Accelerator API includes the following:

- Host API

  C++ classes for the host portion of your application. These classes provide the ability to load software onto, initialize, and manage the Policy Accelerator.

- Action Services Library (ASL) for the Policy Accelerator

  C++ classes and functions for the Policy Accelerator portion of your application. This library provides efficient implementation for common packet-manipulation tasks.

- Network Classification Language (*NCL*™) for the Policy Accelerator

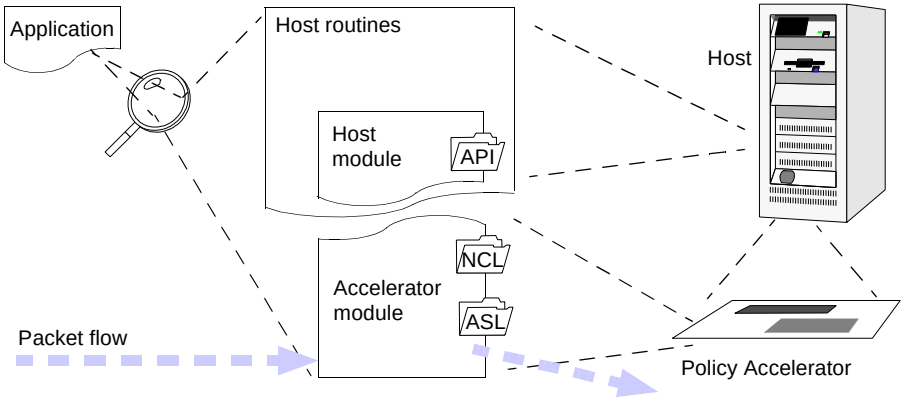
  A special-purpose language in which you define rules for implementing your company's policies. Using this language, you classify packets and direct the actions to be taken.

Structure of an Application

When you create a policy-based application for the Policy Accelerator, you use the API in the following modules:

- The *host module,* which runs on the host:
  - Uses the host API to initialize and communicate with the Policy Accelerator
  - Uses standard host services for other operations
- The *accelerator module,* which runs on the Policy Accelerator:
  - Uses NCL to classify packets
  - Uses the ASL to act on and dispose of packets

The following figure shows the structure of such an application:



The Host Module

This module manages the policies and network devices involved in policy-enforcement networking.

The host module is a C++ program that runs in the programming environment of your host's operating system. When you write this program, you can use all the conventional APIs in the host operating system to do traditional network processing.

In addition, the host module has access to all Policy Accelerators available on the system, so it can also do packet-level processing in conjunction with the accelerator module, using the host API.

The Accelerator Module

The accelerator module is responsible for enforcing your company's policies by classifying packets and by acting on them based on the classification.

The accelerator module consists of two types of code:

- *Rules* that describe your policies about processing packets

  You write the rules using the Network Classification Language (NCL™). These rules first *classify* packets according to your own set of criteria, and then specify what action to take with given classes of packets.

- An accompanying set of compiled *actions*, which implement the packet-processing policies defined by the rules

  You write the actions in C or C++ using the ASL.

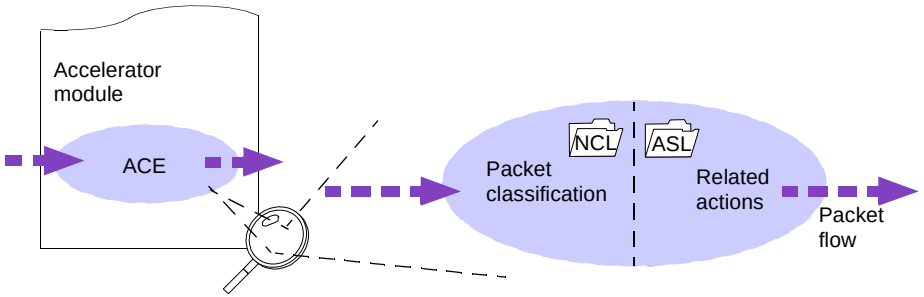
Enforcing Policy by Classifying and Acting on Packets

Because the accelerator module contains two types of code—NCL and C++—you need a structure that connects one set of rules with one related set of actions. This structure is an *Action/Classification Engine (ACE).*

The API provides methods with which you construct an ACE.

Action/ Classification Engines (ACEs)

An ACE represents the primary part of the accelerator module as shown in the following figure:



NOTE: An ACE is the primary object for processing packets through a policy-enforcement application.

Essentially, an ACE does three things with a packet:

- Classifies it
- Acts on it
- Determines its disposition

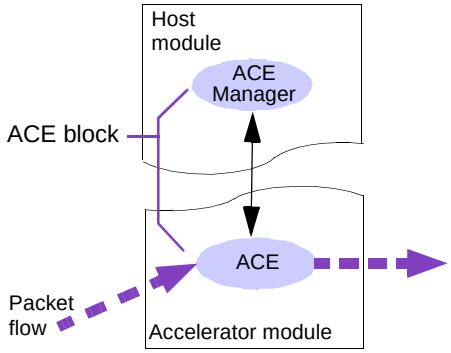
It classifies a packet using rules that you write in NCL. It performs specific actions and dispositions using the *action code* that you write in C++ using the ASL.

Managing ACEs

To reference an ACE on the Policy Accelerator, your host module must have an *ACE manager* to do the following:

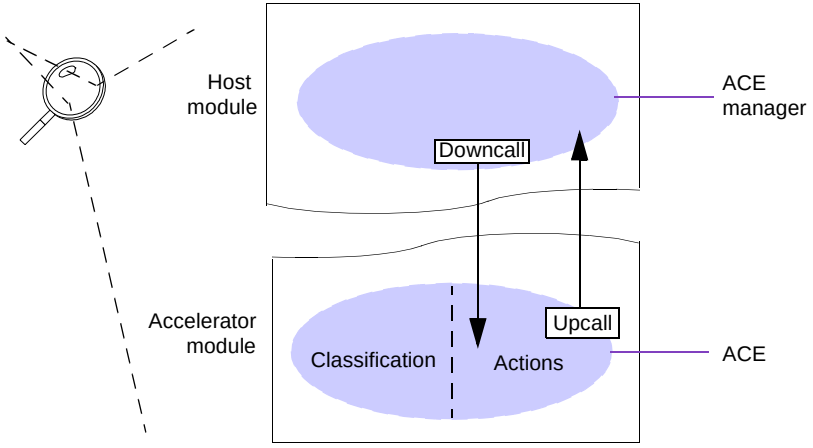
- Create and manage the ACE
- Load the accelerator module into the Policy Accelerator
- Communicate with the Policy Accelerator

Each ACE has exactly one ACE manager, and each ACE manager manages exactly one ACE. The combination of an ACE and an ACE manager is known as an *ACE block*, as shown in the following figure:



Messages Between the Host and the 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:



Developing Applications

The following chapter, “Tutorial: Creating a Simple Application,” takes you through the steps to creating and running a simple application. In short, you will:

- Create the host module, which consists of
  - Class definitions and other header declarations for the main code
  - The C++ code to run the host process and to manage the Policy Accelerator
- Create the accelerator module, which consists of
  - Network Classification Language (NCL) declarations that classify packets and specify the actions to perform
  - Action code that is invoked by the NCL
  - Class definitions and other header declarations for the action code

Chapter 2
Tutorial: Creating a Simple Application

In this chapter, you create a small sample application that demonstrates the important basic aspects of building a policy-enforcement application using the IX-API SDK. This tutorial helps you to quickly create an application, to learn what pieces are required, and how to compile and run the application.

This chapter also introduces the basic set of C++ classes from the host API and from the ASL, which you will use in all of your applications.

Overview of the Tutorial

This tutorial takes you through the steps to create a sample application that uses a Policy Accelerator to count packets flowing through a network. The finished application displays text about the packet count.

This chapter contains the following topics:

Topic	Content
Creating Source File Outlines	■ Introduces the basic API C++ classes and header files ■ Creates outlines for all the files that the sample application needs
Turning On Debugging	■ Shows how to turn on verbose mode
Preparing for Error Handling	■ Defines error codes ■ Shows simple error handling strategies
Creating the Primary Application Object (NBAppl)	■ Introduces the primary application object
Creating ACE Objects in the Host Module	■ Introduces ACE groups and ACE managers

Topic	Content
Loading and Initializing the Policy Accelerator	■ Creates an ACE in the Policy Accelerator module ■ Loads NCL code and compiled C++ code onto the Policy Accelerator
Creating an ACE Method	■ Shows how to implement application functionality in methods
Sending Messages from the Policy Accelerator to the Host	■ Shows how the host and accelerator modules communicate ■ Shows a simple example of byte-order management
Defining Packet Flow	■ Explains how packets flow through a Policy Accelerator ■ Binds interfaces and ACEs to control packet flow
Classifying and Acting on Packets	■ Explains NCL classification rules ■ Describes action functions
Compiling, Linking, and Running the Application	■ Verifies your development environment ■ Shows how to compile your application ■ Shows how to run the application and view output

Using the Tutorial

You can use the sample code in the tutorial by copying and pasting the code shown here into your own files.

NOTE: If you are working on a machine that does not have the IX-API SDK installed, you can still follow the tutorial by copying and pasting the code as directed into a text file. This demonstrates how an application is structured.

If you are viewing this document using Acrobat® Reader, choose the Select Text command from the Tools menu to select and copy text.

Different Platforms

This sample source code contains conditional inclusions for slightly different coding requirements for Windows NT applications and for UNIX applications.

Complete Source Code

You can view the complete code that you will create in this tutorial in the following places:

- Appendix B, “Packet-Counting Application.”
- Online in *SDKinstallpath*/demos/CountApp

Creating Source File Outlines

In this section, you create an outline of the application in the form of comments. In later sections, you will replace the comments with actual code to create the source files which you will later compile and execute.

C++ Classes for Tutorial

This tutorial introduces the minimum set of API C++ classes that you need when you develop an application for the Policy Accelerator. The following tables show the classes used for objects in this sample application.

On the host side:

NBAppl	Represents the Intel portion of the host application code
AceGroup	Contains one or more ACE managers and their related ACE objects
AceManager	Communicates with ACEs on the Policy Accelerator
UpcallHandler	Receives messages from upcall objects on the Policy Accelerator

On the Policy Accelerator side:

Ace	Moves packets through a Policy Accelerator
Upcall	Delivers messages to the host
Message	Contains message blocks to be sent in an upcall
MessageBlock	Describes and contains data to be sent in an upcall

Files Required

In general, an application needs the following files:

- For each application:
  - .cpp file for host module source code
  - .h header file for host module class declarations

- For each ACE in the application:
  - `.cpp` file for accelerator module action code
  - `.ncl` file for accelerator module classification rules in NCL

The tutorial sample application uses only one ACE, defined in one set of files. You will create files named:

- `CountApp.h` (host header file)
- `CountApp.cpp` (host module file)
- `CountActions.cpp` (actions file)
- `CountRules.ncl` (classification rules, or NCL, file)

Your Development Environment

You can use any convenient tool, such as a C++ editor, to create and edit your C++ source files. Use a simple text editor to create the NCL file.

Creating the Source Files

In this tutorial, you first create all of the files you will need and fill them with placeholders in the form of comments. In the rest of the tutorial, you gradually replace the placeholders with actual code.

NOTE: You do not need to understand the purpose of the placeholder comments yet. The following sections in this tutorial provide explanations as you build your application.

Create all of your source files on your host, as follows:

1. Create the host header file, `CountApp.h`, and insert the following placeholders:

```
// #include system interface header files
// user-#defined error codes
// AceManager subclass declaration
//       showPacketCount method
//       pointer to UpcallHandler object
// AceGroup subclass declaration
//       pointer to AceManager subclass object
// NBAppl subclass declaration, including:
//       pointer to AceGroup subclass object
```

2. Create the host module file, `CountApp.cpp`, and insert the following placeholders and code:

```
#ifdef Win32
   #include <iostream.h>
   #include <direct.h>
```

```
#else
   #include <unistd.h>
   #include <stdlib.h>
#endif

#include CountApp.h

// NBAppl subclass definition, including:
//       AceGroup subclass instantiation
//       Network interface bindings
// AceGroup subclass definition, including:
//       AceManager subclass instantiation
// AceManager subclass definition, including:
//       UpcallHandler instantiation
//       load accelerator module into Policy Accelerator
//       showPacketCount method definition
void main (int argc, char** argv) {
   // get current directory
   // NBAppl subclass instantiation
   while(1)
        #ifdef Win32
           _sleep(999999);
        #else
           sched_yield();
        #endif
}
```

3. Create the actions file, `CountActions.cpp`, and insert the following placeholders:

```
// #include system interface header files
// Ace subclass declaration, including:
//       showPacketCount method
//       packet counter declaration
//       Upcall subclass declaration
// Ace subclass definition, including:
//       Upcall subclass instantiation
//       initialize packet counter
//       showPacketCount method definition, including:
//          increment counter
//          take snapshot of value with proper byte order
//          create message
//          invoke upcall
// action function entry point (sends packet count)
// main-equivalent (init_actions), including:
//       instantiate and return Ace subclass
```

4. Create the NCL file, `CountRules.ncl`, and insert the following placeholder:

```
// rule to invoke action for all packets
```

Using the SDK Header Files

The IX-API SDK includes header files that declare the classes, methods, functions, and types you need when developing your application.

The header files are grouped into directories based on where you use them, as follows:

Purpose	Located in directory	Primary header file
System definitions for host module and accelerator actions	*SDKinstallpath*`/include/`	`NBswap.h`
Host classes and methods	*SDKinstallpath*`/include/NBapi/`	`nbappl.h`
Accelerator classes, functions, and types	*SDKinstallpath*`/include/NBaction/`	`NBAction.h`
NCL common protocol declarations	*SDKinstallpath*`/include/NBncl/`	`NBbase.h` or `NBtcpip.h`

NOTE: The installation path for the IX-API SDK is represented here by *SDKinstallpath*; installing the IX-API SDK automatically sets the environment variable `NBPATH` to the correct path.

Each directory contains several header files. Because the primary header file in each directory includes the rest of the files in the same directory directly or indirectly, you need to include only the primary header files, as follows:

To include the appropriate header files:

1. In the *host header file*, `CountApp.h`, replace:

```
// #include system interface header files
```

with the following:

```
#include "NBswap.h"
#include "NBapi/nbappl.h"
```

2. In the *actions file*, `CountActions.cpp`, replace:

   ```
   // #include system interface header files
   ```

   with the following:

   ```
   #include "NBaction/NBAction.h"
   ```

NOTE: This sample application does not require any of the predefined NCL header files.

Turning On Debugging

Debugging methods are different for the host module and for the accelerator module.

This section provides only an overview. For more information on debugging, see Chapter 11, "Debugging and Troubleshooting."

Host Module Debugging

To debug the host module, use the standard host debugging tools.

The SDK provides an additional macro that you can use to can turn on tracing information related to objects defined using the API. This tracing information displays in the host application's console window.

1. Turn on verbose mode in the SDK. To do this, add the highlighted line to `main`:

   ```
   void main (int argc, char** argv) {
      nb_trace_verbose(1);
   ```

Accelerator Module Debugging

To debug the accelerator module, you can use two methods:

- Embed standard C-language `printf` statements in the code.
- Attach a debug card to the Policy Accelerator and follow the instructions for compiling and debugging the accelerator module given in Chapter 11, "Debugging and Troubleshooting."

This tutorial does not show debugging for the accelerator module.

Preparing for Error Handling

The host API handles error checking differently in the following situations:

- For most class methods, returns a code of type `NBError`; the value `NB_SUCCESS` indicates that the method succeeded
- For class constructors, throws an exception of type `NBError` if the constructor fails; you can retrieve an error code that describes the reason for the failure

Defining Error Codes

You can use the detailed set of error codes already defined by `NBError.h`, or you can add your own unique numbered error codes for providing information about failed operations. This sample application adds its own error codes.

Your error codes must be greater than the constant `NBERROR_USER_BASE`, which is defined in the host API header files that you included in "Creating Source File Outlines" on page 15.

In the sample's host module, there are five basic situations that require error handling. The following table shows the situations and how this sample handles them:

Situation	Error handling strategy
`NBAppl` subclass instantiation	Catch any `NBError` and exit.
`AceGroup` subclass instantiation	Catch any `NBError` and throw an exception using your own error codes.
`AceManager` subclass instantiation	
`UpcallHandler` subclass instantiation	
Loading the accelerator module into the Policy Accelerator	Check error code for `NB_SUCCESS` and throw an exception using your own error codes.

To define your own errors:

1. To the *host header file*, `CountApp.h`, add the definitions for your own application's base error code. To do this, replace:

   ```
   // user-#defined error codes
   ```

   with the following basic definitions:

   ```
   #define NBERROR_COUNT_BASE (NBERROR_USER_BASE + 0x1000)
   #define NBERROR_COUNT_ERRCODE (x) (NBERROR_COUNT_BASE + (x))
   ```

2. After the basic definitions, add the following error definitions:

```
#define NBERROR_COUNT_CANNOTCREATEGROUP \
                        NBERROR_COUNT_ERRCODE(1)
#define NBERROR_COUNT_CANNOTCREATEACE \
                        NBERROR_COUNT_ERRCODE(2)
#define NBERROR_COUNT_CANNOTCREATEUPCALL \
                        NBERROR_COUNT_ERRCODE(3)
#define NBERROR_COUNT_CANNOTLOADCODE \
                        NBERROR_COUNT_ERRCODE(4)
```

3. In the *host module file*, `CountApp.cpp`, this sample catches possible `NBError` exceptions thrown by subclass instantiations and throws its own custom error codes.

   You will do this later in this tutorial. The code looks similar to the following:

```
catch (NBError&) {
   throw NBError(
        NB_ERROR(NBERROR_COUNT_CANNOTCREATEACE));
}
```

   Passing a reference to `NBError`, as shown, could be more efficient than passing an `NBError` element, such as in `catch (NBError E)`.

4. For the `load` method, which loads the accelerator module into the Policy Accelerator, this sample checks the returned error code. If the error code is not `NB_SUCCESS`, it throws its own custom error code.

   You will do this later in this tutorial. The code looks similar to the following:

```
int errorcode;
if ((errorcode = load ("CountRules", // NCL (.ncl) file
                  "CountActions")) // action (.nbo) code
          != NB_SUCCESS)
{
   throw NBError (NB_ERROR(NBERROR_COUNT_CANNOTLOADCODE));
}
```

5. For the creation of the `NBAppl` subclass, this sample simply exits if `NBError` occurs.

   You will do this later in this tutorial. The code looks similar to the following:

```
catch (NBError&) {
   fprintf(stderr, "CountAppl creation failed!\n");
   exit(2);
}
```

Creating the Primary Application Object (NBAppl)

Each application must create a subclass of the NBAppl class to serve as the primary application object in the host module. This NBAppl object represents the Intel portion of the application. A system process called the *Resolver* uses this NBAppl object to track and to communicate with all IX-API SDK applications and objects. (For more information on the Resolver, see "The Resolver and Multiple Applications" on page 54.)

About the Main Function

The main function of the host module instantiates an NBAppl object and then goes to sleep, as shown in "Creating the Source Files" on page 16. This is a typical model for the main function. The NBAppl object runs in a different thread from main so that it can more efficiently handle the operations that it must do in conjunction with the Policy Accelerator.

Creating the NBAppl Object

To create the NBAppl object:

1. Choose the following names:
   - A dictionary name for your application; for this sample, use *Count-Packets*. This name is used when creating objects within the application.
   - The subclass name for your NBAppl subclass; for this sample, use *CountAppl*.
2. To the *host header file*, CountApp.h, add the declaration for the NBAppl subclass CountAppl. To do this, replace:

   ```
   // NBAppl subclass declaration, including:
   //       pointer to AceGroup subclass object
   ```

   with the following:

   ```
   class CountAppl: public NBAppl {
   public:
      CountAppl (char* name, char* curdir, char* cmdLine);
      ~CountAppl();
   //       pointer to AceGroup subclass object
   };
   ```

3. To the *host module file*, CountApp.cpp, add the class definition for CountAppl. To do this, replace:

   ```
   // NBAppl subclass definition, including:
   //       AceGroup subclass instantiation
   //       Network interface bindings
   ```

with the following:

```
CountAppl::CountAppl (char* name,
          char* cwd, char* cmd):
   NBAppl (name, cwd, cmd)
{
   //       AceGroup subclass instantiation
   //       Network interface bindings
}
CountAppl::~CountAppl()
{
   //       delete AceGroup subclass object
}
```

4. Add the instantiation of `CountAppl`. To do this, replace the following in `main`:

```
//       // NBAppl subclass instantiation
```

with the following, using the dictionary name you chose for your application (CountPackets):

```
CountAppl* appl;
   try {
        appl =
           new CountAppl ("CountPackets", NULL, NULL);
   }
   catch (NBError&) {
        fprintf(stderr, "CountAppl creation failed!\n");
        exit(2);
   }
```

Creating ACE Objects in the Host Module

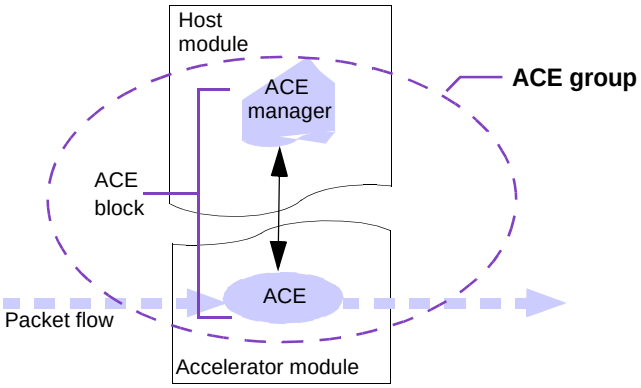
This section introduces ACE groups and shows how to create ACEs and ACE managers. ACEs, ACE managers, and ACE blocks were introduced in “Enforcing Policy by Classifying and Acting on Packets” on page 8 in Chapter .

About ACE Objects

An *ACE block* is the combination of an ACE in the accelerator module and its ACE manager in the host module. In an application, every ACE block must be contained in an *ACE group.* An ACE group is a way of connecting ACE blocks that have some kind of relationship, such as sharing resources or performing related functions.

NOTE: You can have more than one ACE block in an ACE group, although the sample application has only one.

The following figure shows the relationship among the various ACE objects:



The ACE group and the ACE manager are objects that reside on the host; the ACE object resides on the Policy Accelerator.

Creating an ACE Group

To create an ACE group:

1. To the *host header file*, `CountApp.h`, add the declaration for the `AceGroup` subclass `CountAceGroup`. To do this, replace:

```
// AceGroup subclass declaration
//       pointer to AceManager subclass object
```

with the following:

```
class CountAceGroup: public AceGroup {
   public:
   CountAceGroup (NBAppl* appl, NBFactory* nbf,
        char* name, NBStringList* list);
   ~CountAceGroup();
   //       pointer to AceManager subclass object
   };
```

2. To the *host module file*, `CountApp.cpp`, add the class definition for `CountAceGroup`. To do this, replace:

```
// AceGroup subclass definition, including:
//       AceManager subclass instantiation
```

with the following:

```
CountAceGroup::CountAceGroup (NBAppl* appl, NBFactory* nbf,
       char* name, NBStringList* list) :
   AceGroup (appl, nbf, name, list)
//       AceManager subclass instantiation
```

```
CountAceGroup::~CountAceGroup()
{
//       delete AceManager subclass object
}
```

Creating an ACE Manager

As introduced in "Enforcing Policy by Classifying and Acting on Packets" on page 8, the ACE manager is responsible for creating and managing an ACE on the Policy Accelerator, and loading the accelerator module into the Policy Accelerator.

The ACE manager also provides a convenient way to group operations on the host that support the ACE and to represent on the host the state of the ACE.

To create an ACE manager:

1. To the *host header file*, `CountApp.h`, add the declaration for the `AceManager` subclass `CountAceManager`. To do this, replace:

   ```
   // AceManager subclass declaration
   //       showPacketCount method
   //       pointer to UpcallHandler object
   ```

   with the following:

   ```
   class CountAceManager: public AceManager
   {
      public:
      CountAceManager (NBAppl* appl, AceGroup*
           acegroup, char* name);
      ~CountAceManager();
   //       showPacketCount method
   //       pointer to UpcallHandler object
   };
   ```

   Note that the constructor for the ACE manager shows that you must specify which ACE group to associate the ACE manager with when you create the manager.

2. To the *host module file*, `CountApp.cpp`, add the class definition for `CountAceManager`. To do this, replace:

   ```
   // AceManager subclass definition, including:
   //       UpcallHandler instantiation
   //       load accelerator module into Policy Accelerator
   //       showPacketCount method definition
   ```

   with the following:

   ```
   CountAceManager::CountAceManager (NBAppl* appl,
                AceGroup* acegroup,
   ```

```
                char* name):
AceManager (appl, acegroup, name)
{
   //       UpcallHandler instantiation
   //       load accelerator module into Policy Accelerator
}
//       showPacketCount method definition
CountAceManager::~CountAceManager()
{
//       delete UpcallHandler object
}
```

Cascading Instantiations

You must create an ACE group before you create an ACE manager, and you must create an `NBAppl` before you create an ACE group. It is good practice to cascade the instantiations so that creating the `NBAppl` then creates all of the ACE groups used inside it, and creating an ACE group creates its ACE managers.

1. To the *host header file*, `CountApp.h`, add pointers for the nested classes that will be instantiated. To do this:

   a. In the `CountAceGroup` class declaration, replace:

   ```
   //       pointer to AceManager subclass object
   ```

   with the following:

   ```
   protected:
        CountAceManager* countAceManager;
   ```

   b. In the `CountAppl` class declaration, replace:

   ```
   //       pointer to AceGroup subclass object
   ```

   with the following:

   ```
   protected:
        CountAceGroup* countAceGroup;
   ```

2. To the *host module file*, `CountApp.cpp`, add the instantiations for the nested classes to the class definitions, and delete the nested classes in the class destructors. To do this:

   a. In the `CountAceGroup` class definition, instantiate the `AceManager` subclass and check for thrown exceptions. To do this, replace:

   ```
   //       AceManager subclass instantiation
   ```

with the following:

```
{
     try {
        countAceManager =
           new CountAceManager (appl, this, "CountAce");
     }
     catch (NBError&) {
        throw NBError(
           NB_ERROR(NBERROR_COUNT_CANNOTCREATEACE));
     }
}
```

b. In the destructor, replace:

```
//       delete AceManager subclass object
```

with the following:

```
delete countAceManager;
```

c. In the `CountAppl` class definition, instantiate the `AceGroup` subclass and catch any exception that might be thrown. To do this, replace:

```
//       AceGroup subclass instantiation
```

with the following:

```
try {
     countAceGroup = new CountAceGroup (this, NULL,
        "CountAceGroup", NULL);
}
catch (NBError&) {
     throw NBError (NB_ERROR
           (NBERROR_COUNT_CANNOTCREATEGROUP));
}
```

d. In the destructor, replace:

```
//       delete AceGroup subclass object
```

with the following:

```
delete countAceGroup;
```

Loading and Initializing the Policy Accelerator

The most important task of an ACE manager is to load the accelerator module (the action code and the classification rules) into the Policy Accelerator, which simultaneously initializes the Policy Accelerator and creates the ACE that is associated with the ACE manager. The steps to do this are:

- Loading the Accelerator Module
- Implementing an ACE
- Creating the Initialization Function for the Accelerator Module

Loading the Accelerator Module

Your application loads the accelerator module into the Policy Accelerator. It does this using the `load` method of your `AceManager` subclass. A convenient time to do this is when the ACE manager is created, by calling `load` in the `AceManager` subclass constructor.

The accelerator module consists of two files:

- Your application's classification rules NCL file; in this sample, `CountRules.ncl`
- Your application's action code, a compiled C++ object-code file with the special suffix `.nbo`; in this sample, `CountActions.nbo`

To load the accelerator module into the Policy Accelerator:

1. In the *host module file*, `CountApp.cpp`, in the `AceManager` subclass constructor, replace:

   ```
   //       load accelerator module into Policy Accelerator
   ```

   with the following:

   ```
   int errorcode;
   if ((errorcode =
          load ("CountRules",    // NCL (.ncl) file
                "CountActions")) // action (.nbo) code
             != NB_SUCCESS)
   {
      throw NBError (NB_ERROR(NBERROR_COUNT_CANNOTLOADCODE));
   }
   ```

Implementing an ACE

As described in "Enforcing Policy by Classifying and Acting on Packets" on page 8, an ACE controls the flow of packets by connecting your NCL rules with your compiled C++ action code. The NCL rules classify packets and call specified action functions, which then act on the packets and their contents.

To create an ACE, you define and implement an Ace subclass in the accelerator module as follows:

1. To the *actions file*, CountActions.cpp, add the definition for the Ace subclass CountAce. To do this, replace:

```
// Ace subclass declaration, including:
//       showPacketCount method
//       packet counter declaration
//       Upcall subclass declaration
```

   with the following:

```
class CountAce: public Ace {
public:
   CountAce (ModuleId id, char* name, image* obj);
   //       showPacketCount method
   //       packet counter declaration
   //       Upcall subclass declaration
};
```

2. Add the class implementation for CountAce. To do this, replace:

```
// Ace subclass definition, including:
```

   with the following:

```
CountAce::CountAce (ModuleId id, char* name, Image* obj) :
   Ace (id, name, obj), // note ending (incomplete) comma
```

Creating the Initialization Function for the Accelerator Module

Just as a C or C++ program in a standard environment has a main entry point, so does the C++ action code for each ACE in the accelerator module.

In a standard environment, the main function is named main. In the action code, the function is named init_actions. The Policy Accelerator transfers control to this function as soon as the accelerator module is loaded, which occurs as described in “Loading and Initializing the Policy Accelerator” on page 28.

This function must:

- Take three arguments—an ID, a name, and an object—which the SDK run-time libraries automatically pass to the function, and which the function must pass to the ACE constructor
- Instantiate an ACE
- Return the ACE

To create this main function:

1. In the *actions file*, `CountActions.cpp`, replace:

```
// main-equivalent (init_actions), including:
//       instantiate and return Ace subclass
```

with the following:

```
INITF init_actions (void* id, char* name, Image* obj)
{
//       instantiate and return Ace subclass
}
```

2. Create and return the ACE object. To do this, replace:

```
//       instantiate and return Ace subclass
```

with the following:

```
return new CountAce (id, name, obj);
```

Creating an ACE Method

The purpose of this simple example application is to count packets that flow into the Policy Accelerator. A method in the ACE object in the accelerator module keeps this count. Every time a packet flows into the Policy Accelerator, a rule triggers an action function that calls this method. You will define the rule and action function later.

This method, which is in the accelerator module, also periodically informs the host module of the current packet count, using a message and upcall. You will create the message and upcall in the next section.

To create a packet counter method:

1. In the *actions file*, `CountActions.cpp`, add a packet counter method and counter to the declaration for the subclass `CountAce`. To do this, replace:

```
//       showPacketCount method
//       packet counter declaration
```

with the following:

```
void showPacketCount (void);
int packetCounter;
```

2. In the constructor for `CountAce`, initialize the counter. To do this, replace:

```
//       initialize packet counter
```

with the following:

```
{
   packetCounter = 0;
};
```

3. Define the packet counter method for `CountAce` to increment the counter, and check whether it is time to send the count to the host.

To do this, replace:

```
//       showPacketCount method definition, including:
//          increment counter
//          take snapshot of value with proper byte order
//          create message
//          invoke upcall
```

with the following:

```
void CountAce::showPacketCount (void)
{
   packetCounter++;
   if (!((packetCounter-1)%0x20))
   // Don't do upcall for every packet; could
   // overwhelm the driver
   {
//          take snapshot of value with proper byte order
//          create message
//          invoke upcall
   }
};
```

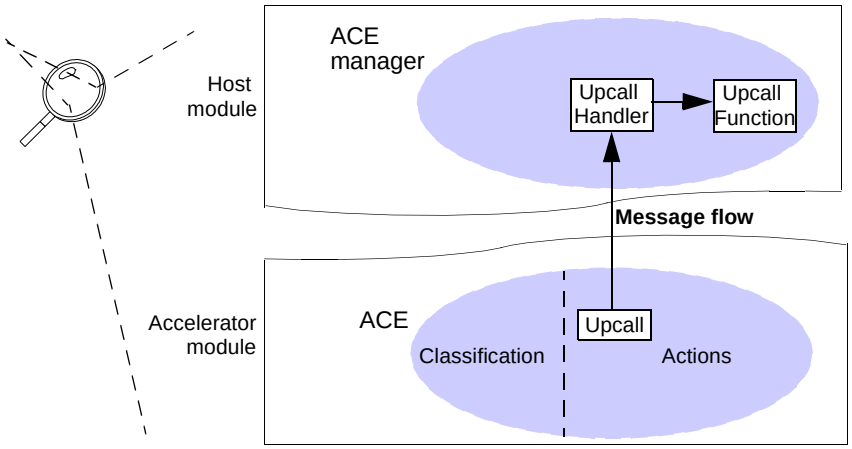
Sending Messages from the Policy Accelerator to the Host

The host module and the accelerator module communicate using asynchronous messages. The sending module must have an object that it uses to send the message; the receiving module must have an object that it uses to look for and receive messages. When sending messages from the Policy Accelerator to the host, these objects are subclasses of `Upcall` and `UpcallHandler`, respectively.

In this sample, the accelerator module sends to the host module a count of the number of packets processed. The host module then displays a message about the packet count.

To send a packet count, your application must:

- Create an `Upcall` object in the accelerator module
- Create an `UpcallHandler` object in the host module
- Take a snapshot of the packet count to use in the message
- Create a message using message blocks in the accelerator module
- Ensure that the byte order (endianness) of numbers sent between the modules is correct
- Send the message using the upcall
- Receive the message in the host module using an upcall's callback function, interpret for byte order, and delete the message



Creating a Message

The `Message` class creates a container for information that must travel between the host and the Policy Accelerator. You must describe the information's type and size. The `MessageBlock` class provides a simple way to do this.

The message block points to a memory location that contains the data to be sent. Because message passing is asynchronous, the location that you define for this data must have the following properties:

- The buffer must still be there when the message is passed—that is, it cannot be an automatic variable that goes out of scope and might be freed before the call is completed.
- The buffer must still contain the same data when the message is passed—for example, you cannot use the dynamic `packetCounter` variable, which continues to be updated before the call is completed.

For this small piece of data, you will define another variable in the ACE object, copy a snapshot of the current packet count into it, and point the message to it.

To create a message:

1. In the ACE subclass definition in the *actions file*, `CountActions.cpp`, add an ACE variable to hold the message data. The variable should hold a `nuint32` value (the next subsection, "Using a Network-Byte-Ordered Integer," explains why).

   To do this, add the highlighted line to the definition of the ACE subclass:

   ```
   void showPacketCount (void);
   int packetCounter;
   nuint32 countSnapshot;
   ```

2. In the packet counter method in the *actions file*, `CountActions.cpp`, declare a message block to point to the new ACE variable. To do this, insert the highlighted line into the packet counting method as shown:

   ```
      MessageBlock b ((char *) &countSnapshot,
                              sizeof (countSnapshot));
   //          create message
   ```

3. Create a message to contain the message block. To do this, replace:

   ```
   //          create message
   ```

   with the following:

   ```
      Message msg (mb);
   ```

Using a Network-Byte-Ordered Integer

Network programming requires that you use numbers of specific known widths. The C and C++ languages, however, do not guarantee any particular width for basic data types nor do they guarantee that the coding sequence to establish widths will be the same for all compilers. The SDK defines numeric types that you use to ensure a known width.

In addition, when passing data between the host and the Policy Accelerator, you must ensure that the data is stored in the correct byte order. The byte order of data, know to compilers as its *endianness*, can be most-significant-byte-first (*big endian* compilers) or least-significant-byte first (*little endian* compilers). The network uses most-significant-byte-first order, so the IX SDK calls this network byte order.

The application host or the originator of network data might or might not use a different byte order from the network. Therefore, to ensure accuracy, portability, and efficiency, the SDK includes several basic types, type classes, functions, and methods that you use to ensure a known byte order when transporting numbers over a network.

Before you send any numeric data in a message, you must convert the data from host to network byte order. When you receive the message, you convert it from network to host byte order. The conversion methods ensure that, whether the byte orders are the same or different, the integrity of the data is protected. This conversion process is called *marshalling* and *unmarshalling* the message arguments.

This sample application uses the following:

- The class `nuint32` to hold a network-compatible version of the packet counter variable
- Methods to convert the data to and from network byte order:
  - `htonl` (host-to-network-byte-order long integer, where *host* in this case means the sender of the data, not necessarily the host system or host module)
  - `ntohl` (network-byte-order-to-host long integer)

To create message data of the proper type:

1. In the *actions file*, `CountActions.cpp`, in the packet counter method for `CountAce`, take a snapshot of the current counter value and copy it to the message variable. Convert the integer value to `nuint32` using the `htonl` method.

   To do this, replace:

   ```
   //          take snapshot of value with proper byte order
   ```

   with the following:

   ```
   countSnapshot = htonl (packetCounter);
   ```

The packet counter method now looks like this:

```
void CountAce::showPacketCount (void)
{
   packetCounter++; //increment counter
   if (!((packetCounter-1)%0x20)) // Don't do upcall for every
                        //packet; could overwhelm the driver
   {
      countSnapshot = htonl(packetCounter); //take snapshot
      MessageBlock mb ((char *) &countSnapshot,
                           sizeof (countSnapshot));
```

```
      Message msg (mb); //create message
   //          invoke upcall
   }
}
```

Sending a Message with an Upcall

An application can use upcalls to share information or to signal from the accelerator module to the host module.

NOTE: You should not use upcalls to pass large quantities of packets between the host and the Policy Accelerator. Use the stack to forward packets, as described in “Moving Packets between the Policy Accelerator and the Host” on page 114 in Chapter .

An upcall is represented by a class that contains a method, `call`, for sending a message to the host. To create an upcall and send a message:

1. Choose a dictionary name for your upcall.

   This name is used as its internal identifier; you must use the same name spelled the same way, including capitalization, on the host to make the connection work.

   For this sample, use the name `showPacketCount`.

2. In the *actions file*, `CountActions.cpp`, add a declaration for a subclass of `Upcall` to the declaration for the subclass `CountAce`.

   You can give the `Upcall` subclass any name; it does not have any relationship to the dictionary name. For readability, however, this sample names the subclass `showPacketCountUpcall`.

   To add the declaration, replace:

   ```
   //       Upcall subclass declaration
   ```

   with the following:

   ```
   protected:
      Upcall showPacketCountUpcall;
   ```

3. Use member initialization to create the `Upcall` subclass in the `CountAce` constructor.

   Error handling is intentionally limited in the ASL, so member initialization is an efficient way to instantiate classes where possible.

   To create the `Upcall` subclass, replace:

   ```
   //       Upcall subclass instantiation
   ```

with the following, using the dictionary name you chose in Step 1:

```
showPacketCountUpcall (id, this, "showPacketCount")
```

4. Using the `call` method of the upcall handler, send the message that you created earlier. To do this, replace:

   ```
   //          invoke upcall
   ```

   with the following:

   ```
   if (showPacketCountUpcall.call(&msg) != 0 )
      printf("Upcall failed.\n");
   ```

Receiving a Message with an Upcall Handler

Because message passing is asynchronous, you must have a way of telling your host module to expect that the accelerator module might attempt to communicate. The `UpcallHandler` class provides this capability.

This class:

- Identifies the name of the upcall coming from the accelerator module
- Specifies a local *upcall function* to execute when an upcall is received; this upcall function receives the message and then does whatever actions you choose

In this sample, the upcall function converts the network-byte-ordered packet counter to the correct byte order and displays its value.

To create an upcall handler and its local function in an ACE manager:

1. To the *host header file*, `CountApp.h`, add a pointer for the `UpcallHandler` object.

   To do this, replace:

   ```
   //       pointer to UpcallHandler object
   ```

   with the following:

   ```
   UpcallHandler* showPacketCountUpcallHandler;
   ```

2. Add a function that takes a `Message` as an argument and converts the number from network byte order.

   This function should be a method of the `AceManager` subclass. To create this method:

   a. In the *host header file*, `CountApp.h`, replace:

      ```
      //       showPacketCount method
      ```

with the following:

```
void showPacketCount (Message* m);
```

b. In the *host module file*, `CountApp.cpp`, replace:

```
//       showPacketCount method definition
```

with the following:

```
void CountAceManager::showPacketCount (Message* m)
{
     NB_ASSERT (m->getLen1() == sizeof(nuint32));
     printf ("Packet Counter: 0x%08x\n",
        ntohl (*(nuint32 *)m->getBuffer1()));
     releaseMessage (m); // Message passed here,
                              // dispose of it.
}
```

3. Add the instantiation for an `UpcallHandler` object to the `AceManager` subclass constructor. To do this, replace:

```
//       UpcallHandler instantiation
```

with the following:

```
try {
#ifdef WIN32
   showPacketCountUpcallHandler =
        new UpcallHandler (appl,
                           acegroup,
                           this,
                          "showPacketCount",
                           (UpcallFp) showPacketCount);
#else
   showPacketCountUpcallHandler =
        new UpcallHandler (appl,
                           acegroup,
                           this,
                          "showPacketCount",
                           (UpcallFp)&showPacketCount);
#endif
}
 catch (NBError&) {
   throw NBError (NB_ERROR
           (NBERROR_COUNT_CANNOTCREATEUPCALL));
 }
```

This does all of the following required tasks:

- Associates the upcall handler with the ACE manager for the ACE associated with the upcall in the accelerator module.
- Specifies the relationships that an ACE manager is part of an ACE group, which is part of an `NBAppl`.
- Specifies the dictionary name that you used for the upcall in the accelerator module, in this case, `showPacketCount`.
- Specifies the name of the function that handles the message, in this case, also `showPacketCount` for consistency but the name does not need to match the dictionary name.
- Checks for exceptions.

4. In the destructor, replace:

```
//       delete UpcallHandler object
```

with the following:

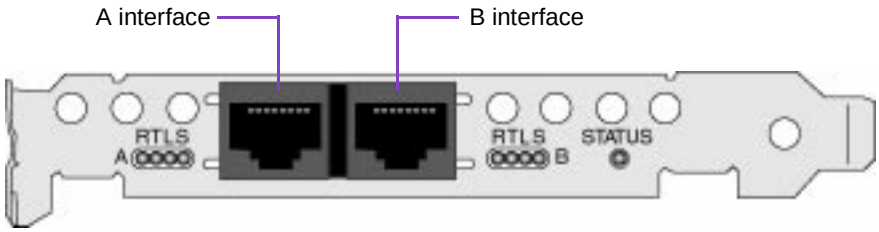
```
delete showPacketCountUpcallHandler;
```

Defining Packet Flow

The flow of packets through a Policy Accelerator is determined partly by the physical connections but primarily by actions that your application takes to specify how to handle packets moving through the physical connections.

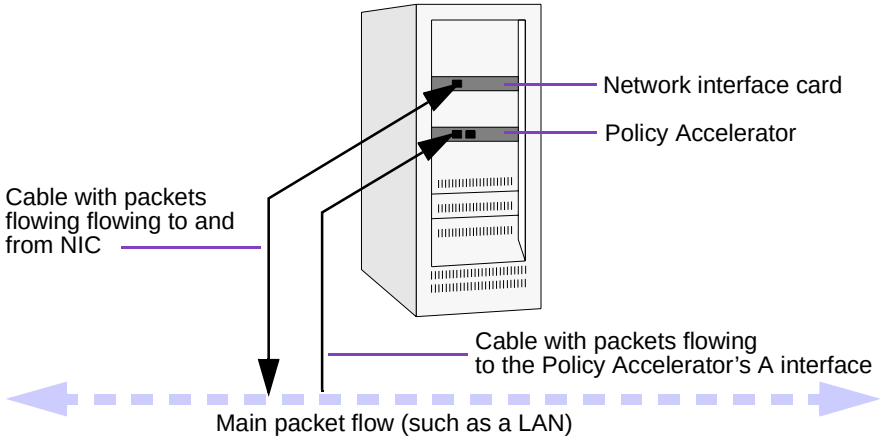
Physical Packet Flow

Each Policy Accelerator has two physical interfaces, named A and B, as shown in the following diagram:



By default, Policy Accelerators work in promiscuous mode; that is, when a cable is connected, all packets flow through the cable to the Policy Accelerator. It is your application that determines what happens to the packets after they reach the Policy Accelerator, if anything.

This sample application counts packets and does nothing else. In this case, the Policy Accelerator "sniffs" the packets flowing across the main network. That is, the Policy Accelerator receives copies of all packets, counts them, and drops them. For this purpose, your cabling might look as follows:



For more information on different ways to physically connect the Policy Accelerator interfaces, see "Defining Packet Flow" on page 75.

Logical Packet Flow

The flow of packets through Policy Accelerators is determined by logical connections among interfaces and applications, called *bindings*. The API provides the tools for you to bind interfaces and applications together. Packets flow through a Policy Accelerator only after you have created bindings.

ACEs are the entities that receive and dispose of packets. In a simple application such as the sample in this tutorial, packets arrive at the application ACE from a network interface, represented by a default system ACE named `nbhwpe0A`.

The system ACE has a FROM part and a TO part, for packets received from and passed to the interface. This application only receives packets from the interface; it does not send them back out. Therefore, it only binds the FROM part of the system ACE to the application ACE.

NOTE: For more information on `nbhwpe0` and other interface names, see "Naming Objects for the Resolver" on page 55.

Binding the Interfaces

For this sample, bind the FROM part of the A interface to your `CountAce` subclass to receive incoming packets. To do this:

1. In the *host module file*, `CountApp.cpp`, in the `NBAppl` subclass definition, replace:

   ```
   //       Network interface bindings
   ```

   with the following:

   ```
   uint32 rval;
   rval = bind
      ("/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass",
      "/CountPackets/CountAceGroup/CountAce");
   if (rval != NB_SUCCESS) {
      NB_ABORT(rval);
   }
   ```

As soon as bindings take place, packets flow through the Policy Accelerator under the control of your application as described in the following section.

Classifying and Acting on Packets

The NCL classification rules in your accelerator module determine what happens to packets flowing through your bindings. Your application does not need to explicitly look for or read packets.

About Rules

A rule has a name and has two parts:

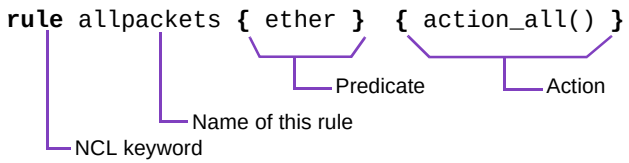
- A *predicate*

  This is a Boolean expression that describes the conditions a packet must meet to have the specified action performed on it.

- An *action*

  This is the name of a function in your action code to be run when the predicate is true.

The following figure shows a rule named `allpackets`, which runs a function named `action_all` if an incoming packet is an Ethernet packet:



About Action Functions

An action function must be of type ACTNF and must take at least two arguments, which the SDK automatically passes from the rule that calls the action function:

- A buffer containing the current packet
- The ACE that contains the NCL

You can define and pass additional arguments; this sample does not do so.

The function must also return a predefined constant that specifies the disposition of the packet after its processing is complete. One such constant, RULE_CONT, indicates that you have not modified the packet and you want processing to continue sequentially, if there is any more processing, before sending the packet along its bound path.

For further descriptions of RULE_CONT and other constants, see Chapter 7, "Acting on Packets in Your Action Code."

The following is an example of an outline of an action function:

```
ACTNF action_all (Buffer* buf, SimpleAce* ace)
{
   // operations go here
   return RULE_CONT;
}
```

Adding a Rule and an Action

In this sample, your application will run the showPacketCount method in the Ace subclass for all packets. To do this:

1. In the *NCL rules file*, CountRules.ncl, add a rule whose predicate is true for all packets and that will call a function named action_all. To do this, replace:

   ```
   // rule to invoke action for all packets
   ```

   with the following:

   ```
   rule all_packets { 1 } { action_all() }
   ```

2. In the *actions file*, CountActions.cpp, add an action function that runs the showPacketCount method of your Ace subclass. To do this, replace:

   ```
   // action function entry point (sends packet count)
   ```

   with the following:

   ```
   ACTNF action_all (Buffer* buf, CountAce* ace)
   {
      ace->showPacketCount ();
      return RULE_CONT;
   ```

```
}
```

Compiling, Linking, and Running the Application

This section describes:

- Verifying Your Development Environment
- Compilation Model
- Running the Application
- Viewing Accelerator Module Output

Verifying Your Development Environment

The *Resolver* is the resource manager for the Policy Accelerator and related objects. It must be running before you can run your applications.

The related document *Installing the IX-API SDK* contains detailed information on installing and testing the SDK on your system. This section provides only a summary of this information.

Prerequisites

Before you can compile and run applications, verify the following on your system:

- The SDK is operational as described in the following section
- You are using compatible compilation tools:
  - On a Windows NT system, use Microsoft® Developer Studio *with MFC support*
  - On a UNIX system, use any tools that are compatible with the supported compilers
- A Web browser is available for viewing online documentation

Verifying the Operation of the SDK under Windows NT

To verify that the SDK is installed correctly:

1. Verify that the Resolver icon  appears in the lower right-hand corner of your screen.

   This indicates that the driver software is installed and is operational.

   If the icon *does not* appear, restart your system. If it still does not appear, you might need to reinstall the SDK. Refer to the document *Installing the IX-API SDK*.

2. Verify whether the Resolver is running.

   The Resolver must be running before you can run or debug an application.

   The Resolver is *not* running if either of the following is true:

   - The icon is disabled
   - Placing your mouse cursor over the icon displays the pop-up text "Resolver is NOT running"

3. If the Resolver is not running, start it. To do this, right-click the icon and choose Start Resolver from its pop-up menu.

   - When the Resolver starts, the icon becomes colored and placing your mouse cursor over the icon displays "Resolver is running."
   - If the Resolver still does not start, check your log file. To do this, right-click the Resolver icon and choose View Log from its pop-up menu.

NOTE: You can also use the `resolver` command to start the Resolver, as described in the *IX-API SDK Reference.*

4. You can confirm that all of the compilers and related run-time libraries are installed and running by doing the following:

   a. Change directory to

      `SDKinstallpath``/demos/BasicApp`

   b. Run `nmake`.

      This runs a local makefile that compiles and links sample source code.

Verifying the Operation of the SDK under UNIX

To verify that the SDK is installed correctly in a UNIX environment:

1. Verify whether the Resolver is running. In a command shell, look for the `resolver` process. For example:

   ```
   # ps -auxw | grep resolver
   ```

NOTE: Use the syntax appropriate to your operating system and shell environment.

The Resolver must be running before you can run or debug an application. When you install the IX SDK, you normally set the user profile of all SDK developers to set the IX SDK envrionment variables and start the Resolver automatically on startup.

2. If the Resolver is not running, start it in a command shell. Set the SDK environment variables, then issue the `resolver` command:

```
# cd $NBPATH
# . setnbenv
# cd $NBPATH/bin
# ./resolver &
```

3. You can confirm that all of the compilers and related run-time libraries are installed and running by doing the following:

```
# cd $NBPATH/demos/BasicApp
# ./nmake
```

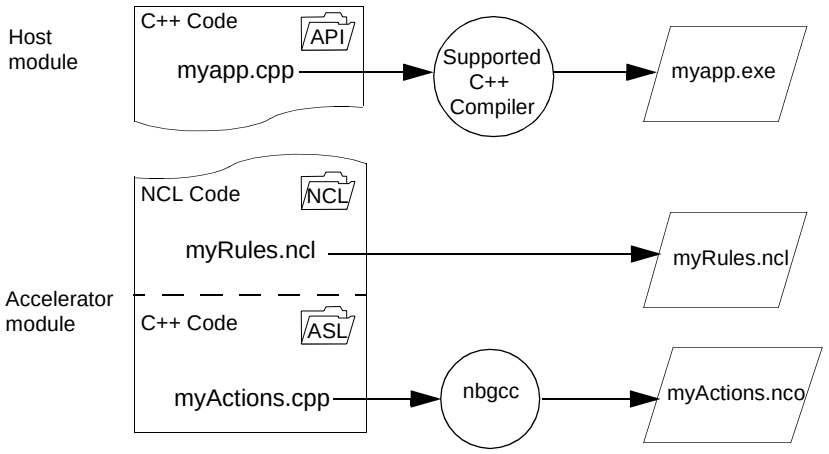
This runs a local makefile that compiles and links sample source code.

Compilation Model

To create an executable application:

- Compile and link your host module using a supported C++ compiler and linker.
- Compile your actions using the GCC compiler included with the SDK as shown in the next section.
- You do not need to compile your NCL code; this code is downloaded as source code to the Policy Accelerator and is compiled there at run time. The SDK includes an NCL compiler for use in certain circumstances; this is described in Chapter 4, “Compiling Applications.”

NOTE: Typically, you use a makefile to perform these steps. However, this tutorial takes you through each step so that you can see how the pieces work, as shown in the following figure:



Compiling the Host Module

1. Compile and link your host module using any supported compiler.

   In this sample, this creates an executable file, `CountApp.exe`.

Compiling the Action Code

These instructions compile your actions with debugging turned off and with the highest level of optimization to make the code most efficient. To do this:

1. Use the following command to compile your actions file, replacing *`SDKinstallpath`* with your installation path:

   ```
   nbgcc -c -O2 -D_BIGENDIAN -DNDEBUG \
      -ISDKinstallpath/include CountActions.cpp
   ```

2. Rename the generated `CountActions.o` file to `CountActions.nbo`.

Running the Application

To run the application:

1. Set up your Policy Accelerator so that packets are directed to and from the interfaces.

   For this example, connect the Policy Accelerator's interface A to a packet source.

2. In any shell window, run `CountApp.exe`.

   This downloads the accelerator module to the Policy Accelerator and begins counting packets.

Viewing Accelerator Module Output

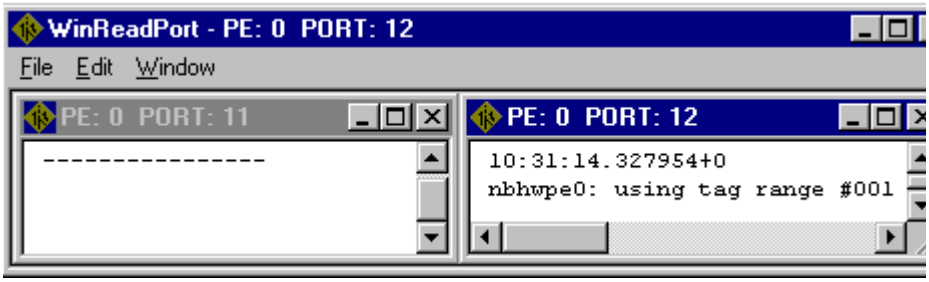
The Resolver can display output from the accelerator module of your application in windows, one for Policy Accelerator system output (`sysout`) and one for standard output (the Policy Accelerator's `stdout`). The way you access this output depends on your operating system.

Viewing Output in Windows NT

To view accelerator module output:

1. Right-click on the Resolver icon in the lower right-hand corner of your screen.
2. Choose WinReadPort from its pop-up menu.

   This displays a window similar to that shown in the following figure; PORT:11 displays `stdout` and PORT:12 displays `sysout`:



Viewing Output in UNIX

To view accelerator module output:

1. Open a command shell and execute these commands:

   ```
   cd $NBPATH/bin
   ./readport 11
   ```

   This shell displays output that is written to `stdout` by the accelerator module of your application.
2. Open a second command shell and execute these commands:

   ```
   cd $NBPATH/bin
   ./readport 12
   ```

   This shell displays errors and warnings from the Policy Accelerator that are written to `sysout`.

If You Have Problems

If you have problems, check your PATH environment variable. For Windows NT, it should include all Microsoft Visual Studio bins and libs, and nmake. To do this in Windows NT, use the Properties:Environment command from the MyComputer menu or use a command prompt.

Consider using the vcvars32.bat batchfile from Microsoft Visual Studio's bin directory, which sets your PATH and does other setup for each DOS session.

Chapter 3
Elements of an Application

The IX SDK defines an object-oriented environment that you use to develop applications. Your application creates objects and calls their methods to perform the policy enforcement functionality that you define.

This chapter gives an overview of the class and object framework, introduces the IX system software that coordinates objects on the host and in Policy Accelerator memory, and describes the error handling system.

This chapter contains the following topics:

- The Object Framework
- The Resolver and Multiple Applications
- Return Values and Error Codes

The Object Framework

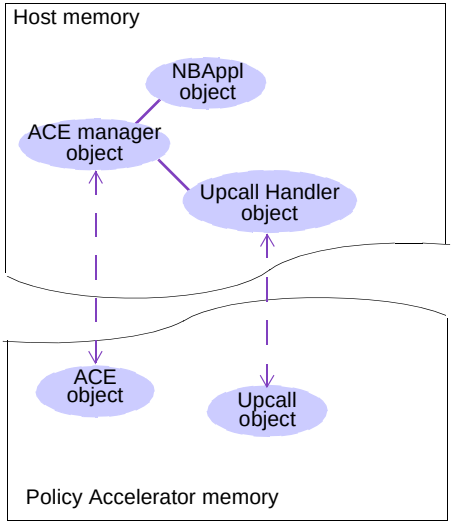
Every application that uses the IX-API SDK must have the basic framework of objects to define at least one ACE. This includes:

- An application object in the host module
- An ACE group and ACE manager object in the host module
- An ACE object in the accelerator module

An application creates additional objects to support the particular features it uses.

An application usually defines its own subclasses of the basic host API and ASL classes that it needs. This allows the application to add to the constructors whichever details that it needs. For example, if the application uses calls and message passing, the application-specific subclass of the Ace class would contain the methods for sending upcalls and crosscalls and the methods for handling crosscalls and downcalls. Its constructor might also create the call objects for the accelerator module side of the ACE.

Many objects are paired—that is, objects that are defined in the host module and reside in memory on the host have counterparts defined in the accelerator module, which reside in memory on the Policy Accelerator. The *Resolver* coordinates the host and Policy Accelerator memory spaces; see “The Resolver and Multiple Applications” on page 54.



If the application needs such an object on one side, it must also create its manager or handler on the other side. You associate the paired objects by giving them the same *dictionary name*—that is, by specifying the same string value for the *name* argument when you construct the two objects. (For more information about naming, see “Naming Objects for the Resolver” on page 55.)

This section provides a basic overview of what subclasses and objects an application must or can have, and their required relationships.

ACE Framework Objects

The following table shows the classes of objects that an application must or can create in the host module and accelerator module to support the basic framework of the application ACEs. You usually define application-specific subclasses of these types.

Host module API class	Accelerator module ASL class	Description
`NBAppl`		Required. Create one object for an application.
`AceGroup`		Required. Create at least one object for an application.
`AceManager`	`Ace`	Required. Create one pair of objects for each ACE block. You must have at least one ACE block.
`TargetManager`	`Target`	Create these object pairs if you are defining your own targets. Your own targets are named destinations for packets other than the default pass and drop targets. See “Defining Targets” on page 81.

Message-Sending Framework Objects

The following table shows the classes of objects that an application can create in the host module and accelerator module to support message sending using upcalls, downcalls, or crosscalls. You usually create application-specific subclasses of the call classes. You create message objects directly, without defining subclasses.

Host module API class	Accelerator module ASL class	Description
`Downcall`	`DowncallHandler`	Create one pair of objects for each kind of downcall the host will send to an ACE.
`UpcallHandler`	`Upcall`	Create one pair of objects for each kind of upcall an ACE will send to the host.

Host module API class	Accelerator module ASL class	Description
`CrosscallManager`	`Crosscall`	Create one pair of objects for each kind of crosscall an ACE will *send* to another ACE. Use the `NBAppl` object's `link` method to associate the pair with a crosscall handler.
`CrosscallHandlerManager`	`CrosscallHandler`	Create one pair of objects for each kind of crosscall an ACE will *receive* from another ACE. Use the `NBAppl` object's `link` method to associate the pair with a crosscall sender.
`Message`, `MessageBlock`		Create message objects on the host to send to the Policy Accelerator in downcalls. You do not need to create subclasses of these types.
	`Message`, `MessageBlock`	Create message objects on the Policy Accelerator to send to the host in upcalls, or to send to other ACEs in crosscalls. You do not need to create subclasses of these types.

For more information on calls and message passing, see Chapter 8, "Communication Within an Application."

String Search Framework Objects

The following table shows the classes that support the string search facility, which you use to find strings in the data portion of packets. These are all on the accelerator module side.

Accelerator module ASL class	Description
`NBSearchContext`	Create an object of this type to keep track of a specific string search, which can cross packet buffer boundaries.

Accelerator module ASL class	Description
`NBStringSearchEngine`	Create an object of this type to send a specific packet buffer to a specific search, as defined by a context object. This object also keeps a list of strings for which to find matches.
`NBStringMatchReport`	Create an object of this type to hold reports on matching strings found by a string search.

For more information on string searches, see Chapter 10, “Finding Strings in Packets.”

Data Set Framework Objects

The following table shows the classes that support data sets and their associated searches. These classes are all on the accelerator module side. You do not create set subclasses directly; instead, you define the sets and searches in NCL, then use the NCL compiler to automatically generate a header file containing the set subclass definitions. You then include the header in your action code file, where you can extend the element subclasses as needed.

Accelerator module ASL subclass	Description
`Set_`*`setname`*	The generated header file contains one of these subclasses for each defined set. You do not need to create a further subclass. Your ACE object must contain an object of this subclass, using the same name that you use when defining it in NCL.
`Elt_`*`setname`*	The generated header file contains one of these subclasses for each defined set. You define a further subclass to extend the definition of a set element to contain your application-specific data. Create objects of this type with the `new` operator, to add to the set using the `insert` method of a search.
`setname.searchname`	The generated header file contains one of these subclasses for each defined search on a set. You do not need to create a further subclass or object.

For more information on generating a header file from an NCL file, see Chapter 4, “Compiling Applications.” For more information on defining and using sets and searches, see Chapter 9, “Using Sets of Data to Classify Packets.”

Auxiliary Objects

The following table shows additional ASL classes that you might use in your application for specific purposes, such as keeping statistical data about your traffic flow.

Accelerator module ASL class	Description
`Event` `Time`	Create a subclass of `Event` to define an event callback that is triggered after a certain amount of time, as defined by a `Time` object. This allows you to schedule future actions.
`Rate`	Create an object with a specific sampling period to track event rates and bandwidths so you can watch for rates that exceed desired values. Use `Time` objects to specify sampling periods.
`NBRmon`	The application automatically creates an object of this type that provides access to RMON block counters for each of the two MAC interfaces on the Policy Accelerator board.
`NBInterfaceProp` `NBLinkwatch`	These allow you to manage properties of the MAC interfaces and monitor the network connection.

For more information on using these classes, see the *IX-API SDK Reference.*

The Resolver and Multiple Applications

The Resolver is an independent process that is normally always running in the background on the host. The Resolver coordinates the host and Policy Accelerator memory spaces for multiple IX applications.

When you install the IX 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:

- You can 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. See Chapter 7, “Command-Line Tools,” in the *IX-API SDK Reference.*
- You can 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.”

- On a Windows NT system, you can 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.”
  - 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.”

Starting and Stopping 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.

Naming Objects for the Resolver

ACEs, targets, and crosscalls are logical entities that reside partly in the host’s application space (in manager objects) and partly in Policy Accelerator memory (in managed objects). The Resolver coordinates the two memory spaces.

The Resolver keeps track of these distributed logical entities using the *dictionary name*—the string value of the *name* argument that you provide when you construct the associated objects. The dictionary names of paired objects are the same, and that name identifies the logical entity.

For example, suppose you create a target manager object in the host module (in the context of an existing application and ACE group) using the following code:

```
targetMgr1 = new MyTargetMgr (appl, this,
                                  myAceMgr1, "Target1Name");
```

This binds the variable `targetMgr1` to the object handle, and defines the string “`Target1Name`” as the target’s dictionary name. The target belongs to the ACE whose manager object has the handle `myAceMgr1`.

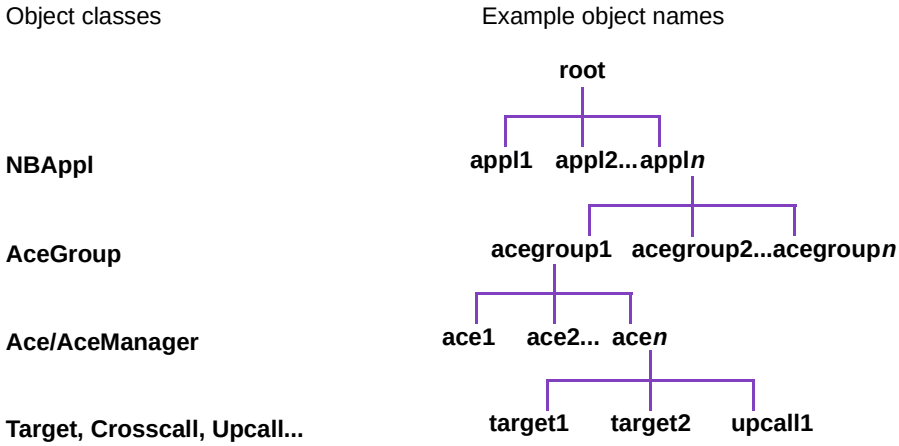
In the accelerator module, you define the corresponding target object in the context of the ACE object, using the same dictionary name, as follows:

```
target1 = new MyTarget (id, this, "Target1Name");
```

This binds the variable `target1` to the object handle of the new `Target` object. The new object is paired with the `TargetManager` object with the same dictionary name. `TargetName` refers to the target entity, which comprises both objects.

Full Name Paths

The name of a target (for example) needs to be unique only within the ACE, and the name of the ACE needs to be unique only within the ACE group and application. To completely and uniquely identify an entity to the Resolver, you must use the *full name*, or complete path to the entity, which includes the containing application, ACE group, and ACE. Full names are derived from the hierarchy of named objects as shown in the following figure:



You use the full name:

- 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)

The full name of an target or crosscall has the following format:

`/appname/acegroupname/acename/objectname`

Each of these elements is the dictionary name of the respective entity, not an object handle.

For example, suppose you define and create application, ACE group, ACE manager, and target manager objects in the host module, using code such as the following:

```
NBMyAppl::NBMyAppl (void):
   NBAppl ("MyAppName", "myapp.exe")
...
//within the context of the application object
myGroup = new NBMyGroup (appl, "MyGroupName");
//within the context of the ACE group object
myAceMgr = new NBMyAceMgr (appl, this, "MyAceName");
//within the context of the ACE manager object
target1 = new NBMyTgtMgr (id, ace, "MyTgtName");
```

To create a binding between the Policy Accelerator interface A and this target (associated wtih the target manager object whose handle is `target1`), use the following full name in the `bind` method:

```
bind ("/nbhwpe0/FromInterface:nbhwpe0A/Interface",
      "/MyAppName/MyGroup/MyAceName/MyTgtName");
```

System ACE Names

Notice that the reference to the Policy Accelerator interface uses a predefined system ACE name. These use the same format, but with predefined names that identify the specific Policy Accelerator board (in place of the application) and source or destination (in place of the ACE group).

The following ACEs are defined by the system to allow access to the Policy Accelerator interfaces and to the host protocol stack :

To interface A	`/nbhwpe0/ToInterface:nbhwpe0A/Interface`
From interface A	`/nbhwpe0/FromInterface:nbhwpe0A/Interface`
To interface B	`/nbhwpe0/ToInterface:nbhwpe0A/Interface`
From interface B	`/nbhwpe0/FromInterface:nbhwpe0B/Interface`
To host stack bound to interface A	`/nbhwpe0/ToStack:nbhwpe0A/Stack`
From host stack bound to interface A	`/nbhwpe0/FromStack:nbhwpe0A/Stack`

To host stack bound to interface B	`/nbhwpe0/ToStack:nbhwpe0B/Stack`
From host stack bound to interface B	`/nbhwpe0/FromStack:nbhwpe0B/Stack`

NOTE: The name `nbhwpe0` refers to the first Policy Accelerator installed in a system. If you install additional cards, their names are `nbhwpe1`, `nbhwpe2`, and so on.

There are also system-defined targets named `pass` and `drop` in each of the system ACEs. For example:

```
/nbhwpe0/ToInterface:nbhwpe0A/Interface/pass
```

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, using the name `nbhwpe`*n*`C` (where *n* is the number identifying the card). For more information, see *Customizing a NIC Driver Using the ODX Protocol*.

For More Information

- For more information on the names of system ACEs and targets, see Appendix C, “Policy Accelerator Name Space,” in the *IX-API SDK Reference*.
- For more information on bindings, see Chapter 5, “Controlling Packet Flow.”
- For more information on crosscalls, see Chapter 8, “Communication Within an Application.”
- For more information on the `nbgdb` debugger, see Chapter 11, “Debugging and Troubleshooting.”

Return Values and Error Codes

Return values and error codes are handled differently in the ASL and in the host API.

- In the ASL, negative return values indicate an error. Use the IX-API SDK debugger (`nbgdb`) to evaluate the problem. See Chapter 11, “Debugging and Troubleshooting.”
- Host API functions that fail return `NULL` or an error code of type `NBError`. Object constructors throw an exception of type `NBError`.

The following class represents errors in the host module:

Host module API class	Description
`NBError`	An object of this type is returned by host API methods on failure, or is thrown by object constructors. Use the `getErrorcode` method to access the descriptive code explaining why a method failed. The predefined error codes, defined in `NBError.h`, are listed and described in the *IX-API SDK Reference.*

For a host API object constructor, which throws the error rather than returning it, use a `catch` statement to access the error object. For example, the following code fragment uses the `getErrorcode` method in a `catch` routine to print out a debugging message:

```
catch (NBError E) {
   fprintf(stderr,
           "Demo app caught NBError 0x%X\n",
           E.getErrorcode ());
   NB_ABORT (1);
}
```

You can add your own uniquely numbered error codes to provide information about failed operations in the host API. Give your own error codes numbers greater than the constant `NBERROR_USER_BASE`. For an example of how to define your own error codes, see “Defining Error Codes” on page 19.

Chapter 4
Compiling Applications

Compilation tools, such as makefiles or Microsoft Visual Studio*, are the most effective way to ensure that all of an application's files are compiled and linked in the correct manner and order. This chapter describes each of the compilation steps in detail and then provides a sample set of makefiles. It contains the following topics:

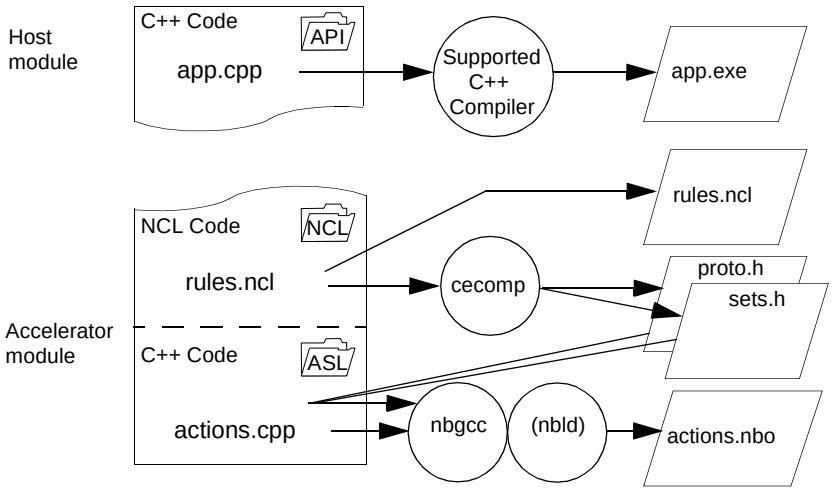
- Overview of the Compilation and Linking Process
- Compiling the Host Module
- Compiling NCL Files
- Compiling Action Code
- Code Development on a Windows NT System
- Using Makefiles for a UNIX System
- Running an Application
- Debugging an Application

Overview of the Compilation and Linking Process

There are three parts of an application that can be compiled and linked:

- Host module source code: You compile and link this with a supported compiler to create an executable file.
- Action code for the Policy Accelerator: You compile this with a modified version of the GNU C++ Compiler (`nbgcc`) provided with the IX-API SDK. You link it only if you have more than one action code file for a specific ACE.
- Network Classification Language (NCL) code: This is compiled by the Policy Accelerator at runtime, but you can use the NCL compiler (`cecomp`) directly to generate header files that synchronize the NCL with the action code, or to verify the syntax.

The following figure shows the general flow of application files through the compilation process.



Code Development on a Windows NT System

If your application is not already a Microsoft* Visual C++ 6.0 project, you must create such a project. You must specify build and compilation information in the project, rather than in an exported makefile.

To obtain the proper project settings, refer to the project files for the sample applications in *SDKinstallpath*/demos. Note the settings in the following sections of the Project Setting dialog box:

- C/C++ and Link tabs, Project Options
- Custom Build tab, Commands

The options and commands are slightly different for the release and debug versions of the applications. You must replace filenames and paths as appropriate for your application.

NOTE: To configure Microsoft Visual C++ 6.0 for use with the IX-API SDK, choose Options from the Tools menu, then, in the Options dialog box Directories tab, add the following pathname:

SDKInstallPath/include

You need to set this configuration only once. It is not necessary to do so for each project.

Compiling the Host Module

Compile and link your host module to create an executable file using a supported C++ compiler. For the latest information on supported compilers, refer to the document *Installing the IX-API SDK* or to the *IX-API SDK Release Notes*. Currently, use the following:

- Windows NT: Microsoft* Visual C++ 6.0
- UNIX: BSD / OS

Compiling With Static Libraries for Windows NT

Under Windows NT, you can compile the host module using either the dynamic libraries for the host API, which is the default, or the static libraries. You might use the static libraries to protect your executable code from changes that might occur in dynamic libraries due to such changes as the following:

- An update or new release of the libraries
- An attempt to bypass security by modifying the libraries

To compile the host module using the static libraries:

1. Use a static versions of the IX libraries (which begin with `nb`). For example, in the Link Project Options, replace `nbapid.lib` with `nbapistatic.lib` for nondebug mode, or with `nbapistaticd.lib` for debug mode.

2. In the compilation command-line options (C / C++ Project Options):
   - Use `/MT` instead of `/MD` or `/MDd`.
   - Do *not* use `/D _AFXDLL`.
   - Use `/D _NBAPI_STATIC`.

Compiling NCL Files

Classification files for the accelerator module contain rules written in NCL. Your application loads the source NCL file directly into the Policy Accelerator, where it is compiled at run time, so in most cases you do not need to explicitly compile your NCL code.

NOTE: For an application that is to run in an IX environment customized for an unsupported host using the IX SDK-E, you must specifically compile the NCL code using `cecomp` and make the object file available to the load function.

You can use the NCL compiler (`cecomp`) as a stand-alone tool for the following reasons:

- Check for NCL compile-time errors
- Generate protocol and set header files for inclusion in your action files

Checking f or NCL Errors

To check for NCL compile-time errors, or to find the size of the NCL object, pass the name of the NCL file to the NCL compiler (`cecomp`), and then pass the resulting object file to the NCL linker (`celink`). For example:

```
cecomp myNCL.ncl
celink -o myNCL.exe SDKinstallpath\lib\mrt0.o myNCL.o
```

This creates the relocatable, executable image `myNCL.exe`.

Generating Headers for Action Files

Each NCL file is associated with exactly one action file. The action file must use certain information that is contained in the NCL file:

- **Sets and Searches:** If you have defined sets and searches in the NCL file, they 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.

  For more information on sets and searches, see Chapter 9, “Using Sets of Data to Classify Packets.”

- **Field Accessors:** It is most efficient for the action code to have its own access methods for protocol fields, so that rules do not need to pass every field value that an action might need. The field information needed to create the accessors is found in the protocol definition in the NCL file. Therefore, you use the NCL compiler to generate a C++ header file for the action code, which defines field accessors for each protocol defined in the NCL file.

Before you compile the action code so that your application can load it into the Policy Accelerator, you must make sure that it includes these generated header files to synchronize it with the NCL file.

To generate the action header files and include them in the action code:

1. Compile the NCL file, using the NCL compiler (`cecomp`) with the appropriate command-line options:
   - The `-Fs` option generates base classes and objects for each of the sets and named searches in the NCL file and defines a data structure for each set.

- The `-Fa` option 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 and Inter-Module Communication" in Chapter 2, "System Types and Methods," in the *IX-API SDK Reference*.

For example, to generate both header files from the source file `myNCL.ncl`:

```
cecomp -Fa proto.h -Fs sets.h myNCL.ncl
```

2. Move or copy the resulting C++ source header files (in this example, `proto.h` and `sets.h`) to the action code's standard include directory.
3. Use `#include` to include the header files in your action code.

NOTE: Do not modify the set header file created by the NCL compiler, because any changes you make to it would be overwritten the next time you generate it. To extend the set element definitions, modify the main action source file.

Compiling Action Code

Before your application can load the action code into the Policy Accelerator, you must compile the source code using the modified GNU C++ compiler included with the IX-API SDK, `nbgcc`. The source code should include the header files generated from the NCL part of the same ACE.

To compile the action code:

1. Ensure that the code includes the following header files:
   - Generated from the corresponding NCL file (`proto.h` and `sets.h` in the preceding example)
   - ASL classes used in your application from the *`SDKinstallpath`*`/include` directory (replace *`SDKinstallpath`* with your IX-API SDK installation path)
2. Compile using the `nbgcc` compiler. For example:

   ```
   nbgcc -c -03 -D_BIGENDIAN -ISDKinstallpath\include
      myActions.cpp
   ```

   This creates a file named `myActions.o`.
3. If you have only one action code file for an ACE, proceed to the next step. If you have more than one action code file for a single ACE, compile them all and then link them as in this example:

   ```
   nbld -r -EB myActions1.o myActions2.o
   ```

4. Rename the resulting object file (by default `a.out`) to *`filename`*`.nbo`.

   For example, rename `a.out` to `myActions.nbo`.

For more information on options available during compilation, see:

- Chapter 7, "Command-Line Tools," in the *IX-API SDK Reference*
- GNU C++ compiler documentation, available in the `\usr\local\docs` directory

Using Makefiles for a UNIX System

A makefile is the most effective way to ensure that all of an application's files are compiled and linked in the correct manner and order.

You can issue compilation and linking commands directly from the command line in a command shell, but you typically use a makefile to compile and link your application. Makefiles are designed to handle dependencies in compilation order, such as generating protocol and set headers from your NCL for inclusion by your action code.

The following example makefiles, based on the sample application `BasicApp` (in *`SDKinstallpath`*`/demos`), show how you assemble all of the files needed for an application. In this sample, the main makefile includes a definition file and an installation file, which are defined separately.

```
######################################################################
# Checking for must-have environment variables
##   NBPATH: Required. Directory where IX-API SDK installed.
##   SRCDIR: Optional. Location of source files for
##             host module, accelerator action code, and NCL;
##             sets to current directory if undefined.
##   BUILD_MODE:   Optional. "debug" or "release" (default).
##             Sets name of directory for generated output and
##             determines compiler flags. Use as in:
##                 nmake BUILD_MODE=debug, or:
##                 make BUILD_MODE=debug
######################################################################
######################################################################

ifndef SRCDIR
   SRCDIR=.
endif

ifndef BUILD_MODE
   BUILD_MODE= release
endif
```

```
#################################################################
# Determining the OS. Valid OSTYPEs are:
#     bsdi & cygnus32
# if OSTYPE is not defined then set it to winnt (DOS Shell)
#
# PLATFORM sets name of directory for generated output.
#################################################################

ifeq ($(OS), Windows_NT)
   PLATFORM= $(OSTYPE)
   ifndef $(OSTYPE)
      PLATFORM= winnt
   endif
else
   ifeq ($(OSTYPE),bsdi4.0)
      OSTYPE= bsdi
   endif
   PLATFORM= $(OSTYPE)
endif

#########################################################
##  Output directory, BINDIR, for all generated output
#########################################################

BINDIR= $(PLATFORM)/$(BUILD_MODE)

#########################################################
## In addition to the preceding, this makefile uses
## the following variables set externally
## or on the command line:
##   HOSTMOD:Required. Name of host module C++ source file.
##   PEMOD:  Required. Name of accelerator module C++ action
code.
##   NCLRULE: Required. Name of NCL file.
##   SETFILE: Optional. Base filename (e.g., myfile)
##            under SRCDIR into which the NCL compiler
##            should generate a set header (e.g., myfile.h)
##            and a static data header (e.g., myfile_def.h).
##            If not defined, these files are not
##            generated.
##   ACCSFILE:Optional. Filename under SRCDIR into which
##            the NCL compiler should generate a header
##            containing a class for each defined protocol, with
##            accessor methods for each field. If not defined,
##            no protocol header is generated.
#########################################################
##   Example:
##       nmake HOSTMOD=myhost.cpp PEMOD=mype.cpp
```

```
##             NCLRULE=myrules.ncl SETFILE=mysets
############################################################
## Note: This makefile supports up to 4 pairs of
##       action code/NCL; additional sets n=1,2,...:
##  PEMODn
##  NCLRULEn
##  SETFILEn
##  ACCSFILEn
##
############################################################
##
##   Command and command-flag aliases:
##      NBLIB:  location of IX-API SDK libraries
##      NBGCC:  The IX-API SDK compiler for accelerator module
C++ code
##      CECOMP: IX-API SDK NCL compiler
##      CECOMP_FLAGS:  NCL compiler flags; -w=off turns off
##             compiler warnings
##
############################################################
NBLIB= $(NBPATH)/lib
NBGCC= nbgcc
LD= nbld
RM= rm -rf
CP= cp
CECOMP= cecomp
CECOMP_FLAGS= -w=off
MKDIR= mkdir -p


############################################################
##   General compilation settings.
##       CPP:        Host compiler.
##       CPP_FLAGS:  Host compiler flags.
##       EXE_EXT:    Host executable extension:
##                      .exe for WinNT, nothing otherwise
##       LINK_OPT:   Host linker options.
##       OBJ_EXT:    Host object code extension.
#        LINK:
##       LINKFLAGS:
##       NB_INCLUDE: IX-API SDK include-file directories
##       NBGCC_FLAGS: Compiler flags for accelerator module
compiler.
##       NB_LIBS: IX-API SDK libraries for host module.
##
############################################################
##
##  Specific Compiler Flags
##
```

```
## For host module: Choose the ones
## from your host compiler that you need.
## This sample makefile uses the following.
##
##    For all compilation:
##      -D_ENDIAN_H_  : Byte-ordering specifier
##
##    For Debug mode:
##      -g            : Generates symbol table
##                      for use with debugger.
##
## For accelerator module: Choose the ones from nbgcc that you
need,
## and use required ones. This example uses the following.
##
##    For all compilation:
##      -c            : Required; compile only, don't link.
##      -D_BIGENDIAN  : Required byte-ordering specifier.
##    For Debug mode:
##      -o0           : No optimization.
##      -g            : Required to generate symbol table
##                      for use with nbgdb.
##    For nondebug (Release) mode:
##      -o2           : Highest level of optimization
############################################################
CPP= gcc
EXE_EXT=
LINK_OPT= -o $@
OBJ_EXT= .o
LINK=
LINKFLAGS=

NB_INCLUDE= -I$(NBPATH)/include -I$(NBPATH)/include/nbossl

ifeq ($(BUILD_MODE),debug)

   CPP_FLAGS= -g -D_ENDIAN_H_
   NBGCC_FLAGS= -c -g -D_BIGENDIAN -DNDEBUG

   NB_LIBS= \
          $(NBLIB)/nbapi.a \
          $(NBLIB)/nbrif.a \
          $(NBLIB)/nberror.a \
          $(NBLIB)/libnbossl.a \
          $(NBLIB)/nbdif.a
else

   CPP_FLAGS= -D_ENDIAN_H_
   NBGCC_FLAGS= -c -O2 -D_BIGENDIAN -DNDEBUG
```

```
   NB_LIBS= \
          $(NBLIB)/nbapi.a \
          $(NBLIB)/nbrif.a \
          $(NBLIB)/nberror.a \
          $(NBLIB)/libnbossl.a \
          $(NBLIB)/nbdif.a
endif



#####################################################################
# SETFILE and PROTOCOL FIELD ACCESS METHOD settings
#####################################################
#### Compile NCL code conditionally:
####   If action code uses sets or protocols as defined in
####   NCL, then NCL must be compiled:
####     using -Ft to generate set header file
####     using -Fa to generate protocol accessor header file
#####################################################################

ifdef SETFILE
   SRCSET= $(SRCDIR)/$(SETFILE).h
   SETS= -Ft$(SRCDIR)/$(SETFILE)
ifdef ACCSFILE
   SRCACCS= $(SRCDIR)/$(ACCSFILE).h
endif
endif

ifdef SETFILE1
   SRCSET1= $(SRCDIR)/$(SETFILE1).h
   SETS1= -Ft$(SRCDIR)/$(SETFILE1)
ifdef ACCSFILE1
   SRCACCS1= $(SRCDIR)/$(ACCSFILE1).h
endif
endif

ifdef SETFILE2
   SRCSET2= $(SRCDIR)/$(SETFILE2).h
   SETS2= -Ft$(SRCDIR)/$(SETFILE2)
ifdef ACCSFILE2
   SRCACCS2= $(SRCDIR)/$(ACCSFILE2).h
endif
endif

ifdef SETFILE3
   SRCSET3= $(SRCDIR)/$(SETFILE3).h
   SETS3= -Ft$(SRCDIR)/$(SETFILE3)
ifdef ACCSFILE3
```

```
   SRCACCS3= $(SRCDIR)/$(ACCSFILE3).h
endif
endif

######################################################################
# multiple accelerator module settings
# Note: If PEMOD1 is defined, assume NCLRULE1 is also defined,
etc.
######################################################################

ifdef PEMOD1
   PEMOD_OBJ_1 = $(BINDIR)/$(PEMOD1).nbo
    NCL1 = $(BINDIR)/$(NCLRULE1).ncl
endif

ifdef PEMOD2
   PEMOD_OBJ_2 = $(BINDIR)/$(PEMOD2).nbo
    NCL2 = $(BINDIR)/$(NCLRULE2).ncl
endif

ifdef PEMOD3
   PEMOD_OBJ_3 = $(BINDIR)/$(PEMOD3).nbo
    NCL3 = $(BINDIR)/$(NCLRULE3).ncl
endif

######################################################################
# The Definitions of all targets
######################################################################
all: $(BINDIR)/$(HOSTMOD)$(EXE_EXT) \
   $(BINDIR)/$(PEMOD).nbo $(BINDIR)/$(NCLRULE).ncl \
   $(PEMOD_OBJ_1) $(NCL1) \
   $(PEMOD_OBJ_2) $(NCL2) \
   $(PEMOD_OBJ_3) $(NCL3) \

######################################################################
# Create host module executable
######################################################################
$(BINDIR)/$(HOSTMOD): $(SRCDIR)/$(HOSTMOD).cpp
   @[ -d "${BINDIR}" ] || $(MKDIR) "$(BINDIR)"
   $(CPP) $(CPP_FLAGS) $(NB_INCLUDE) $^ $(NB_LIBS) $(LINK_OPT)

$(BINDIR)/$(HOSTMOD).exe: $(SRCDIR)/$(HOSTMOD).cpp
   @[ -d "${BINDIR}" ] || $(MKDIR) "$(BINDIR)"
   $(CPP) $(CPP_FLAGS) $(NB_INCLUDE) $^ $(NB_LIBS) $(LINK_OPT)
   $(LINK) "$(NBAPILIB)" $(LINKFLAGS)
/out:$(BINDIR)/$(HOSTMOD).exe \
      $(BINDIR)/$(HOSTMOD).obj

##########################################################
```

```
##
##  Accelerator Module:
##      Action & Classification code
##
####   Compile action code (accelerator module).
####    .nbo is designation for compiled Action code.
############################################################
$(BINDIR)/$(PEMOD).nbo: $(SRCDIR)/$(PEMOD).cpp $(SRCSET)
   $(NBGCC) $(NBGCC_FLAGS) $(NB_INCLUDE) -I$(SRCSET) -c
$(SRCDIR)/$(PEMOD).cpp -o $@

############################################################
##   Compile NCL to generate Sets include file
##   Copy NCL source into output directory
############################################################
$(SRCSET): $(SRCDIR)/$(NCLRULE).ncl
   @echo building $(SETFILE).h and $(ACCSFILE).h
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fs$(SRCSET)
$(SRCDIR)/$(NCLRULE).ncl
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fa$(SRCACCS)
$(SRCDIR)/$(NCLRULE).ncl

$(BINDIR)/$(NCLRULE).ncl: $(SRCDIR)/$(NCLRULE).ncl
   $(CP) $(SRCDIR)/$(NCLRULE).ncl $(BINDIR)/$(NCLRULE).ncl

######################
# If there's more than one accelerator module, do the 2nd:
######################
$(PEMOD_OBJ_1): $(SRCDIR)/$(PEMOD1).cpp $(SRCSET1)
   $(NBGCC) $(NBGCC_FLAGS) $(NB_INCLUDE) -I$(SRCSET1) -c
$(SRCDIR)/$(PEMOD1).cpp -o $@

$(SRCSET1): $(SRCDIR)/$(NCLRULE1).ncl
   @echo building $(SETFILE1).h and $(ACCSFILE1).h
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fs$(SRCSET1)
$(SRCDIR)/$(NCLRULE1).ncl
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fa$(SRCACCS1)
$(SRCDIR)/$(NCLRULE1).ncl

$(NCL1): $(SRCDIR)/$(NCLRULE1).ncl
   $(CP) $(SRCDIR)/$(NCLRULE1).ncl $(BINDIR)/$(NCLRULE1).ncl

######################
# For the 3rd accelerator module
######################
$(PEMOD_OBJ_2): $(SRCDIR)/$(PEMOD2).cpp $(SRCSET2)
   $(NBGCC) $(NBGCC_FLAGS) $(NB_INCLUDE) -I$(SRCSET2) -c
$(SRCDIR)/$(PEMOD2).cpp -o $@
```

```
$(SRCSET2): $(SRCDIR)/$(NCLRULE2).ncl
   @echo building $(SETFILE2).h and $(ACCSFILE2).h
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fs$(SRCSET2)
$(SRCDIR)/$(NCLRULE2).ncl
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fa$(SRCACCS2)
$(SRCDIR)/$(NCLRULE2).ncl

$(NCL2): $(SRCDIR)/$(NCLRULE2).ncl
   $(CP) $(SRCDIR)/$(NCLRULE2).ncl $(BINDIR)/$(NCLRULE2).ncl

######################
# For the 4th accelerator module
######################
$(PEMOD_OBJ_3): $(SRCDIR)/$(PEMOD3).cpp $(SRCSET3)
   $(NBGCC) $(NBGCC_FLAGS) $(NB_INCLUDE) -I$(SRCSET3) -c
$(SRCDIR)/$(PEMOD3).cpp -o $@

$(SRCSET3): $(SRCDIR)/$(NCLRULE3).ncl
   @echo building $(SETFILE3).h and $(ACCSFILE3).h
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fs$(SRCSET3)
$(SRCDIR)/$(NCLRULE3).ncl
   $(CECOMP) $(CECOMP_FLAGS) $(SETS) -Fa$(SRCACCS3)
$(SRCDIR)/$(NCLRULE3).ncl

$(NCL3): $(SRCDIR)/$(NCLRULE3).ncl
   $(CP) $(SRCDIR)/$(NCLRULE3).ncl $(BINDIR)/$(NCLRULE3).ncl

clean:
   @rm -rf "$(PLATFORM)"
```

Running an Application

To run an application:

1. Ensure that the IX-API SDK Resolver is running.

   See “Verifying Your Development Environment” on page 42. The `resolver` command is also described in the *IX-API SDK Reference*.

2. Run the application’s host module executable file.

   This begins executing and loads the accelerator module into the Policy Accelerator.

Debugging an Application

For information on debugging an application, see Chapter 11, “Debugging and Troubleshooting.”

Chapter 5
Controlling Packet Flow

This chapter explains how packets flow into and out of the Policy Accelerator according to both the physical connections and the logical connections, or bindings. It describes the system and default targets to which you can direct packets, and explains how to create additional targets.

This chapter contains the following topics:

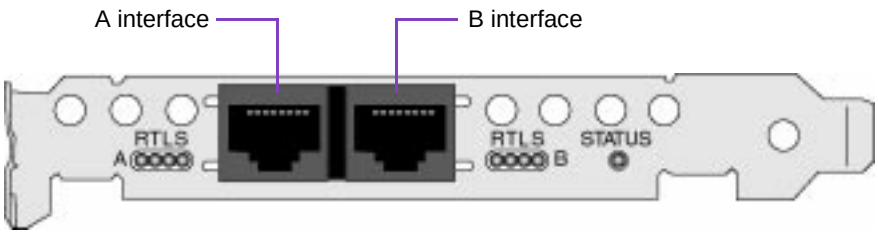
- Defining Packet Flow
- Binding Targets as Packet Destinations
- Defining Targets
- Directing Packets to a Target
- Using Targets to Serialize Packet Processing

Defining Packet Flow

The flow of packets through a Policy Accelerator is determined both by the physical connections between the Policly Accelerator interfaces and the network, and by actions that your application takes to specify how to handle packets moving through the physical connections.

Physical Packet Flow

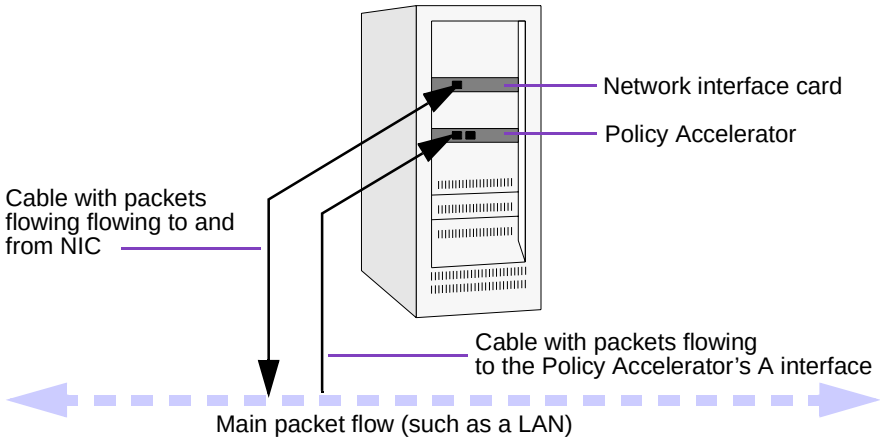
Each Policy Accelerator has two physical interfaces, named A and B, as shown in the following diagram:



By default, Policy Accelerators work in promiscuous mode; that is, when a cable is connected, all packets flow through the cable to the Policy Accelerator. It is your application that determines what happens to the packets after they reach the Policy Accelerator, if anything.

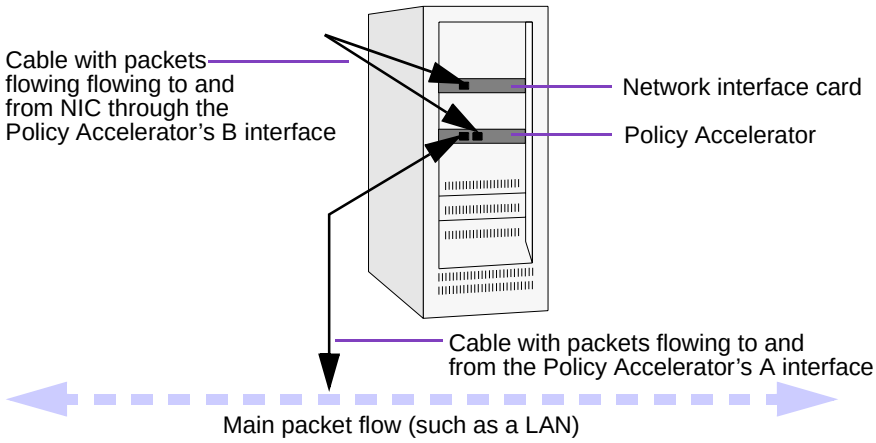
Here are two possible ways in which you might connect the Policy Accelerator, assuming that it is installed in the host development system on which you are testing your application:

- The Policy Accelerator "sniffs" the packets flowing across the main network. This kind of application looks at packets but does not pass them on. For example, the Policy Accelerator receives copies of all packets, counts them, and drops them. In this case, your cabling might look as follows:



  The CountApp application in Chapter 2, "Tutorial: Creating a Simple Application," assumes this kind of connection.

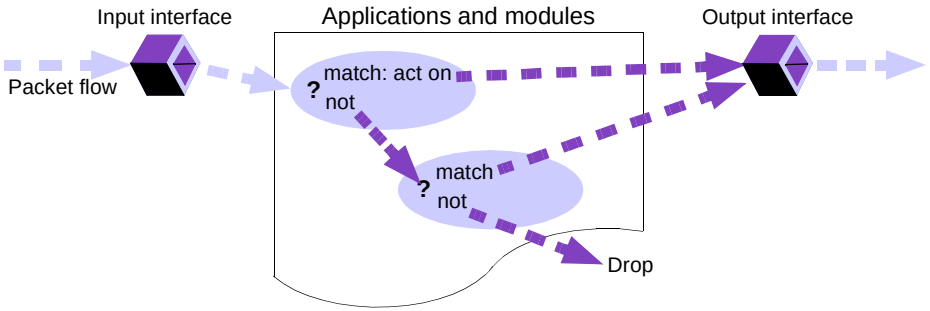
- The Policy Accelerator sits between your standard network interface card (NIC) and the main network, so packets must flow back and forth across the Policy Accelerator the same as in a standard network connection:



When you have this kind of physical connection, you must create logical bindings for packet flow in both directions—that is, you must bind both the TO and FROM system ACEs for each interface.

Logical Packet Flow

The flow of packets through Policy Accelerators can be thought of as connections among interfaces and applications, as shown in the following figure:



These connections are called *bindings*. The API provides the tools for you to bind interfaces and applications together. Packets flow through a Policy Accelerator only after you have created bindings.

ACEs are the entities that receive and dispose of packets. In a simple application such as the sample in this tutorial, packets arrive at an ACE, which you define, from a network interface and leave using another network interface. Because these interfaces must also be represented by ACEs, the SDK includes

system-defined ACEs for each of the two network interfaces on the Policy Accelerator. The interface names, from which the system ACE names are derived, are as follows for the first Policy Accelerator installed in your system:

- `nbhwpe0A`
- `nbhwpe0B`

Binding Targets as Packet Destinations

In an ACE, packets flow in only one direction: through the ACE to a *target* within the ACE. A target provides a way to connect an ACE to a destination for packets. Each ACE can have multiple targets.

For example, destinations can be:

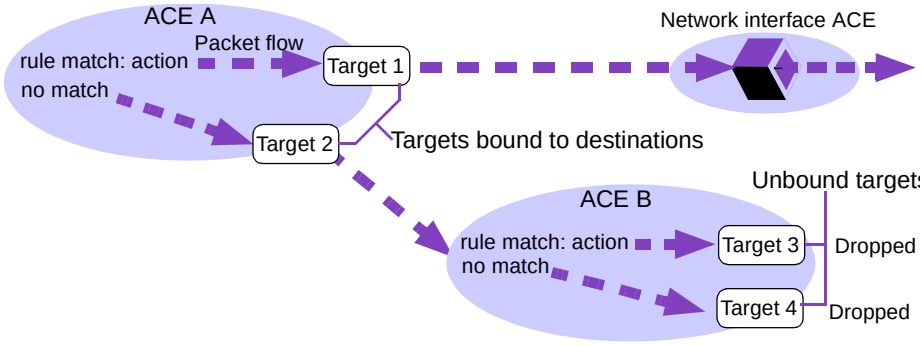
- Another ACE in the same application
- An ACE in a different application
- A network transmission queue
- A network interface
- A built-in service for dropping packets or for cryptography

Requirement for Packet Flow

For a packet to flow out of an ACE, your application must:

1. Bind a target to a destination ACE.
2. Direct the packet to the target.

NOTE: Directing a packet to a target without binding the target to a destination ACE causes packets to be dropped, as shown in the following figure.



The IX-API SDK programming model requires that:

- Every source that sends packets must be represented by an ACE that contains targets.
- Every destination must be represented by an ACE.

System-Defined Targets and ACEs

Targets are represented in your action code by `Target` objects. Each ACE has two default target objects:

- Target named `pass`
- Target named `drop`

NOTE: These targets are named for your convenience only; remember that packets flow only after you have created bindings. For example, if you bound the target named `drop` to a network interface, packets would not be dropped.

In addition, the SDK includes system-defined ACEs for each of the two interfaces on the Policy Accelerator. The interface names, from which the system ACE names are derived, are the following, where *n* indicates which Policy Accelerator:

- **nbhwpe***n***A**
- **nbhwpe***n***B**

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, using the following name:

- **nbhwpe***n***C**

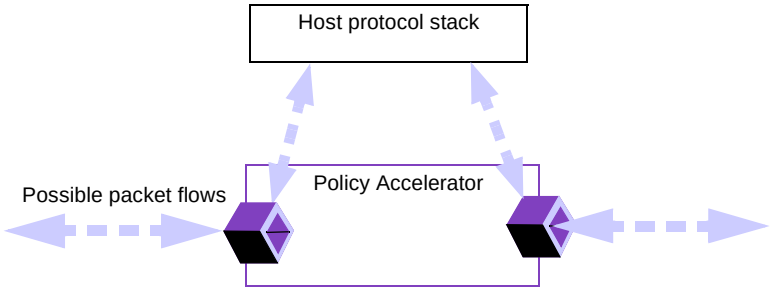
Each interface can be used for one or more of the following functions:

- Input from network
- Input from host protocol stack
- Output to network
- Output to host protocol stack

Each of these functions corresponds to a system ACE. For example, to pass packets out to the network through interface A on Policy Acclerator 0, you would bind the system-defined pass target as follows:

```
bind ("/MyAppl/MyAceGroup/MyAce/pass",
          "/nbhwpe0/ToInterface:nbhwpe0A/Interface");
```

The following figure shows possible directions for packet flow:



NOTE: For details on names of interfaces and default system ACEs, see Appendix C, “Policy Accelerator Name Space,” in the *IX-API SDK Reference*.

Binding Example

The example given in Chapter 2, “Tutorial: Creating a Simple Application,” binds the pass target of the first Policy Accelerator interface to your `CountAce` subclass for incoming packets, and then binds the default pass target in `CountAce` to the system ACE for the second interface:

```
rval = bind
   ("/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass",
   "/CountPackets/CountAceGroup/CountAce");
if (rval != NB_SUCCESS) {
   NB_ABORT(rval);
}
rval = bind ("/CountPackets/CountAceGroup/CountAce/pass",
          "/nbhwpe0/ToInterface:nbhwpe0B/Interface");
if (rval != NB_SUCCESS) {
   NB_ABORT(rval);
}
```

Unbound Targets

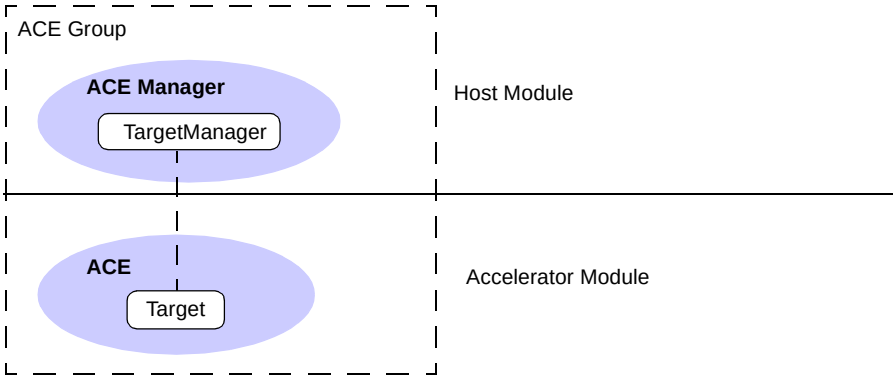
If you attempt to pass a packet to a target that has not been explicitly bound, the packet is passed to the drop target.

Defining Targets

To create your own targets within ACEs, you must:

1. Use the `TargetManager` class to define and create a *target manager* object in the host module.
2. Use the `Target` class to define and create a target object in the accelerator module, using the same dictionary name that you specified for the target manager.

Target managers on the host manage targets, which reside in ACEs on the Policy Accelerator. Normally, the target objects are contained within the ACE objects—that is, the constructor for the ACE object is defined to create the contained target objects.



Typcially, you define a subclass of `TargetManager` in your host module and a subclass of `Target` in your accelerator module—for example, `appTargetManager` and `appTarget`. The constructor for the ACE manager and ACE object then create instances of each for each target that you require.

For example, in an `NBAppl` named `myApp`, in `myAce` managed by `myAceManager` within `myAceGroup`:

- In the host module, you create the target manager object:

```
myTargetManager =
   new appTargetManager (myApp, myAceGroup,
                         myAceManager, "myTarget");
```

- In the accelerator module, you create the target object. The ID of the application and the ACE object are automatically passed from the host module. You specify only the dictionary name of the new target:

```
myFirstTarget =
       new appTarget (id, ace, "myTarget");
```

NOTE: You must give the same dictionary name string—in this case, `myTarget`—to both the target manager and to its associated target.

The set of outgoing targets in an ACE represents the set of all possible packet destinations in that ACE, across all actions. An ACE that performs a demultiplexing function, for example, might have multiple outgoing targets.

Directing Packets to a Target

Action functions in the accelerator module determine the path that a packet takes through an application by specifying whether the packet has completed processing and by directing the packet to a specific target.

Packets are represented within action code by a `Buffer` object.

An action function must:

1. Use one of the following methods to designate the target through which the packet buffer is directed when the ACE's processing is complete:
   - A `Target` object's `take` method directs the buffer to this target.
   - An `Ace` object's `pass` method directs the buffer to the default pass target.
   - An `Ace` object's `drop` method directs the buffer to the default drop target.
2. Return one of the following constant values to indicate how rule processing should continue, if at all, on this packet buffer:
   - `RULE_DONE`, `RULE_TOOK`, `RULE_CONT`, or `RULE_DEFER`

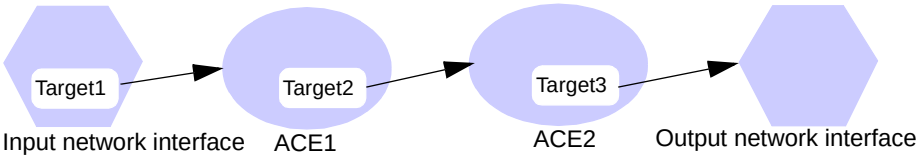
For more information on these constant values and on writing action code, see the following:

- Chapter 7, "Acting on Packets in Your Action Code."
- "Action Functions" on page 115 in the *IX-API SDK Reference*

Using Targets to Serialize Packet Processing

An application can bind targets to several ACEs, possibly on different Policy Accelerators, in order to perform serial processing on packets.

The following figure shows a simple cascade of ACEs used by a single application. Processing occurs serially from left to right. In this example, each ACE has a single destination. Targets 1 and 2 are bound to ACEs 1 and 2, respectively. Target 3 is bound to a network interface in the outgoing direction.



When several Policy Accelerators are installed, ACEs can execute more efficiently (using pipelining). Note, however, that after one ACE has finished processing a packet, the packet is given to another ACE that *might execute on the same hardware resource.* Thus, the number of ACEs an application employs should be limited.

Chapter 6
Classifying Packets Using NCL

This chapter describes how an ACE classifies incoming packets using its classification code, which you write in Network Classification Language (NCL).

This chapter contains the following topics:

- How ACEs Handle Packets
- What's in the NCL Rules File
- Defining Protocols
- Defining Rules
- How Rules Are Evaluated
- What Rules Do

How ACEs Handle Packets

An ACE does three things with a packet:

- Classifies it
- Acts on it
- Disposes of it

Packets are received by the Policy Accelerator as network traffic flows through it. A packet arrives at an ACE from the source to which it is bound as a destination target, as described in Chapter 5, "Controlling Packet Flow." The source of a packet could be, for example, either a network interface or another ACE earlier in the processing chain. The ACE classifies each incoming packet according to classification rules defined in the ACE's NCL rules file.

As a result of the classification, NCL rules trigger actions that are defined in the ACE's compiled action code. Actions can act on packets by examining or retrieving their contents, or dispose of packets by sending them to targets. An

action can make a final disposition, ending the classification process for that packet, or it can tell the ACE to continue with the classification process and evaluate more rules.

What's in the NCL Rules File

An ACE classifies incoming packets according to the protocol and predicate definitions in its NCL rules file. Based on the classification,rules in the NCL rules file trigger actions in the associated action file.

The NCL rules file has two general parts. In the first part, you define the protocols and predicates that are the basis of classification. In the second part, you define the rules, which trigger specific actions based on how a packet satisfies the predicates. In addition to these basic elements, you can define sets and searches in the NCL rules file.

Classification Elements

The basic classification elements you define in the NCL rules file are rules and protocols.

- A rule determines whether some statement is true for the current packet. If it is, the rule specifies an action to take. The action is a function that you defined in the action code that is part of the same ACE as the NCL rules file.
- A protocol definition describes a set of fields of particular lengths. A protocol is usually added on as a header to one or more other nested protocols. The definition states which other protocols can be nested, and where they start.

For convenience, you can also define named predicates to use in the rules and protocol definitions. A predicate determines whether something is true of a current packet — for example, whether it conforms to a certain protocol, or whether a particular field has a particular value. Predicates are named Boolean expressions, which can contain other predicates and Boolean expressions. You can define predicates as part of a protocol definition or separately.

Sets and 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.

You define and name the sets and searches in the NCL rules file, then use the compiler to generate corresponding objects in the action code. The sets are implemented on initialization. You populate the sets with data through actions.

Searchable sets define collections of packets which are associated with each other by virtue of their *contents*. A set member (element) matches packets with a specific combination of field values. 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; see "Defining Protocols" on page 88.

Sets are an efficient way to track a large number of classes of packets. They allow you to keep state information for all packets that have the same values in specific fields. For example, you can count the number of packets that flow between any two specific IP address pairs, or maintain state for each TCP stream.

For more information on sets and searches, see Chapter 9, "Using Sets of Data to Classify Packets."

Defining Rules

NCL *rules* determine what action to take, based on the classification. A rule consists of two parts:

- A *predicate* part

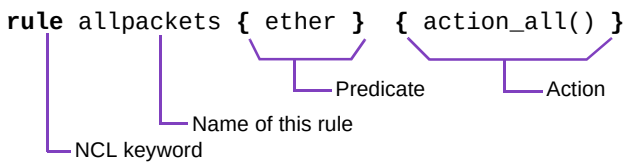
  The predicate part of a rule is a Boolean expression that describes the conditions a packet must meet to have the specified action performed on it.

  Predicates might include static comparisons of packet contents or comparisons against dynamically-created state.

- An *action* part

  The action part of a rule is the name of an action function to be executed when the predicate is true, along with any additional arguments the named function expects. The action function performs some action upon the incoming packet.

The following example shows the parts of a rule:



The predicate part of the rule can be the name of a predicate defined earlier in the file or in an included file, or it can be a Boolean expression that can, in turn, use named predicates. A predicate uses packet protocol elements that are defined earlier in the NCL rules file or in an included file. You access protocol fields using the form *protocol_name.field_name*; for example, `ip.src`. In a predicate, the name of the protocol currently being parsed evaluates to `TRUE`.

The action part of the rule contains the name of an action function defined in the compiled action code that is part of the same ACE. You can pass arguments to the function, as required by its definition. The action is executed only when and if the predicate part of the rule evaluates to `TRUE`.

Defining Protocols

A rule determines whether its predicate is true by looking at fields in the current packet. The fields are interpreted according to the protocol definitions in the NCL rules file.

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.

The IX-API SDK distribution includes NCL include 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`

Protocols often contain other protocols. The keyword `demux` in a protocol definition identifies 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.) 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.

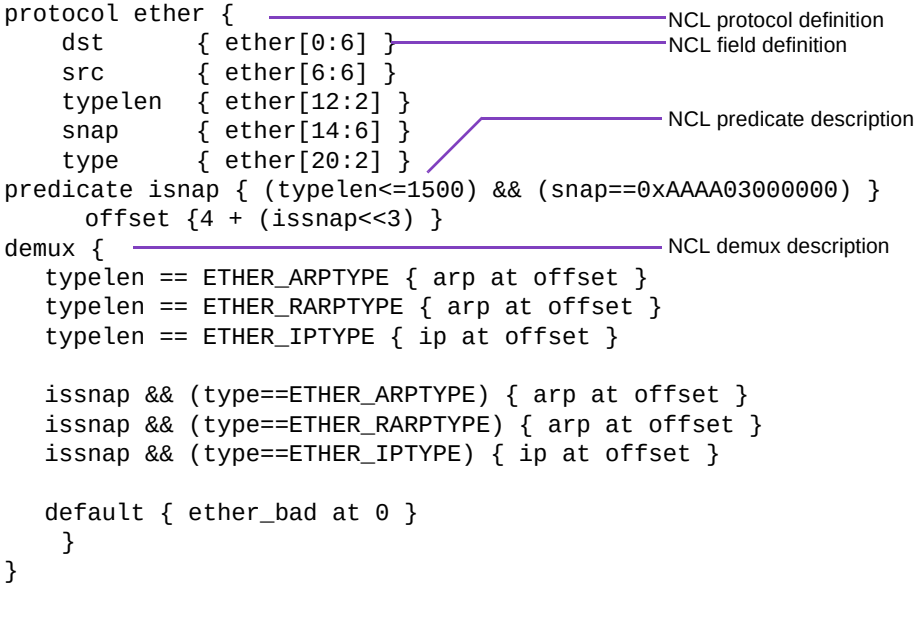
For example, the IP protocol definition contains the following `demux` statement that specifies what other protocols could be nested in an IP packet, and at what offset they would be found:

```
demux {
   invalid       { ip_bad at 0 }
```

```
   badsrc        { ip_badsrc at 0 }
   (proto == 1)  { icmp at hlen }
   (proto == 2)  { igmp at hlen }
   (proto == 6)  { tcp at hlen }
   (proto == 17) { udp at hlen }
   default       { ip_unknown_transport at hlen }
}
```

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`.

The following example shows an excerpt from the `NBtcpip.ncl` include file. This excerpt defines the header fields for Ethernet packets, defines a reusable Boolean function (predicate) for the protocol, and specifies the demultiplexing (demux) method to high-layer protocols.



NCL syntax is described in detail in the *IX-API SDK Reference.*

How Rules Are Evaluated

Packets arrive at an ACE as described in earlier chapters. When each packet arrives at the ACE, it goes through a classification phase. During classification, the ACE evaluates the rule predicates in the order in which they appear in the NCL rules file. When a predicate evaluates to `TRUE`, the rule's action goes on a queue to be executed. The order of actions in the execution queue is the same as the order of the successful rules in the NCL rules file.

When all of the rules have been evaluated, the Policy Accelerator begins executing the actions in the execution queue. When an action is executed, it returns a value indicating whether it *disposed of* the packet. An action *must* return this value. For more information on defining actions, see Chapter 7, "Acting on Packets in Your Action Code."

Disposing of a packet corresponds to taking the final desired action on the packet for a single classification step; for example, dropping it, queuing it, or delivering it to a target.

- If an action returns the code `RULE_CONT`, indicating that it did not dispose of the packet, the ACE continues executing queued actions, starting with the next one.
- If an action returns the code `RULE_DONE`, indicating that it did dispose of the packet, the classification phase terminates. The ACE does not execute any further actions remaining in the queue.
- If all actions are executed without disposing of the packet, the packet is delivered to the *default target* of the ACE.

For more information on what actions can be taken and how to define them, see Chapter 7, "Acting on Packets in Your Action Code."

What Rules Do

A rule can simply check the protocol type and take an appropriate action. For example, a VPN application might have a rule to check for an IPSec packet. When the rule succeeds, the action decrypts the packet. The rule might look like this:

```
rule check_ipsec { ipsec } { action_decrypt () }
```

A rule can pass field values to the action. For example, the following rule passes the source and destination addresses:

```
rule check_ip { ip } { action_ip (ip.src, ip.dst) }
```

The rule itself can check the source or destination address, or both, against a particular address, or table of addresses.

A rule that checks a specific address might look like this:

```
rule check_src { ip.src == 10.10.10.30 } { action_A() }
```

You can use sets to keep tables of addresses, and named searches to check incoming packet addresses against the table. For more information on sets and searches, see Chapter 9, “Using Sets of Data to Classify Packets.”

The rule can check other fields in the packet. For example, in an intrusion detection application, a rule might check whether a TCP packet is part of an HTTP stream. The rule might look like this:

```
rule check_http { tcp && (tcp.sport == 80 || tcp.dport == 80) }
   { action_scan () }
```

When successful, this rule passes the packet to an action function, where it is queued, reordered, and then searched for interesting strings. If the function finds such a string, it sends a notification to the application using an upcall.

For more information on:

- Defining action functions, see Chapter 7, “Acting on Packets in Your Action Code.”
- String searches, see Chapter 10, “Finding Strings in Packets.”
- Upcalls, see Chapter 8, “Communication Within an Application.”

Chapter 7
Acting on Packets in Your Action Code

This chapter describes the action portion of an ACE. It explains what is in the action code file, how to write it, and what kind of things it might do.

This chapter contains the following topics:

- Action Code Overview
- What Is in an Action Code File
- Initializing the Action Part of an ACE
- What Is in the Action Part of the File
- What Action Functions Do

Action Code Overview

You implement the action portion of an ACE, which consists of action code that is compiled on the host, loaded into an ACE during initialization, and run on the Policy Accelerator. You write action code using the following:

- The C and C++ runtime environment
- Restricted standard libraries appropriate to the Policy Accelerator execution environment, which does not support floating-point math, file system access, or multithreading
- The Action Services Library (ASL), which provides added functionality for developing network applications; specifically, it 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)
  - Support for the ACE structure
  - Inter-module communication via upcalls, downcalls, and crosscalls
  - The association of arbitrary data with packets using data sets
  - Support for string searches in packet data

- Support for some basic system services, including timers, statistical counters, and memory management
- Support for action code, by means of the ASL extensions, to handle many TCP/IP functions such as IP fragmentation and reassembly, network address translation (NAT), and TCP connection monitoring (including stream reconstruction)

The ASL API is described in detail in Chapter 4, "Action Services Library," of the *IX-API SDK Reference.*

What Is in an Action Code File

Each ACE contains one action file associated with one NCL rules file. Your action code file contains two loosely defined parts:

- The initialization part creates the ACE object, and also creates and initializes any other subclasses, objects, and structures that you will need. The entry point is the ASL `init_actions` function, which acts as a `main` function for the ACE.
- The action part contains the action and callback function definitions. These are entry points that implement the behavior of your application in response to incoming packets, messages, or other events. The action part also contains method definitions for any methods you have created in your subclasses.

Initializing the Action Part of an ACE

In the initialization portion of your action code file, you define all of the ASL subclasses and other data structures that the application will need. You must also create and initialize the ACE itself, as well as any other objects and data you will need.

Defining ASL Subclasses and Objects

Each action code file is associated with one ACE, which is represented by an object of the `Ace` class. Typically, the object is a member of an `Ace` subclass that you define in the action file.

Your `Ace` subclass must contain references to objects in the ACE. These can include:

- Communication objects. These can include `Upcall`, `DowncallHandler`, `Crosscall`, and `CrosscallHandler` objects. For more information, see Chapter 8, "Communication Within an Application."

- String search objects. These can include `NBSearchContext`, `NBString-SearchEngine`, and `NBStringMatchReport` objects. For more information, see Chapter 10, "Finding Strings in Packets."

  You do not usually need to create subclasses for communication and string search objects. The constructor for the `Ace` subclass creates the objects as members of the parent class.

- Data set objects. For each data set, the ACE must contain a set object of the subclass defined in the header file generated from the NCL. The object must have the same name that was declared for the set in NCL. (Your action code file must also define subclasses for set elements, but the ACE need not point to the element objects.) For more information, see Chapter 9, "Using Sets of Data to Classify Packets."

Your `Ace` subclass also typically contains methods to perform application functions, such as sending messages or initiating searches, as well as callback methods for various purposes. For more information, see "Defining Callbacks" on page 97 and "Defining Other Methods" on page 99.

Initializing the ACE

Your action file must contain a "main" function, `init_actions()`. In this function, you must create and return the `Ace` object for the ACE to which the code belongs, using code such as the following:

```
init_actions(ModuleID id, char * name, Image*  obj)
{ ...
   return new MyAce(id, name, obj);
}
```

The Policy Accelerator calls this function immediately after receiving the NCL and action code from the host (that is, when you call the application's `load` method in the host module). In addition to creating the ACE object, the function can perform any other initialization your application needs. It can, for example, create additional objects, populate sets, or initialize other data structures.

What Is in the Action Part of the File

In the action part of the action code file you define the methods and functions that implement the behavior of your application. You can implement behavior in any of the following ways:

- Action functions

  An action function is executed when an NCL rule's predicate is `TRUE` for an incoming packet. Action functions can call methods or other functions directly, or send messages that result in the execution of callbacks.

- Callbacks

  Callbacks are executed in response to downcalls or crosscalls, scheduled events, or the expiration of set elements. You define a callback as a member of your subclass of the appropriate class.

- Other subclass methods

  You can provide definitions for methods you have added, or redefine existing methods in your own subclasses. Methods are not entry points. To execute a method, you must call it from an action function or callback.

Defining Action Functions

Actions are function entry points implemented according to the calling conventions of the C/ C++ programming language. An action function is executed when an NCL rule predicate is satisfied for an incoming packet. The rule must pass any arguments that you have defined for the action function, and the action function must return one of the disposition codes that tell the NCL compiler whether to continue processing the rules.

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.

Action Function Return Values

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

Return Code	Description
`RULE_DONE`	Return `RULE_DONE` 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 `RULE_TOOK` 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. **NOTE:** The pass and drop methods return this code.
`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.

Predefined Action Functions

Two action functions are already defined in the ASL to perform simple and common operations:

- The `action_pass` function routes a packet to the ACE's pass target.
- The `action_drop` function routes a packet to the ACE's drop target.

Both of these functions return the code `RULE_TOOK`, which halts further rule processing for the packet. Neither takes any argument. To call one of these functions in an NCL rule, use a rule like the following:

```
rule passip { ip } { action_pass() }
```

This would pass all IP packets through to the ACE's pass target without further processing.

For More Information

For more information on defining action functions, see "Action Functions" on page 115 in the *IX-API SDK Reference*.

Defining Callbacks

In the action part of your action code file, you define the callbacks that are executed in response to various events. You can define the following kinds of callbacks:

- Downcall and crosscall handler callbacks

  The callback or service function for a downcall or crosscall handler implements the behavior to be executed when the downcall or crosscall is received. In response to the `Crosscall` or `Downcall` object's `call` method, the host executes the service function specified in the associated `CrosscallHandler` or `DowncallHandler` object, passing it the specified message.

  A call handler service function takes as its argument the message passed by the call, and returns nothing. It must be a member of a subclass of the `Ace` class.

  For more information on call handler callbacks, see "Defining Message Handling Callbacks," in Chapter 8, "Communication Within an Application."

- Message completion callbacks

  The message completion callback implements behavior to be executed when a message transfer is finished. Because message transfer is asynchronous, you can use this callback to ensure that values do not change or memory is not freed before the message transfer is complete.

  This kind of callback function does not need to be a member of a subclass.

  For more information on call completion callbacks, see "Creating Messages and Message Blocks," in Chapter 8, "Communication Within an Application."

- Scheduled event callbacks

  The `Event` class provides for execution of functions at arbitrary times in the future. You can reschedule or cancel an event, or change the function that it executes. The event callback method implements the behavior to be executed at the scheduled time.

  You must define an event callback method as a member of the same `Event` subclass that uses it.

  For more information on event callbacks, see "`Event` Class" on page 166 in the *IX-API SDK Reference.*

- Set element expiration callbacks

  When you call the `expire` method of a set element, you can pass a callback function that you have defined. This function normally deletes the set element, using the `delete` operator for the subclass. Elements are not automatically deleted on expiration; to cause this, you must define a callback to do it.

  You must define an expiration callback method as a member of the same `Element` subclass as the `expire` method that uses it.

  For more information on expiration callbacks, see "`Elt_setname` Class" on page 160 in the *IX-API SDK Reference.*

- String search callbacks

  You must define callbacks to handle the strings found by string searches in packet buffers. These callbacks are of two types, per-match and per-buffer, and you define them in your action code as part of a `NBStringSearchContext` object.

  Changes of state that you make in the string search engine or context can be delayed if a search is in progress; you can define callbacks to be invoked when such a change actually takes place.

  For more information on string searches, see Chapter 10, "Finding Strings in Packets."

Defining Other Methods

You can define other new methods for your subclasses, or redefine existing methods. A method is executed when you call it from an action function, a call-back, or another method. For example, an Ace subclass typically has a method that creates a message and sends it to the host in an upcall.

The following is an example of a method defined in an Ace subclass. It creates a new packet from scratch to send from an ACE. Before defining this method, you would need to create the MyAce subclass and declare the member method.

```
void MyAce::SendOnePacket ( void )
{
   // Create a new buffer object
   Buffer *pBuf = new Buffer;
   if ( pBuf ){
      // Use the append method to allocate memory in the buffer
      if ( pBuf->append ( 1024 ) != NULL ){
           // Use packetBase to get pointer to start of buffer
           memset ( pBuf->packetBase (), 1,
                              pBuf->packetSize () );
           // Use pass_.take method to pass it along
           pass_.take ( pBuf );
           // Delete ref count to this buffer.  Buffers only get
           // TX when the ref count == 0
           pBuf->decref ();
      }
      else {
           // Can't append to the buffer
           // Free the buffer by decref, never 'delete'
           buffer pBuf->decref ();
      } }
   else{  // Can't create a new buffer
   } }
```

What Action Functions Do

You can define action functions to respond to the successful application of a rule in any way you want. Actions can do any of the following:

- Pass the current packet to any target
- Modify a packet before passing it to a target
- Increment the buffer reference counter to indicate that the buffer is in use, and decrement the reference counter when the function is finished with the buffer
- Increment and decrement application-specific counters

- Build and send messages using upcalls or crosscalls; for more information, see Chapter 8, “Communication Within an Application.”
- Add or delete elements or change data in sets; for more information, see Chapter 9, “Using Sets of Data to Classify Packets.”
- Start or continue a search for a specific string, or a string matching a pattern, in the data portion of a packet buffer; for more information, see Chapter 10, “Finding Strings in Packets.”
- Call other functions or methods

The following table shows some typical actions you might define for different types of applications. Notice that, while the action is triggered by a rule that checks a particular condition, the action itself can continue to check conditions before deciding what to do.

Application Type	Classification	Action
Firewall Controls access by maintaining a set of acceptable source addresses	If packet IP src address is in my table of addresses that have access to my network	Route the packet to the appropriate location inside my network
Firewall Translates packet addresses based on a set of address mappings	If packet IP dst address matches a known address for inbound traffic	Change the dst address to the new address specified by the matching set element Recompute checksums
	If the packet IP src address matches a known address for outbound traffic	Change the src address back to the original mapping address specified in the matching set element Recompute checksums

Application Type	Classification	Action
Load-balancing Redirects packets to least-loaded servers by maintaining a set of possible destination servers	If the IP src and dst match an element in the server set	Perform address translation (NAT) to change dst to value from matching element Recompute the checksum Send the packet to the identified server
	If the IP src and dst do not match any element in the server set, pass the pointer to where the record should go to the action	Use a load-balancing algorithm to determine the server to which the connection should go Create a new element for the server and add it to the set Perform NAT on the packet and send it to the identified server
Intrusion detection	If the packet is part of a TCP stream	Search the packet for interesting strings If found, notify the application using an upcall
Network monitoring	If the packet is HTTP and has a specified IP src address	Search the packet for the name of the web page of interest If found increment a counter called `mywebpage`

Application Type	Classification	Action
Quality of service Maintains higher and lower priority queues for transmitting traffic	If the IP destination address equals that of the SAP financial server	Place the packet on a high priority queue for transmission
	If the IP source address equals that of a known customer who buys a lot of merchandise at your online store	Place the packet on a high priority queue for transmission
	If the IP destination address is the ESPN web site and the source address is Bob's machine who works in finance	Place the packet on a low priority queue for transmission
VPN	If the packet is an IPSec packet	Decrypt the packet using the crypto API calls.

Chapter 8
Communication Within an Application

This chapter contains the following topics:

Overview

To pass data between different parts of your application, you encapsulate the data in a *message*, which you construct from *message blocks*. You can pass the message to a handler function, or *callback*, in another part of the system, by means of associated objects in the host module and accelerator module. One kind of object, a *call*, sends the message, and another kind of object, a *call handler*, receives the message and passes it to the callback for processing. Calls are asynchronous.

You can send messages:

- From the host module to the accelerator module, using *downcalls*
- From the accelerator module to the host module, using *upcalls*
- From one ACE to another ACE, in the same or another Policy Accelerator, using *crosscalls*

Communication Between the Host and the Policy Accelerator

Upcalls and downcalls allow communication between the host and the Policy Accelerator. You can use upcalls and downcalls in your application to share information or to signal between the host module and the accelerator module. For example, you can pass

- Memory blocks
- Packet contents
- Notifications

In general, you can construct and pass messages of any sort, as if you were making asynchronous remote procedure calls.

You typically use upcalls and downcalls to send configuration or statistical information, or control messages. You can use upcalls and downcalls to pass individual packets, or small collections of packets, between the host and Policy Accelerator, but it is not an efficient way to forward packets. To forward packets or to transfer large amounts of data between the Policy Accelerator and the host, use the host protocol stack. See “Moving Packets between the Policy Accelerator and the Host” on page 114.

NOTE: To make applications run faster, do most of your packet processing on the Policy Accelerator.

Calls and Call Handlers

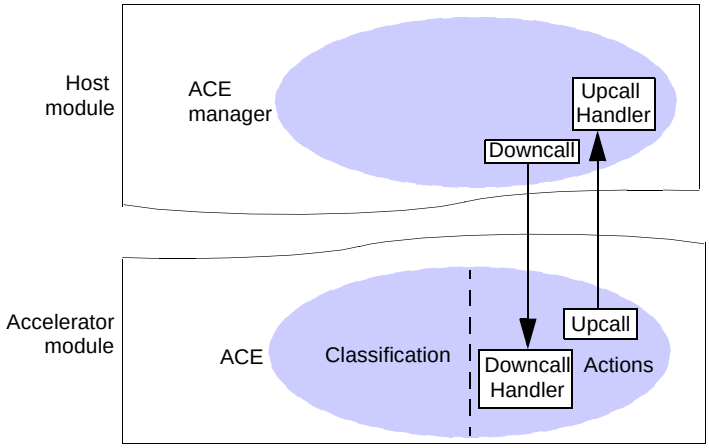
To send messages between the host module and the accelerator module, construct a pair of objects: a call object on one side that sends the message, and a handler object on the other side that receives the message.

There are two types of calls and call handlers:

- A downcall sends a message from the host module to the accelerator module action code. You create a downcall object in the host module, and a corresponding downcall handler object in the accelerator module.
- An upcall sends a message from the accelerator module action code to the host module. You create an upcall object in the accelerator module and a corresponding upcall handler object in the host module.

You associate a call object and its handler object by giving them the same dictionary name when you construct them; that is, you pass the same value to each as the *name* argument of the constructor.

The relationship between calls and handlers is shown in the following figure.



Making Upcalls

To make upcalls from the accelerator module to the host module:

1. In the host module, define a method in the ACE manager subclass to act as a message handler. This is known as a *callback* or service function.
2. Create an `Upcall` object in the accelerator module, noting the `name` value in the constructor.
3. Create an `UpcallHandler` object in the host module:
   - For the *`name`* argument, specify the same value that you used for the `Upcall` object in Step 2.
   - For the *`argUpcallFunction`* argument, specify the name of the callback that you defined in Step 1.
4. In the accelerator module, use the `Message` and `MessageBlock` constructors to create a message object containing the data you want to send.
5. In the accelerator module, pass the message to the `call` method of the `Upcall` object.

The handler object in the host module receives the message and directs it to the callback that you supplied.

Making Downcalls

To make downcalls from the host module to the accelerator module:

1. In the accelerator module, define a method in the ACE subclass to act as a message handler.
2. Create a `Downcall` object in the host module, noting the `name` value in the constructor.

3. Create an `DowncallHandler` object in the accelerator module:
   - For the *`name`* argument, specify the same value that you used for the `Downcall` object in Step 2.
   - For the *`func`* argument, specify the name of the callback that you defined in Step 1.
4. In the host module, use the `Message` and `MessageBlock` constructors to create a message object containing the data you want to send.
5. In the host module, pass the message to the `call` method of the `Downcall` object.

The handler object in the accelerator module receives the message and directs it to the callback that you supplied.

For More Information

For more information on downcalls, see the following sections in the *IX-API SDK Reference*:

- "`Downcall` Class" on page 56
- "`DowncallHandler` Class" on page 150

For more information on upcalls, see the following sections in the *IX-API SDK Reference*:

- "`Upcall` Class" on page 273
- "`UpcallHandler` Class" on page 88

Communication among ACEs

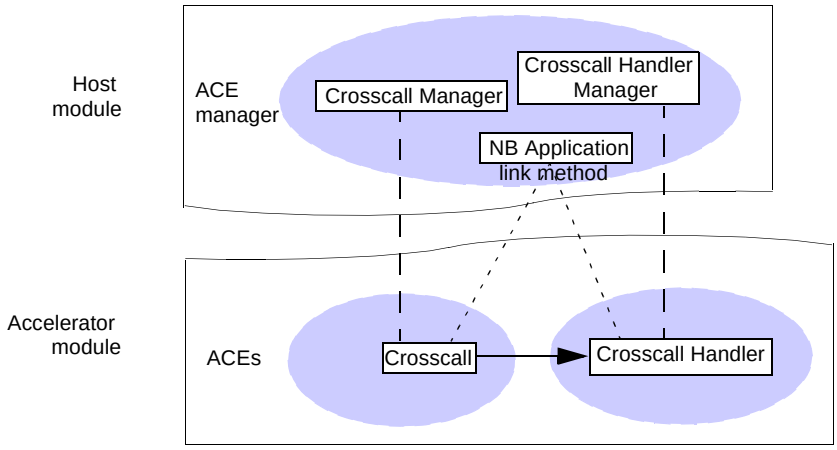
You can send messages from the action code in one ACE to the action code in another ACE using crosscalls. The sending ACE must have a crosscall object to send the message, and the receiving ACE must have a crosscall handler object to receive the message and direct it to a callback.

In addition, both the crosscall and crosscall handler objects must be paired with corresponding manager objects on the host. You associate the objects on the accelerator module with their manager objects on the host module by giving the paired objects the same names on creation.

To associate a crosscall with a crosscall handler in another ACE, use the `link` method of the application's `NBappl` object. To disassociate the two, use the `unlink` method. You can associate any number of crosscalls with the same handler. You can associate a crosscall with a handler in any ACE, whether or not it is in the same ACE group or on the same Policy Accelerator. For example, you

can use a single crosscall handler in one ACE to coordinate the activities of a number of other ACEs, by linking it with a crosscall object in each of the ACEs and sending it messages whenever those ACEs take particular actions.

The relationship between crosscalls and their handlers and managers is shown in the following figure.



Making Crosscalls

To make a crosscall from AceA to AceB:

1. In the host module, create a `CrosscallManager` object and a `CrosscallHandlerManager` object, noting the values of the name arguments.
2. In the accelerator module for AceA, create a `Crosscall` object, passing the same *`name`* value you used for the `CrosscallManager` object in Step 1.
3. In the accelerator module for AceB, define a method in the ACE subclass to handle the message.
4. In the accelerator module for AceB, create a `CrosscallHandler` object:
   - For the *`name`* argument, specify the same value that you used for the `CrosscallHandlerManager` object in Step 1.
   - For the *`func`* argument, specify the name of the callback that you defined in Step 3.
5. In the host module, call the `link` method of the `NBappl` object:
   - For the *`from`* argument, specify the complete path to the `Crosscall` object in AceA; for example, `MyAppl/MyAceGroup/AceA/MyCrosscall`.
   - For the *`to`* argument, specify the complete path to the `CrosscallHandler` object in AceB; for example, `MyAppl/MyAceGroup/AceB/MyCrosscallHandler`.

6. In the accelerator module for AceA, create `Message` and `MessageBlock` objects with the data you want to send to AceB.
7. In the accelerator module for AceA, pass the Message object you created in Step 6 to the `call` method of the `Crosscall` object.

The `CrosscallHandler` object in AceB receives the message and passes it to the callback you defined.

For More Information

For more information on crosscalls, see the following sections in the *IX-API SDK Reference*:

- "`Crosscall` Class" on page 139 and "`CrosscallHandler` Class" on page 144
- "`CrosscallManager` Class" on page 52 and "`CrosscallHandlerManager` Class" on page 48
- "`NBAppl` Class" on page 68

Creating Messages and Message Blocks

Messages and message blocks were introduced in "Sending Messages from the Policy Accelerator to the Host" on page 31. Message blocks specify the memory to be used in constructing a message. They are represented by different classes on the host and on the Policy Accelerator.

- For a downcall, you construct a message on the host using the `Message` and `MessageBlock` classes of the host API.
- For an upcall or crosscall, you construct a message on the Policy Accelerator using the `Message` and `MessageBlock` classes of the ASL.

These classes are similar, but not identical. In each case, you construct a message from one or two message blocks. There are several constructors for message blocks, which allow you to construct messages from arbitrary data or from the packet buffer.

The memory that you use in building a message is usually freed automatically on the calling side after the call is complete. (Because calls are asynchronous, the completion can occur some time after you actually make the call.) To retain the message on the calling side, or to clean up other memory associated with the message object, specify a callback method to be executed when the call is complete.

More memory is allocated for a copy of the message on the receiving side. You are responsible for freeing this memory when you have finished processing the message. You normally do so in the message handling callback method.

Allocating Space for Message Data in ASL

Message passing is asynchronous—that is, when you make an upcall or cross-call, the Policy Accelerator queues the message, but does not immediately send it. Also, messages that you build in the accelerator module contain pointers to the data to be sent, not copies of the data. This means that, by the time a message is actually sent, it is possible for the data in the message buffer to have changed. If the message buffer is an automatic variable (allocated on the stack), it will probably not even exist by the time the message is sent.

You can avoid this kind of problem by creating the message from a buffer that you specifically allocate for the purpose.

- When you are passing a small amount of fixed-length data to the host it is most efficient to allocate memory for the message data in an object field.

  For example, the simple application CountApp in Chapter sends a snapshot of the packet counter value to the host using an upcall. It defines fields in the ACE subclass for both the counter (`packetCounter`) and a snapshot of the current count (`CntSnap`). It uses the snapshot buffer to build the message:

```
class CountAce: public Ace {
public:
   CountAce (ModuleId id, char* name, image* obj);
int packetCounter;
nuint32 cntSnap;
...
void CountAce::showPacketCount (void){
   ...
   cntSnap = htonl (packetCounter);
   MessageBlock b ((char *) &cntSnap, sizeof (cntSnap));
   Message m (b);
   peekPacketUpcallHandle.call (&m);
   ...
}
```

- If you need to pass a large amount of data, or data of variable length, it is better to dynamically allocate memory for the data on the heap.

  For example:

```
   ...
   char databuf[size];
   ...
   MessageBlock mb(sizeof(databuf));
   Message msg (mb);
```

```
char *PtrSnapshot = msg.msg1();
memcpy (PtrSnapshot, databuf, sizeof(databuf));
...
```

Constructing Messages in ASL

There are several different constructors for the `MessageBlock` class of the ASL, for constructing messages from different kinds of data. The data pointers in the different constructors are handled in different ways.

- Use one of the following constructors to dynamically allocate a message area of a specified size, which is automatically released to the free pool after the message is sent:

  ```
  MessageBlock(int len)
  MessageBlock(int len, int off)
  ```

  After constructing the message block object in this way, write the data to the location returned by `MessageBlock.msg()`. This is probably the easiest, most generic way to send constantly changing data to an application. For example:

  ```
  void SendText( char *str )
  {
     int i = strlen ( len );
     MessageBlock m ( i+1 );
     strcpy ( m.msg (), str );
  // strcpy and memcpy are not efficient on accelerator -
  // a real app would not use them
     msg->call (&m);
  }
  ```

  When you use the *off* argument, the message starts at a small byte offset from the normally word-aligned allocated storage area.

- Use the following constructors to create messages from static strings or data:

  ```
  MessageBlock (char *msg)
  MessageBlock (char *msg, int len)
  MessageBlock (char *msg, int len, DoneFp done)
  ```

  The string or data argument should be static. When the message block is passed to `Message->call`, the data copy might not have happened by the time `Message->call` returns. If the contents pointed to by the argument changes, or if the argument goes out of scope (declared on the stack in a function that returns), the data the application receives by way of the upcall will have changed, or become invalid.

  For example, suppose you have the following two arrays, one declared static, and one not:

```
static WORD aStatic [5]={1,2,3,4,5};
       WORD aChange [5]={1,2,3,4,5};
```

You can use construct the following message from the static string "HELLO" :

```
MessageBlock m1 ("HELLO");
msg->call (&m1);
```

Because the array is static, you can also construct a message like this:

```
MessageBlock m2 ( &aStatic [0], sizeof ( aStatic ));
msg->call (&m2);
```

However, you cannot do the same with the dynamic variables. For example, consider the following message construction:

```
MessageBlock m3 ( &aChange [0], sizeof ( aChange ));
msg->call (&m3);
aChange [0] = 7;
```

This could fail because aChange might go out of scope and be invalid by the time the message is sent. Similarly, the assignment (aChange[0] = 7) might execute before the message reaches the application, in which case the application would receive the array 7,2,3,4,5, rather than 1,2,3,4,5.

If you use the constructor with the *DoneFp* argument, the data you pass to it must remain valid and unchanged until the callback function you specify has been called. The callback is executed when the message has finished being sent, not when it has been received. In the case of a crosscall, this means that the message has been received by the host, on its way to its destination. It might not yet have been received by the crosscall handler in the destination ACE.

- Use the following constructor to create a message block to send a network buffer to an application:

  ```
  MessageBlock(Buffer *buf)
  ```

  When you construct a MessageBlock from a buffer, the method automatically increments the reference count on the Buffer object to indicate that it is in use, and decrements the count when the 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.

Constructing Messages on the Host

There are only two ways to construct messages in the host module. You can create a message from a fixed-length buffer or from a NULL-terminated buffer. Use one of the following constructors:

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

Omit the length argument to build a message from a NULL-terminated buffer, such as a single string. Provide the length argument to build a message from a fixed-length buffer, such as a list of NULL-terminated strings.

Byte Order in Message Data

Two computers, or a computer and the network, can disagree about the order in which the bytes of a 16-bit or 32-bit number are stored—most-significant-byte-first or least-significant-byte-first. If the receiver of a message uses a different byte order from the sender of the message, the content of the message is interpreted wrongly.

Because calls go across the network, you must be careful that the byte order of the data that you send in the calls is preserved. The sender and the receiver could use different byte orders, and either or both could use a different byte order from the network.

To preserve data integrity, use the `ntoh` and `hton` conversion functions provided in the API to convert all data values to a known byte order before creating the message, and to convert the values to the locally applicable byte order after receiving the message. This process is known as *marshalling* the arguments.

For example, the following code converts a number to network order before creating a message from it, and sending the message to the host in an upcall:

```
msg = htonl (packetCounter);
MessageBlock b ( (char *)&msg, sizeof (msg));
Message m (b);
peekPacketUpcallHandle.call (&m);
```

The following upcall handler callback is defined to receive this message. It converts the argument back to host order from network order:

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

For More Information

For more information on:

- Constructing messages on the Policy Accelerator, see the ASL's "`Message` Class" on page 180 and "`MessageBlock` Class" on page 184 of the *IX-API SDK Reference*.
- Constructing messages on the host, see the host API's "`Message` Class" on page 62 and "`MessageBlock` Class" on page 66 of the *IX-API SDK Reference*.
- Byte order, see "Byte Order Issues" on page 10 in the *IX-API SDK Reference*.

Defining Message Handling Callbacks

Once you construct a message and use the call object to send it to its destination, the call handler object receives it and directs it to a callback, or service function, for processing. You must define this callback as a method in the proper object:

- Define an upcall handler callback as a method in the ACE manager object in the host module. This callback processes messages sent from the accelerator module to the host in upcalls.
- Define a downcall or crosscall handler callback as a method in the ACE object in the accelerator module.
  - A downcall handler callback processes messages sent from the host to the accelerator module in downcalls.
  - A crosscall handler callback processes messages sent from other ACEs in crosscalls.

A message handling callback takes a pointer to a message as its only argument, and returns nothing.

You can specify the callback method to which messages should be directed when you create the call handler object. For downcall and crosscall handlers, you can change the callback at any time by using the `direct` method of the call handler object.

Releasing Message Memory

When a message is received, memory for it is allocated locally. In the case of upcalls, unless you want to keep the received message for further processing, the callback should release the memory allocated to the message when it has finished.

An upcall handler callback releases message memory using the `releaseMessage` method. For example:

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

In the case of messages received by the Policy Accelerator in downcalls and crosscalls, the call handler receives a pointer to a message that is owned by the system. The Policy Accelerator takes care of releasing the message pointer after the call is received. However, you are responsible for releasing the data blocks within the message. You must do this in the callback, after you have finished processing the message data.

A downcall or crosscall handler callback must release message block memory using the message's done method. For example:

```
void myHandler (Message *m)
{
/* find the primary data block */
   char *msg1 = m->msg1 ();
   size_t len1 = m->len1 ();
/* find the secondary data block */
   char *msg2 = m->msg2 ();
   size_t len2 = m->len2 ();

/* XXX- put code here to take action on the data blocks */

/* Get here when we no longer need the original data blocks. */
/* Trigger the appropriate callbacks to recycle them. */
   m->done ();
/* the caller still owns the Message object itself,
** so DO NOT use "delete m" here.*/
}
```

The `Message` object's `done` method triggers the completion callbacks for the message block or blocks, as appropriate to the call. If you specified a completion callback when constructing the message, it is called if needed.

For More Information

For more information on call handler callbacks, see the following sections in the *IX-API SDK Reference*:

- "`UpcallHandler` Class" on page 88
- "`DowncallHandler` Class" on page 150

- “`CrosscallHandler` Class” on page 144

Moving Packets between the Policy Accelerator and the Host

Upcalls and downcalls are not an efficient way to forward packets. Instead, use the host protocol stack to forward packets from the host to the Policy Accelerator or from the Policy Accelerator to the host.

The following ACEs are defined by the system to allow access to the host protocol stack:

To host stack bound to interface A	`/nbhwpe0/ToStack:nbhwpe0A/Stack`
From host stack bound to interface B	`/nbhwpe0/FromStack:nbhwpe0B/Stack`

Each of these system ACEs (like all ACEs) contains predefined targets named `pass` and `drop`.

- To transfer packets from the Policy Accelerator to the host, bind the stack ACE `pass` targets and pass the packets to them. The host treats packets received on the stack like any other data arriving on the network.
- To transfer packets from the host to the Policy Accelerator, use your operating system’s standard data transfer tools (such as sockets on UNIX, or WINSOCK on NT) to transfer the packets to the host protocol stack. The packets arrive at the corresponding ACE in the Policy Accelerator.

For more information on system-defined ACEs, see “Naming Objects for the Resolver” on page 55.

Chapter 9
Using Sets of Data to Classify Packets

This chapter describes sets and searches, which you can use to classify packets based on their contents; that is, the values of fields. Sets are an efficient way to track a large number of classes of packets.

This chapter contains the following topics:

- Overview of Sets and Searches
- When to Use Sets
- Defining Sets and Searches
- Initializing and Populating Sets
- How to Use Sets and Searches

Overview of Sets and Searches

You can define data tables called *sets* that associate any arbitrary application-defined data with packets. For each set, you define one or more *named searches* that determine whether the current packet has a matching element in the set, based on the values of specified fields.

A set is a collection of elements where each element has a header of one or more *keys*, followed by any arbitrary data that you define. A search matches the key values in the header with the values of specific fields in an incoming packet. When all the values match, you can take an action with regard to the matching element; for example, add to a counter, or access and save another field value in the packet.

When a search does not match any element with the incoming packet, you have the option of adding a new element to the set.

When to Use Sets

You use sets to define collections of packets that are associated with each other by virtue of their *contents*. They allow you to keep state information for all packets that have the same values in specific fields. 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.

You can take any kind of action you want as a result of the set classification. For example, you can count packets going to or from each address, keeping the counters as set data. You can set targets for packets based on their set membership, add members to or delete members from the set, or create messages from the packets or the set data and pass them to the host or other ACEs using upcalls and crosscalls.

You can use sets to associate existing data with packets. For example, you can initialize a set with a list of interesting addresses, in order to classify all packets that come from or go to those addresses. One search might compare the `ip.src` field of packets to the elements of the set, to identify any packets that come from one of the addresses. Another search could compare the `ip.dest` field to the addresses, to identify packets that are going to them.

You can also use sets to collect new data about packet flow. For example, you can start with an empty set, and, for each incoming packet that does not already have a matching set element, create a new set element using its `ip.src` value, and add it to the set. When an incoming packet does match, you can increment a counter in the matching element. In this way, you can keep track of the sources of all packets coming through the ACE.

Defining Sets and Searches

You define and name the sets and searches in the NCL file, then use the compiler to generate corresponding objects in the action code. The sets are implemented on initialization. You populate the sets with data through actions.

You use NCL to declare the existence and suggest the size of a set, and to define the searches that evaluate set membership. You use the NCL compiler to generate corresponding ASL objects in a header file that you include in your action code. The set is implemented as a data table in the Policy Accelerator, and is created on initialization.

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.

The NCL Side

Use the **set** keyword to declare, name, and define a set in NCL. You provide the number of keys, and suggest a size for the set. For each set, you can define any number of searches, using the **search** keyword. Each search is associated with one and only one set, and you name it using the form *setname.searchname.* You specify a **requires** clause in the search. If this Boolean expression succeeds, the search is executed, and if it fails the search is not executed.

Each search definition specifies the exact number of keys that you specified for the set, and associates each key with a protocol field. When the search is executed, the Policy Accelerator compares the value of the specified protocol field in the current packet to the value of the corresponding key for each element in the set. If all of the field values in the packet match the key values in an element, the search succeeds, and keeps a pointer to the matching element.

You can use a search in a rule or an action, referring to it by the name, *setname.searchname.* When you refer to it by name, you are actually referring to the result of the search, which the Policy Accelerator obtains when it parses the search definition for an incoming packet. 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.

Example

The following NCL example defines a set named `Stream` that has 4 keys, and two searches in that set that compare each of the keys to fields in the IP or TCP protocol. The searches are only executed for TCP packets.

- The search `Stream.fwd` identifies a set element where the first two keys match the IP source and destination, and the second two keys match the TCP source and destination ports. A matching element would keep data for packets travelling from the first address to the second.
- The search `Stream.back` looks at the same field values, but matches them to keys in a different order. It identifies a set element where the first key matches the IP destination, rather than the source. The element found by this search would keep data for packets travelling in the opposite direction.

A rule passes the search result of `Stream.fwd` to an action function named `fTCPSTream`. The rule also succeeds only for TCP packets.

```
set Stream<4>
 { size_hint{1024} }

search Stream.fwd (ip.src, ip.dst, tcp.sport, tcp.dport)
```

```
 { requires { tcp } }

search Stream.back (ip.dst, ip.src, tcp.dport, tcp.sport)
 { requires { tcp } }

rule tcpstream {tcp}
 { fTCPStream( ip, tcp ,Stream.fwd ) }
```

For More Information

For more information on defining sets and searches in NCL, see “Sets and Named Searches” on page 403 in the *IX-API SDK Reference.*

The ASL Side

When you use the NCL compiler to generate the ASL set and search objects in a header file, it produces base classes to represent each set, its elements, and the results of each named search. For each set, the header file defines:

- A subclass of `Set` named `Set_`*`setname`*.
- A subclass of `Element` named `Elt_`*`setname`*. This is a base class whose constructor creates an element with the proper keys, as defined for the set. You create a subclass of this base class, modifying the constructor to define the data for the set elements.
- An instance of the `Search` class for each search, named *`setname.search-name`*. This object contains the result of the search and points to the found element or to a location where an action can insert an element in the set.

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 will be overwritten if you regenerate it.
- Create an object of the customized set subclass as part of the ACE object.

For More Information

For more information on creating and using ASL set objects, see the following sections of the *IX-API SDK Reference:*

- “Synchronizing NCL with Action Code” on page 411
- “Set Management Classes” on page 101

Initializing and Populating Sets

The generated header file defines the data structure that implements the set. You must include this header file in your action code. For example:

```
#include "CONNactSet.h"
```

Extending the Set Element Class

The generated header file contains a skeleton definition for set elements that has the number of keys you specified for the set, but contains no actual data definitions. You must extend this basic definition in a further subclass to define the kind of data you want to keep in the set. Do this in a file other than the header file, since the header file will be overwritten if you regenerate it.

The following example extends the generated element class for the `nets` set (`Elt_nets`) to contain specific net address data:

```
class NetAddr : public Elt_nets
{
public:
   NetAddr(int ip)
      : Elt_nets(ip),
        rx_pkts(0), tx_pkts(0), rx_bytes(0), tx_bytes(0)
   {}

   int rx_pkts, tx_pkts;
   int rx_bytes, tx_bytes;
};
```

Creating a Set Object

The ACE object for the ACE that is using a set must contain the set object. When you define the ACE subclass, include an object of the type `Set_`*`setname`*, and in the constructor for the ACE subclass, create an object of that type.

For example, the following ACE subclass definition declares two set objects:

```
class CONNAce : public Ace {
public:
   CONNAce (ModuleId id, char* name, Image* obj);
   ~CONNAce ();
   Set_nets nets;
   Set_conns conns;
}
```

The constructor then creates the set objects:

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

```
   ,nets (id, this, "nets")
   ,conns (id, this, "conns")
{}
```

Populating a Set

Creating the set object does not generate any element instances with which to populate it. To create an element, use the `new` operator, sending the arguments that you specified in the constructor for your element subclass.

You can populate a set with an initial set of elements, then later add or delete elements using actions, or you can populate it entirely through actions.

Populating a Set on Initialization

To populate a set during initialization:

1. Gather the data in the host module and build a message from it.
2. After the ACE is created, send the message in a downcall to the accelerator module.
3. In the downcall handler function, loop through the data items. For each item:
   a. Create a set element using the `new` operator of the set element subclass.
   b. Call the set object's `locate` method to initiate a search.
   c. Insert the new element using the search result object's `insert` method.

Populating a Set through Actions

To populate a set through actions:

1. Pass the search name from the rule to the action function. The argument points to the search result object when the action code is executed.
2. In the action function:
   a. Create a new element using the `new` operator of the set element subclass.
   b. Use the search object's `insert` method to insert the new element into the set.

For example, suppose you use a set to keep track of all addresses from which you are receiving packets. When a search fails to find a packet's source address in the set, the action function creates a new element, with the packet's source address as a key, then inserts the new element in the set using the search result object's `insert` method. The following action function creates a net address element (`NetAddr`) and adds it to the `net` set:

```
ACTNF add_net_addr (Buffer* buf, CONNAce* ace,
                        Search net_srch, int flag,
                        unsigned ip_addr)
{
   // Create a new set element object
   NetAddr *new_addr = new NetAddr(ip_addr);
   if (new_addr == 0) {
   printf ("\nERROR while creating a new NET entry \n");
   }
   // Insert at appropriate location pointed by Search object
   net_srch.insert (new_addr);
   return RULE_CONT;
}
```

How to Use Sets and Searches

The Policy Accelerator executes searches when it parses the NCL file for an incoming packet. 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 corresponding ASL `Search` object.

You can also use the `Set_`*`setname`* object's `locate` method to initiate a search anywhere in your action code. For example, you might want to initiate a search in response to an event or a message.

Using Searches in Rules

You can refer to a search by name 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 name acts as a Boolean expression. It succeeds when the search was executed and found a matching record in the set. If the search was not executed because the requirements were not met, or if it was executed but failed to find a matching element, the expression evaluates to `FALSE`.
- When you pass a search name as an action argument on the right side of a rule, the action can use the search result object to obtain a pointer. When the search has succeeded, call the `toElement` method to get a pointer to the matching record in the set. When it has failed, call the `insert` method to insert a new element in the set at the location where a matching element would have been found.

Setting and Comparing Key Values

Keys are stored as network-byte-ordered 32-bit integers (`nuint32`). 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.

If you are passing data in a downcall that will be used to create a new set element or search for an existing one, it is a good idea to store items in your message as `nuint32`. In general, use `nuint16` and `nuint32` when sending multibyte values in upcalls and downcalls, in order to preserve the byte order of the data.

For more information, see “Byte Order Issues” on page 10 of the *IX-API SDK Reference.*

Using Actions to Modify Sets

You can take actions based on the fact of a search succeeding or failing, without regard for the data portion of the set element. For example, an action can simply set the target for the packet based on whether one of the searches succeeds.

An action function can also modify a set element by changing its data. For example, when you have identified a packet that is coming from one of the interesting addresses, you can increment a counter in the matching element.

You can look at or copy the data in a matching element, and take actions based on that data. For example, after a packet comes in from a particular address and you have incremented the counter in the matching element, you can check whether the counter has reached a certain value. If it has, you can build a message from it and send it to the host application in an upcall.

An action function can create a new set element, using the search result object’s `insert` method. It can also delete a set element, directly or after a delay. To delay an action on an element, you can call its `expire` method. The action you specify in an expiration callback is then executed after an amount of time that you specify.

Setting Element Expiration

Set elements can expire. The `Elt_`*`setname`* subclass has an `expire` method that sets an expiration time and a callback function to be executed after that amount of time has elapsed. Typically, the expiration callback function deletes the element from the set, using the `delete` operator for the `Elt_`*`setname`* subclass.

An action function can call the `expire` method to delay the action, or to make it provisional. For example, you might want to delete the element only if no more matching packets come in within a certain amount of time. If another matching packet comes in, the action can cancel or reset the expiration timer.

The meaning of expiration is up to you. Deletion is not automatic, but occurs only if you call the `delete` method in the callback you define to handle the expiration. You can provide more than one expiration callback, and specify a new one when you call the `expire` method.

The time of an expiration is always interpreted as an amount of time elapsed after the method is called. You cannot specify an absolute time.

For more information on expiration, see the description of the "`Elt_setname` Class" on page 160 in the *IX-API SDK Reference.*

Deleting Sets

Before you delete a set, you should delete any remaining elements. The `Set` class contains iteration methods, `first` and `next`, which you can use to walk through all elements of the set.

You can define the destructor of your customized set subclass to use the iterator functions to clean up set elements. For example, the following code fragment would be part of the destructor of a set subclass named `Set_myset`, with elements that belong to the subclass `Elt_myelement`.

```
Elt_myelement * scan;
Elt_myelement * next;
...
scan = first();
while (scan != NULL) {
   next = scan->next();
   delete scan;
   scan = next;
}
```

If the set is defined as part of the ACE, perform this cleanup in the destructor of the `Ace` subclass. In this case, you must use an object reference for the `first` method:

```
Elt_myelement * scan;
Elt_myelement * next;
...
scan = Set_myset.first();
while (scan != NULL) {
   next = scan->next();
   delete scan;
   scan = next;
}
```

Chapter 10
Finding Strings in Packets

This chapter describes the string search facility, which you can use to find specific strings or string patterns in the data portion of packets.

This chapter contains the following topics:

- Overview of String Searches
- Setting Up a String Search
- Initiating and Continuing Searches
- Changing Search Parameters
- Acting on Search Results
- Disposing of Packet Buffers After a String Search

Overview of String Searches

The ASL provides a high-performance string search facility that allows you to search for strings contained in the data portion of packets:

- You can search for occurances of a constant string, or for strings that match a regular expression.
- You can search 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.
- You can continue a search for a particular string or set of strings across packet boundaries, to find a string that is contained in more than one packet.

There are two ASL classes that control a string search:

- The search engine object (`NBStringSearchEngine`) contains the list of strings or patterns for which to search. It also contains the method `SearchBuffer` that initiates a search or sends a new packet buffer to an ongoing search.

- The context object (`NBSearchContext`) keeps track of ongoing searches. It contains the callbacks that you define to respond to the search results.

A third class, `NBStringStringMatchReport`, contains search result information. You can use this class in your callback function to find and act on the strings found by your search.

For details of the string search classes and their methods, see "String Search Classes" on page 98 of the *IX-API SDK Reference*.

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

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

Setting Up a String Search

When you plan to use the string search facility in an ACE, your application must do the following:

- Specify the `ACE_STRINGSEARCH` flag in the `argAceMode` argument when you construct the `AceManager` object in the host module for that ACE.
- Include a reference to the string search context and engine objects in the ACE object in the PE module.
- Create at least one engine object to contain the list of search strings and initiate string searches in specific packet buffers.
- Create a search context object for each search. For example, for a string search in a TCP stream, you would create two context objects, one for each direction of the stream.

  A search associated with a particular context object can be continued into multiple packets. When a search is completed, you can reset and reuse the context object for another search.

For example, the following ACE subclass definition includes references to string 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 constructor for the ACE creates the engine object:

```
CGetPkt::CGetPkt (ModuleId id, char* name, Image* obj):
Ace (id, name, obj)
,str_engine( this )
{...}
```

In this example, the string search is initiated as part of a set search, so the set element class has a method for creating 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);
...}
```

Initiating and Continuing Searches

To create a new string search:

1. Create a new context object (or reset an existing context object).
2. Set the options for the new search using the context object's `SetOpt` method.
   - Set the `NBS_OPT_PERSTR` option if you have defined a per-match callback to deal with each matching string that is found. See "Acting on Search Results" on page 131.
   - Set the `NBS_OPT_SIMPLE` option if you want to search in only one packet.
3. Add the string or strings you want to search for to the list in the search engine object. Use its `AddString` and `DeleteString` methods to maintain this list. The engine must be in maintenance mode to do this; see "Changing Search Parameters" on page 130.
4. Put the search engine object into search mode by calling the `ChangeOpMode` method.
5. Start the new search by calling the `SearchBuffer` method of the search engine object. Pass the current packet buffer along with the new or reset context object. Typically, an action function calls this method.

   If the `NBS_OPT_SIMPLE` option is false (0), the context object will maintain the search state when the search reaches the end of the packet buffer, and you can send additional packets to the same search. You would do this to find a string that might cross packet boundaries, starting in the data portion of one packet and continuing into the data portion of the next packet.

6. To continue the search into a new packet, call the `SearchBuffer` method of the search engine object again, passing the next packet buffer along with the context object for the search.

You can maintain multiple searches simultaneously, as long as each search is associated with its own context object.

To end an ongoing search, delete or reset its context object.

- To delete a context object, call its `SchedDelete` method.
- To reset a context object, call its `SchedReset` method.

The search engine completes any search currently in progress using that context before it takes the requested action.

Changing Search Parameters

To make sure that the list of search strings does not change while a search is in progress, the search engine has a maintenance mode and a search mode. You can only change search parameters when the engine is in maintenance mode.

Before you try to set or change any search parameters:

1. Determine the current operating mode by calling the engine object's `OpMode` method. (When you first create the object, it is in maintenance mode.)
2. If the engine is in search mode, request maintenance mode by calling the engine object's `ChangeOpMode` method.

   It may take some time to complete any current search and change the operating mode. 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 you provide with the method call.
3. Once the engine is in maintenance mode, use the `AddString` and `DeleteString` methods to change the list of search strings.
4. When you have finished making changes, restore the search mode by calling the engine object's `ChangeOpMode` method again.

Acting on Search Results

You provide callback functions to act on the results of the search, to be invoked each time a matching string is found or each time a search is completed for a buffer. There are two ways to act on the search results:

- If you set the per-match option, the search engine invokes the per-match callback every time it finds a matching string. You provide a per-match callback function to take whatever actions you want on each matching string as it is found.
- You can create a match report object, that collects all of the matching strings found in a particular packet buffer. When the search is complete in that buffer, your per-buffer callback can retrieve the matching strings from the match report object and take whatever actions you want on them.

Per-Match Callbacks

To handle matching strings on a per-match basis:

1. Set the per-match option by calling the context object's `SetOpt` method.
2. Define a per-match callback function to handle each matching string as it is found.
3. Associate the callback function with the context by calling the context object's `SetPerMatchCallback` method.

The search engine passes information about the matching string to the callback function. The callback returns a value that tells the search engine whether to continue finding and reporting string matches in the current packet buffer, or to quit searching in the current packet buffer.

The following example shows a per-match callback that identifies which search string was matched, adds to a counter of matches for that search string, and adds the length of the matching string to another counter. This example returns the value `NBS_CONT`, which tells the search engine to continue.

```
int MyStrSearchCtx::OnEveryMatch (void *ace, void *elt1,
                                  Buffer *buf,
                                  NBStringID sid,
                                  void *stringtag,
                                  int endoffset,
                                  int matchlen,
                                  char *payload  )
{
   int host_id = ((int) stringtag);
   // update the global report
   ((CGetPkt*)ace)->report.stringidcount++;
   // update for this string id stream
```

```
    ((CGetPkt*)ace)->report.stringidfound[ host_id ]++;
    ((CGetPkt*)ace)->report.totalbytesmatched += matchlen;
    return NBS_CONT;
}
```

Per-Buffer Callbacks and Match Reports

If you choose to handle all of the matching strings in a packet buffer at one time, you must create a match report object to hold the information about the matching strings, so that you can access it in your per-buffer callback function.

To handle matching strings on a per-buffer basis:

1. Create an `NBStringMatchReport` object.
2. Define a per-buffer callback function that accesses matching string information by using the methods of the match report object.
3. Associate the callback function with the context by calling the context object's `SetPerBufCallback` method.
4. Pass the match report object to the search engine object's `SearchBuffer` method when you initiate the search (or send the current buffer to an ongoing search).

When you start a string search with a match report object, the search engine generates a match report for each match that it finds, and stores it in an array in the report object.

When the search is complete for the packet buffer, the search engine passes the match report object back to the per-buffer callback. The object points to the string match report array. Your callback function can retrieve information about a particular matching string by indexing into the array. The first match found is at index 0.

The information you can get about a matching string using the match report object is the same information that is passed to the per-match callback, such as the identifier for the search string that was matched, the number of bytes in the matching string, and its location in the packet buffer.

Disposing of Packet Buffers After a String Search

The string search engine always invokes the per-buffer callback when it completes a string search in a particular packet buffer. Regardless of which method you choose for handling the matching strings, your per-buffer callback should at least determine how to dispose of the buffer. If other parts of your

application are also interested in the packet, the per-buffer callback is the place to decrement the reference counter, to indicate that this part of the code is finished with the buffer.

The following example shows a per-buffer callback function that disposes of the packet buffer by sending it to the default drop target, and adds to a count of packet buffers that have been searched. In addition, if a global value indicates that the buffer was reference-counted (and thus possibly being used elsewhere), this function decrements the reference count.

```
void MyStrSearchCtx::OnEveryBuffer (void *ace, void *elt,
                                    Buffer *buf,
                                    NBStringMatchReport *rp)
{
   ((CGetPkt*)ace)->drop (buf); //get rid of buffer
   ((CGetPkt*)ace)->report.gotback++;
#ifdef REFCOUNT
    buf->decref();
#endif }
```

Chapter 11
Debugging and Troubleshooting

This chapter provides information on debugging your code. It contains the following topics:

- Debugging Host Module Code
- Using the IX-API SDK Debugger
- Debugging Action Code
- Runtime Troubleshooting Hints

Debugging Host Module Code

You can use `fprintf` and `trace` functions to display debugging messages in the host module of your application.

Using Tracing in Your Host Application

Use the function `nb_trace_verbose` to turn on tracing in your host application. The function takes a single argument, *`mode`*, which is either `1` to turn tracing on, or `0` to turn it off:

```
nb_trace_verbose (mode )
```

This function prints to `STDOUT`, displaying the code currently being executed in the same window where the application is running.

If the application hangs, you can refer to the trace to find the last function that was executed. You can then use the debugger to step through that function.

The following code fragment employs `nb_trace_verbose`:

```
void main (int argc, char** argv) {
   nb_trace_verbose (1);
   ...
}
```

The nberror.h file provides additional functions and macros for tracing in the host module code. You can insert trace macros in the code to print particular messages when that line is reached.

Debugging Short-Running Applications

Normally, applications run continuously. However, if you must debug the host side of an application that completes execution quickly, you must insert a pause into the application logic. You can, for example, insert something like the following code to halt the application's execution until the user presses a key:

```
static int count = 0;
   if (count == 0)
   {
      char key[127];
      key[0] = '\0';
      printf("hit any key to continue...\n");
      gets(key);
      count = 1;
   }
```

Using the IX-API SDK Debugger

The IX-API SDK provides a debugger 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 IX-API SDK debug daughter card attached to your Policy Accelerator. In addition, you must have the entire SDK installed on the computer on which you are running the debugger.

This section contains information supplementary to the *GDB User's Guide*. Refer to the *GDB User's Guide* in the /usr/local/docs directory for documentation on supported debugging commands.

The following gdb commands are *not* supported in nbgdb:

- run, load, and related commands
- Commands related to watchpoints and tracepoints
- Thread- and task-related commands

To use the debugger, you must compile all action code with the gcc debug option (-g), and start the Resolver in verbose mode using the -v switch.

CAUTION: With optimizations of any level turned on, the debugging information generated by the compiler might be inaccurate. Turn optimization off during the early stages of development and debugging of an application.

Makefile Debugging Flag

If you use the demonstration makefiles, you can set all the appropriate flags and libraries for both the host module and the accelerator module using the BUILD_MODE variable as follows for debug mode:

```
For Windows NT: nmake BUILD_MODE=Debug
For UNIX:       make BUILD_MODE=Debug
```

or as follows for nondebug mode:

```
For Windows NT: nmake BUILD_MODE=Release
For UNIX:       make BUILD_MODE=Release
```

Review the comments about debug mode in the makefile in Chapter 4, “Compiling Applications.”

Debugging Tools

You can set debugging flags when compiling code and when starting the IX-API SDK software with the `resolver` command. In addition, you can use command-line debugging tools.

- To debug host module application code, use the standard GNU debugger, `gdb`. For information on using `gdb`, see the *GDB User’s Guide* in the `/usr/local/docs` directory.
- To debug action functions in the accelerator module, the IX-API SDK provides a customized version of the GNU debugger, `nbgdb`. You use a command-line utility, `getaceid`, to create a standalone version of the executable for an ACE on which to run the debugger.

Producing a Debugger Executable

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 debugger executable for that ACE that you can run on the host from a command shell.

To produce a debugger executable for an ACE, use the utility `getaceid`. In addition to finding and returning the ID for an ACE, it produces a debugger executable for that ACE 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 `getaceid` command takes the following arguments:

`getaceid` *`ACEpathname actionfile_basename`*

- The first argument, *`ACEpathname,`* is the complete path to the ACE you want to debug. You can find this path in the `bind` function of your application code (*`appname.`*`cpp`). (See "Naming Objects for the Resolver" on page 55.)
- The second argument, *`actionfile_basename,`* is the file name of the compiled action code for the ACE (without its extension, `.nbo`).

The command displays the ACE's ID (an integer) and also creates a file *`ACEname.`*`exe` in the current directory. To run the debugger on this file, execute a command such as the following in that directory:

```
nbgdb myace.exe
```

You must then connect to the target ACE, using the ID returned by `getaceid`. For example:

```
(nbgdb) target remote com1:11
```

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 a debugger executable on the host, rather than on the Policy Accelerator itself, debugging information is available only for action functions, which are invoked as rules are applied to packets received. 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: The SDK does not support debugging of the action code initialization function, `init_actions`. Do not introduce a breakpoint or other pause into this function.

Debugging Action Code

This section describes how to start and stop the debugger and illustrates what a debugging session might look like. The sample LoopApp test application is used as an example.

Connect the Debug Daughter Card

1. Power down your PC.
2. Attach the debug card and a free COM port on the PC using a DB9 null modem cable and a flat DB9 serial cable (as supplied by Intel).
3. Turn on your PC.

For more detailed information on connecting the debug card to your computer's COM port, see *Installing a Policy Accelerator 100 Board*.

Prepare the LoopApp Application

The LoopApp application normally runs just one loop with 256 repetitions and it finishes too quickly for you to run the debugger and to establish a breakpoint. However, you can modify it to wait for a keystroke before starting the loop, giving you a chance to start the debugger and to set breakpoints. To do this:

1. Locate the LoopApp source code in the *`SDKinstall-path`*`/demos/LoopApp/source` directory.
2. Open the action code file for the application, `loopact.cpp`.
3. Add the following code at the begining of the function `NBloopAce::packetUpcall`:

```
static int count = 0;
if (count == 0)
{
   char key[1];
   key[0] = '\0';

   printf("Go get the Ace ID(getaceid) now,
              then hit return to continue...\n");
   gets(key);
   count = 1;
}
```

4. Save the file.
5. Make the debug version of the application file. To do this, open a command shell, go to the `LoopApp/` directory, and invoke the makefile as required by your operating system.

   Under Windows NT:

```
c:\> cd \SDKinstallpath\demos\LoopApp
c:\> nmake BUILD_MODE=Debug
```

   Under UNIX:

```
% cd /SDKinstallpath/demos/LoopApp
% make BUILD_MODE=Debug
```

This compiles the code with the nbgcc debug option (-g) and without optimization options and writes the compiled file to the new Debug subdirectory.

Start your Debugging Session

1. In a command shell, start the resolver in verbose mode:

   ```
   c:\TEMP> resolver -v
   ```

2. In another command shell, go to the directory where the application objects reside:

   ```
   c:\> cd \SDKinstallpath\demos\LoopApp\Debug
   ```

3. Start the application:

   ```
   c:\SDKinstallpath\demos\LoopApp\Debug> loopapp
   ```

   A prompt appears in this shell and waits for a keystroke to continue.

4. In another command shell, use the utility getaceid in the same directory:

   ```
   c:\> cd \SDKinstallpath\demos\LoopApp
   c:\SDKinstallpath\demos\LoopApp> getaceid \
      /NBloopAppl/NBloopAceGroup/NBloopAce loopact
   ```

   Specify the ACE using its full name, as in your application. Here the ACE's full name is /NBloopAppl/NBloopAceGroup/NBloopAce. The command returns the ID of the ACE in the command shell, and also produces a debugger executable file (in this case loopact.exe) in the same directory.

5. Run the debugger, nbgdb, on the debugger executable file generated by getaceid:

   ```
   c:\SDKinstallpath\demos\LoopApp> nbgdb loopact.exe
   ```

6. Connect to the target ACE using the ID returned by getaceid:

   ```
   (nbgdb) target remote com1:11
   ```

   Here, com1 refers to the COM port on the host that is connected by null modem cable to the debugger card. The number 11 is the ID returned by getaceid.

Step through Code

After you have started the debugger and connected to the ACE, use the standard debugging commands to set breakpoints and step through your code. For example, you might use the following sequence of commands:

1. Set a breakpoint:

   ```
   (nbgdb) break action_all
   ```

2. Use the `next` command to proceed to the next statement:

   ```
   (nbgdb) n
   ```

3. In the application's command shell, press a key to restart the application and run the loop.
4. In the debugger's commad shell, use the `step` command to step into the next function:

   ```
   (nbgdb) s
   ```

5. Use the `continue` command to continue execution when you reach the end of an action function:

   ```
   (nbgdb) c
   ```

Shut down the Debugger

When you have finished stepping through your code, use the following sequence of commands to shut down the debugger:

1. Use the `delete` command to delete all breakpoints:

   ```
   (nbgdb) delete
   ```

NOTE: You *must* delete breakpoints before detaching.

2. When you have deleted all breakpoints, use the `detach` command to detach from the system:

   ```
   (nbgdb) detach
   ```

3. Use the `quit` command to end your debugger session and complete execution of `nbgdb`:

   ```
   (nbgdb) quit
   ```

Runtime Troubleshooting Hints

This section describes some common coding errors which can result in runtime exceptions or failures that can be difficult to interpret.

Problems with Paired Object Naming

Paired objects, such as upcalls and upcall handlers or targets and target managers, must have the same dictionary name. Errors in naming do not cause compilation errors, but do cause runtime errors when the system uses the dictionary to try to find the related object.

NOTE: Paired object dictionary names are case-sensitive. Be sure that paired objects have identically spelled and capitalized dictionary names.

For example, if you spell or capitalize the dictionary name of an upcall differently than that of its associated upcall handler, the constructor for the upcall does not fail. An `Upcall` object is created with the misspelled name. You cannot detect the problem until you issue the related call. When you issue the call at run time, the mismatched names cannot be resolved, and the call fails.

Problems with Sets

The ACE object for an ACE that uses a set must contain the set object. The set object must have the same name that was assigned to the set when it was defined in the NCL file. If a properly named set object does not exist, you will see errors similar to the following:

```
fixce: ace '/CONN/CONNAceGroup/CONNAce' has no Set called 'nets'
fixce: unable to resolve relocation for '__NB_Set_nets'
```

For a description and example of the correct way to create a set object and avoid such errors, see “Creating a Set Object” on page 121.

Problems with Action Function Return Values

If an action function you have defined does not return an acceptable action function return value, you get a compiler warning. If you ignore this warning, your application is corrupted when the function returns, and the corruption may not be detectable. See “Defining Action Functions” on page 96 for more information.

Problems with Action Function Arguments

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. Passing a Boolean argument to an action function can cause a runtime alignment exception.

Reading Output from the Accelerator Module

To see messages printed to `stdout` and `stderr` from the accelerator module of your application, use the provided I/O utility appropriate to your operating system:

- On a UNIX system, use the command-line utility `readport`, in the directory `$NBPATH/bin`. Execute the command in a command shell, specifying the port number (11 or 12).
  - 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.

- 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.

Starting and Stopping 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.

Chapter 12
Delivering Applications

This chapter describes how to package the IX-API SDK run-time files to deliver to your end users along with your applications for the Policy Accelerator.

NOTE: This chapter describes run-time files for Windows NT. There is currently no run-time-only installation for UNIX. The run-time files are included in the SDK installation.

Overview

For your Policy Accelerator application to run, the IX-API SDK run-time files must be installed on the same system. You must decide how you want your end users to receive and run the IX-API SDK run-time installer:

- You can invoke it from your application's installer
- You can deliver it separately from your application's installer and direct the end user to run it
- You can direct the end user to download it from the Intel Web site

Installing the Run-Time Files From Your Own Media

This section describes how to include the run-time installer on your own media, such as CD-ROMs.

The media that you used when installing the IX-API SDK software on your system includes the following directory structure:

```
IX SDK Runtime Install/
   Install Files/
   Doc Files/
      frame/
      pdf/
```

To include the run-time installer on your media:

1. Include all files and subdirectories from the `Install Files/` subdirectory.

   You can place these files together in any location; they do not have to be in a directory named `Install Files/`.

2. Adjust your application's installer, or direct the end user, to run the following file from the set of included files.

   This is an Install Shield script that installs all other files from the subdirectory into a directory named, by default, `NetBoost/` at the system's root directory:

   ```
   setup.exe
   ```

NOTE: Your application's installer must install—or verify the existence of—`mfc42.dll`.

3. Provide instructions for running the `setup.exe` program.

   To do this, either:

   - Include the `Doc Files/pdf/*.pdf` file(s). PDF files can be viewed using Adobe Acrobat Reader[1].
   - Modify the FrameMaker source files, included in `Doc Files/frame/`, for your own purposes and include a readable version of the revised instructions, such as PDF or hardcopy.

You can, and in most cases will, provide all files without modification. If instructions on customizing the installer are available, they will be on the Intel Web site, www.intel.com.

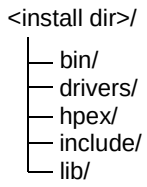
Installation Results

This section describes the directories and files that are created on your end user's system by the installer.

1. Adobe, Acrobat Reader, and FrameMaker are trademarks of Adobe Systems Incorporated.

IX-API SDK Run-Time Tree

The basic directory structure is shown in the following figure and table:



Directory	Description
`bin`	Contains all the basic executable files and tools that run on the host.
`drivers`	Contains all the files needed to install the Policy Accelerator drivers.
`hpex`	Contains all the basic executable files and tools that run on the Policy Accelerator.
`include`	Contains all the IX-API SDK standard protocol files.
`lib`	Contains libraries used on the host and libraries used on the Policy Accelerator.

Environment Variables

The installer prompts the user for where to install the product and sets the following environment variables based on that response:

Variable	Description
`NBCONFIG`	Points to the path of the filename containing the configuration information for the Resolver.
`NBPATH`	Points to the IX-API SDK tree root. The default on a Windows NT system is `C:\IXSDK`

Directory bin

The `bin` directory contains the following files:

Category	File	Description
NCL compiler	`cecomp.exe`	Network Classification Language (NCL) compiler front end
	`celink.exe`	Microcode linker
	`cemasm.exe`	Microcode assembler for NCL
	`concat.exe`	Concatenate `.masm` files
	`hext.exe`	Produce hexadecimal representation of microcode binary

Category	File	Description
IX-API SDK system management	`resolver.exe`	IX-API SDK resource manager (Resolver)
	`NBControlService.exe`	Resolver control service
	`NBInstallResolverSvc.dll`	Resolver installation
	`NBLauncherSvc.exe`	Resolver application launcher
	`NBUninstDrv.exe`	IX-API SDK driver removal
Console control	`readport.exe`	Display Policy Accelerator console output
	`writeport.exe`	Console input program
	`WinReadPort.exe`	Console display for multiple Policy Accelerators
Debugger	`nbgdb.exe`	Action code debugger

Directory drivers

The `drivers` directory contains the following files:

Category	File	Description
Installation	`NBDC.exe`	Installation applets
	`Oemsetup.inf`	Driver installation description
Drivers	`nbether.sys`	Network driver interface specification (NDIS) driver for the Policy Accelerator
	`nbhwpe.sys`	Kernel driver for the Policy Accelerator
	`nboost.sys`	API driver for the Policy Accelerator

Directory hpex

The hpex directory contains the following files, which support the Policy Accelerator hardware, including its classification engine and StrongARM[2] components:

Category	File	Description
Policy Accelerator system software loading tools	`NBChkRev.exe`	Check Policy Accelerator hardware Revision
Policy Accelerator system initialization	`arminit-nc.img`	
	`arminit.img`	Basic StrongARM initialization code
Code for Policy Accelerator system services	`ha_ctl.armeb-coff.nbo`	ARM object for control Action/Classification Engine (ACE)
	`ha_istk.armeb-coff.nbo`	ARM object for input stack ACE
	`ha_ostk.armeb-coff.nbo`	ARM object for output stack ACE
	`ha_rxa.armeb-coff.nbo`	ARM object for receive interface ACE A
	`ha_rxb.armeb-coff.nbo`	ARM object for receive interface ACE B
	`ha_txa.armeb-coff.nbo`	ARM object for transmit interface ACE A
	`ha_txb.armeb-coff.nbo`	ARM object for transmit interface ACE B
	`stringsearchactions.armeb-coff.nbo`	ARM object code for string-search engine

2. StrongARM is a registered trademark of ARM Limited.

Category	File	Description
Policy Accelerator system software	`hPEx.exe`	Policy Accelerator system object for hardware Revision 2
	`hPEx.img`	
	`symtbl.img`	Policy Accelerator Action Services Library (ASL) symbol table for hardware Revision 2
	`hPEx_rev1.exe`	Policy Accelerator system object for hardware Revision 1
	`hPEx_rev1.img`	
	`symtbl_rev1.img`	Policy Accelerator ASL symbol table for hardware Revision 1
Microcode	`nocecode.ncl`	Microcode for idle classification engines (CEs)
	`nxrt0.img`	Microcode for CE0 when in plain NIC mode
	`nxrt1.img`	Microcode for CE1 when in plain NIC mode
	`nxrt2.img`	Microcode for CE2 when in plain NIC mode
	`nxrt3.img`	Microcode for CE3 when in plain NIC mode
Power-on self test	`Post.img`	Power-on self-test image
Debugger support	`stalker-data.img`	Debugger stub data image
	`stalker-text.img`	Debugger stub text image

Directory include

The `include` directory contains the following files:

Category	File	Description
Protocol include files	`NBNcl/NBbase.ncl`	IX-API SDK base protocol NCL specification
	`NBNcl/NBtcpip.ncl`	TCP/IP NCL specification

Directory lib The `lib` directory contains the following files:

Category	File	Description
NCL compiler	`cecomp.dll`	NCL compiler
	`celink.dll`	Microcode linker
	`cemasm.dll`	Microcode assembler
	`ncldrvr.dll`	NCL compiler driver
Cygwin support library	`cygwin1.dll`	Cygwin support library
Boot support	`boot.dll`	Policy Accelerator boot support
Microcode	`ipchksum.o`	IP checksum microcode
	`tcp_udp_chksum.o`	TCP, UDP checksum microcode
	`keylibIndex.o`	Shared set microcode
	`keylibCommon.`*`[1-7]`*`.o`	Common set microcode for each supported number of set keys: 1 to 7
	`keylibEntry.`*`[1-7]`*`.`*`[10-16]`*`.o`	Set entry microcode for each supported number of set keys and each hash table size: ■ Keys: 1 to 7 ■ Hash table size: 2^{10} to 2^{16}
	`keylibInline.`*`[1-7]`*`.`*`[10-16]`*`.o`	Set main microcode for each supported number of set keys and each hash table size: ■ Keys: 1 to 7 ■ Hash table size: 2^{10} to 2^{16}
	`mrt`*`[0-3]`*`.masm`	Classification engine (CE) run-time microcode for each CE, 0 to 3
	`mrt`*`[0-3]`*`.o`	CE run-time microcode object for each CE, 0 to 3
Host API (NBAPI) support	`nbapi.dll`	Main host API library
	`nbapid.dll`	Main host API library, debug mode
	`nbconfig.dll`	Library to read config file

Appendix A
Demonstration Applications

This appendix describes the sample applications that are included in the IX-API SDK distribution. The following table briefly describes the techniques and concepts demonstrated by each of the sample applications. Each of the applications is described in more detail in the following sections.

Sample Application	Description
BasicApp	Shows how to construct the basic components of any application. Creates a single ACE that counts packets and uses an upcall to report the number of packets observed.
Simple	Shows how to use rules to classify packets of different types or with specific source or destination addresses.
FilterApp	Shows how to construct a simple MAC address packet filter. An extension of BasicApp, in which only packets with a specific MAC destination address are forwarded. The ACE accepts packets from one MAC interface and forwards those that pass the filter criteria to the other MAC interface.
ODXFilter	Shows how to transfer packets to and from a NIC whose driver has been customized using the Optimal Data Exchange (ODX) Protocol for PCI. Extends FilterApp to filter packets according to protocol before sending them to the NIC or dropping them.
EventAppl	Shows how to use the `Event` class to schedule an activity that is not associated with a data set.
TwoAceApp	Shows how to create and use an application with multiple ACEs. Extends BasicApp by making two separate ACEs in the same application that keep different counts.
FilterNic	Shows how to emulate a network interface card (NIC) in software. Each of the two ACEs accepts packets from one of the MAC interfaces, filters them, and sends those that pass the filter criteria to the host.

Sample Application	Description
Tap	Shows how to construct a tap, in which the ACE “sniffs” a connection, looking at packets as they go by without redirecting them. In this case, the ACE keeps statistics about the packet flow.
IPPairs	Shows how to use sets and searches to maintain a database about packet traffic. The application maintains a set of source–destination address pairs for IP packets that it has seen recently, and counts additional packets that match any known pair.
Killer	Shows how to construct an application that starts and stops the Resolver automatically. Only one such application can run on a Policy Accelerator at one time.
LoopApp	Shows how to construct a test to ensure that the IX-API SDK operating environment is configured correctly.
Firewall	Shows how to construct a real firewall application that determines which packets should be allowed into and out of the local area network, based on criteria that a user can set.
Crosscall	Shows how to communicate between ACEs using crosscalls.
StringSearch	Shows how to search for a string in packet data.

Building the Sample Applications

The sample applications are included in the IX-API SDK distribution in the directory *SDKinstallpath*/demos. The directory contains:

- Three makefiles that are used to build the sample applications:
  - A master makefile, Makefile.inc, builds an application. This file is included by the individual makefiles for each sample.
  - Makefile.dist builds the entire set of samples by calling the individual makefiles for each.
  - Makefile does a master build for all sample applications. It calls Makefile.dist to build all of the samples, then installs the compiled files.
- A number of subdirectories that contain the applications. Each application directory contains:
  - A source directory with the source code and header files for the application. These generally include a host module source file (.cpp), related header files (.h), an action source file for each ACE (.cpp), and a classification source file for each ACE (.ncl).

- A makefile specific to the application that specifies which actual files and directories to use for this specific application. This file includes `Makefile.inc`. Running this makefile creates the compiled files from the source code.

BasicApp

This sample demonstrates how to construct the basic components of an application. It is a simple packet-counting application. The ACE object in the accelerator module maintains a counter that it increments as packets arrive, before it forwards them to the default pass target. When it counts a packet, it also sends an upcall to the host module, which displays a message showing the current packet count.

The application contains the following source files in the directory `demos/BasicApp/source`:

Source File	Contains
`basicApp.h`	Error constants and subclass definitions for the host module.
`basicApp.cpp`	The main function for the host module, object constructors and destructors, and utility functions, which include the upcall handler callback. The constructor for the application object creates the bindings.
`basicAppActions.cpp`	The initialization and action function for the accelerator module; ACE subclass declaration, constructor, and destructor; and the ACE method that counts packets, constructs a message, and sends an upcall.
`basicAppRules.ncl`	The classification rules for the accelerator module; in this case, a single rule that passes all packets to the single action function.

Simple

This sample is similar to BasicApp, extending it to show the use of a number of classification rules.

Like BasicApp, the ACE object in the accelerator module maintains a counter that it increments as packets arrive, before forwarding them on to the default pass target. When it counts a packet, it sends an upcall to the host module, which displays a message showing the current packet count.

In this case, however, before a packet reaches the rule that sends the upcall, it is submitted to a series of rules that determine what type of packet it is and check for specific source and destination addresses. Each time the packet satisfies the rule predicate, that rule's action function displays a simple message saying which rule matched, then passes the packet along to the next rule.

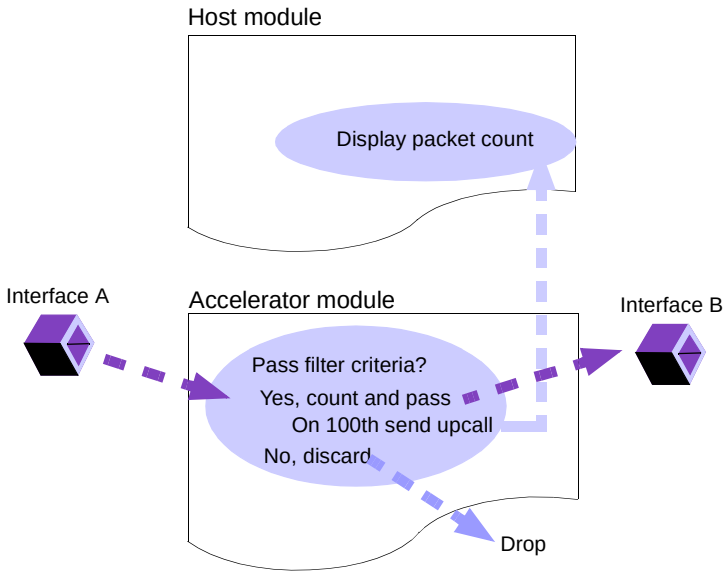
The application contains the following source files in the directory `demos/Simple/source`:

Source File	Contains
`simple.h`	Error constants and subclass definitions for the host module.
`simple.cpp`	The main function for the host module, object constructors and destructors, and the upcall handler callback that displays the packet count. The constructor for the application object creates the bindings.
`simpleactions.cpp`	The initialization and action functions for the accelerator module; ACE subclass declaration, constructor and destructor; and the ACE method that counts packets, constructs a message from the count, and sends an upcall.
`simplerules.ncl`	The classification rules for the accelerator module; in this case, several rules that classify the packet according to its type and source or destination address, as well as the final rule that passes all packets to the counter action function.

FilterApp

This sample is similar to BasicApp, but extends it slightly to apply a filter to incoming packets. Like BasicApp, the accelerator module maintains a packet count, but counts only those packets that meet the filter criteria—in this case, packets with a specific MAC destination. It reports the count to the host module using an upcall. Rather than reporting every packet, it reports only when the count is divisible by 100. When it receives the upcall, the host module displays a message showing the current count.

The ACE also filters the matching packets out of the traffic flow by sending them to the default drop target. Its sends all other packets to the pass target. This filtering function is controlled by a flag, which you can turn off by sending a downcall from the host. When the flag is on, matching packets are dropped. When it is off, all packets are passed.



In this application, all packets are received on interface A, and the pass target is bound to interface B.

The application contains the following source files in the directory `demos/FilterApp/source`:

Source File	Contains
`filterApp.h`	Error constants and subclass definitions for the host module.
`filterApp.cpp`	The main function for the host module, object constructors and destructors, the upcall handler callback, and an ACE manager method that creates a message and sends a downcall. The constructor for the application object binds the in and out targets to the two MAC interfaces.
`filterAppActions.cpp`	The initialization and two action functions for the accelerator module; ACE subclass declaration, constructor, and destructor; the ACE method that constructs a message and sends an upcall; and the downcall handler callback.
`filterAppRules.ncl`	The classification rules for the accelerator module; in this case, protocol definitions for `base` and `ether`, and two rules. One sends packets that match the filter criteria to the action function `action_match`. The other sends all packets to `action_other`.

ODXFilter

This sample demonstrates the C interface, which represents a NIC whose driver has been customized for communication with the Policy Accelerator using the Optimal Data Exchange (ODX) Protocol for PCI. The application ACE's pass target is bound to the system ACE for the C interface, and the C interface pass target is bound to the application ACE.

This application is similar to FilterApp, but extends it to filter incoming packets according to protocol. An action function for each type of protocol maintains a packet count, and forwards undamaged packets to the NIC. The ACE filters bad packets out of the traffic flow by sending them to the default drop target, after counting and reporting on them.

NOTE: This is a usage demonstration and is not intended to test your customized NIC driver. A diagnostic utility to test the NIC driver customization is included in the installation at the following location:

SDKinstallpath/diagnostics/odxloop

The application contains the following source files in the directory demos/ODXFilter/source:

Source File	Contains
ODXFilterAppl.cpp	The main function for the host module, subclass definitions and object constructors and destructors. The constructor for the application object binds the in and out targets to the C interface.
ODXFilterActions.cpp	The initialization and action functions for the accelerator module; ACE subclass declaration, constructor, and destructor.
ODXFilterRules.ncl	The classification rules for the accelerator module; includes the predefined TCP/IP protocol definitions. Rules send packets to the action functions according to their protocols.

EventAppl

This sample, like SimpleApp, counts and reports on packets of different types. It uses an Event object to schedule the reporting function to occur every six seconds (rather than every 100 packets, as in FilterApp). As in SimpleApp, the rules classify packets of different types, and send them to action functions that count them. The counters are kept in the Event object, which, when the time is up, reports the packet counts to the host using an upcall. The host, upon receiving the upcall, displays the packet count information using standard output.

This sample shows how to use the Event class to schedule events that are not associated with data sets. This is much more efficient than, for example, creating a data set element for the sole purpose of having it expire.

The application contains the following source files in the directory `demos/Events/source`:

Source File	Contains
`eventsAppl.h`	Error constants and subclass definitions for the host module.
`eventsAppl.cpp`	The main function for the host module, object constructors and destructors, and the upcall handler callback acts on the message by displaying the count. The constructor for the application object binds the in and out targets to the two MAC interfaces.
`eventsACT.cpp`	The initialization and action functions for the accelerator module; ACE and `Event` subclass declarations, constructors, and destructors. Creates and initializes the event object, which contains the counters, the countdown timer, and a pointer to the upcall object. The Event object also contains the event callback method which, when the six seconds are up, constructs a message and sends it in an upcall, then reinitializes the counters and timer.
`eventsNCL.ncl`	The classification rules for the accelerator module. Includes the predefined TCP protocol definitions. Defines rules to classify packets of different types, sending each to an action function that counts that type of packet.

TwoAceApp

This sample combines the functions of BasicApp and FilterApp into one application with two ACEs. One ACE counts and forwards all packets and periodically reports the count in an upcall. The other ACE counts and forwards only those packets with a specific MAC destination, reporting the count periodically in an upcall.

This application differs from the BasicApp and FilterApp in that each ACE also has a variable reporting period, which the application can change by sending a new value in a downcall.

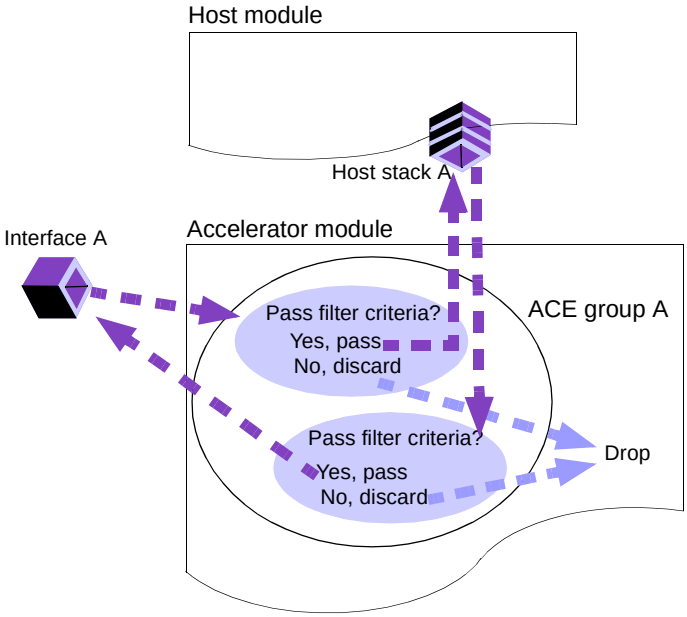
In this case, the host module defines two different upcall handlers and callbacks, to display messages for the two different kinds of upcall it can receive. The host module also defines the downcalls that the application can use to reset the reporting period and filtering flag. Each ACE defines corresponding downcall handlers.

The application contains the following source files in the directory `demos/TwoAceApp/source`:

Source File	Contains
`twoAceApp.h`	Error constants and subclass definitions for the host module.
`twoAceApp.cpp`	The main function for the host module, object constructors and destructors, the upcall handler callback, and an ACE manager method that creates a message and sends a downcall. The constructor for the application object creates the bindings, even though this application has a factory and you could create the bindings with the Plumber.
`aceOneActions.cpp`	The initialization and two action functions for the first ACE's accelerator module; ACE subclass declaration, constructor, and destructor; the ACE method that constructs a message and sends an upcall; and the downcall handler object and callback.
`aceOneRules.ncl`	The classification rules for the first ACE's accelerator module; in this case, a single rule that passes all `ether` packets to the single action function.
`aceTwoActions.cpp`	The initialization and action function for the second ACE's accelerator module; ACE subclass declaration, constructor, and destructor; the ACE method that constructs a message and sends an upcall; and two downcall handler objects and callbacks.
`aceTwoRules.ncl`	The classification rules for the second ACE's accelerator module; in this case, a single rule that passes all packets to the single action function.

FilterNic

This sample is similar to FilterApp, but uses different bindings to demonstrate software emulation of a network interface card (NIC). The application defines two ACE groups, each containing two ACEs. One ACE acts as a filter between packets arriving on a MAC interface and the host. The other filters packets going out from the stack to the interface.



The ACEs in group A filter packets between interface A and stack A. The ACEs in group B filter packets between interface B and stack B.

Like FilterApp, the ACE filters non-matching packets out of the traffic flow by sending them to the default drop target. It sends all other packets to the pass target, which in this case is bound to a host stack for incoming packets, or to the Policy Accelerator interface for outgoing packets.

This sample illustrates the bindings that allow this type of filtering, but does not do any actual filtering. The rules pass all IP packets. You can define rules to set criteria for either acceptance or rejection. That is, you can pass packets that match the criteria and drop those that fail, or drop packets that match and pass all others.

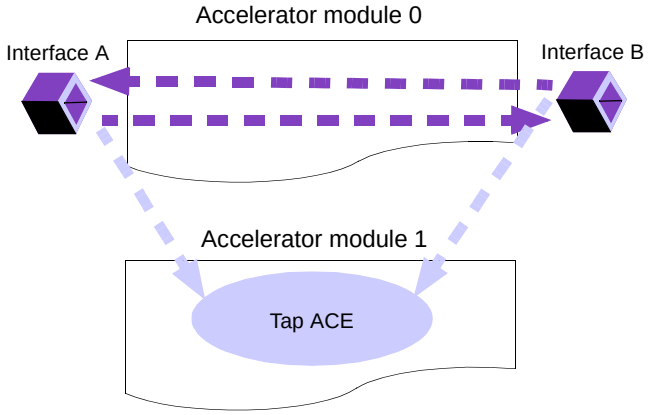
The application contains the following source files in the directory `demos/FilterNic/source`:

Source File	Contains
`filternic.h`	Error constants and subclass definitions for the host module.
`filternic.cpp`	The main function for the host module, object constructors and destructors. There are four ACEs, with two in each ACE group. The constructor for the application object binds the from target of one ACE in each group to one of the MAC interfaces, and its pass target to the corresponding host stack ACE. For the other ACE in the group, the from target is bound to the stack and the pass target to the interface.
`filternicactions.cpp`	The initialization and two action functions for the accelerator module; ACE subclass declaration, constructor, and destructor.
`filternicrules.ncl`	The classification rules for the accelerator module; in this case, the predefined protocol definitions for TCP/IP and two rules. One sends packets that match the filter criteria to the action function `action_accept`. The other sends packets to `action_reject`. (In this example, no packets are rejected.)

Tap

This sample watches traffic flow without redirecting packets. In this case, the application retrieves buffers after the packets have been forwarded to their destination in order to gather statistics on the traffic flow. Retrieving the buffer avoids additional overhead that would result from copying the packet. This technique can also be used for tasks such as intrusion detection.

This application defines only one ACE, but is designed for a system with two Policy Accelerators. The application binds the system ACEs of the first Policy Accelerator directly to one another, binding the *from* interface in each to the *to* interface in the other. It then binds each *from* interface's pass target to the ACE in the second Policy Accelerator.



The system ACEs forward packets directly between the *from* and *to* interfaces without processing. After the packet has gone, the system ACE sends the same buffer to the pass target, where it is processed on the second Policy Accelerator. The processing does not interfere with the speed of traffic, since it takes place after the packet has gone, nor does it take extra time to copy the buffer.

The ACE simply counts packets, and sends an upcall to the host to report every 1000th packet. The host's upcall handler displays the current packet count.

The application contains the following source files in the directory `demos/Tap/source`:

Source File	Contains
`tap.cpp`	Error constants and subclass definitions for the host module. The main function for the host module, object constructors and destructors, and the upcall handler callback The constructor for the application object binds the *from* and *to* interfaces of the system ACEs in the first Policy Accelerator directly to one another, then binds each *from* interface's pass target to the Tap ACE on the second Policy Accelerator.

Source File	Contains
`tapactions_stats.cpp`	The initialization function and single action function for the accelerator module; ACE subclass declaration, constructor, and destructor. The ACE contains an upcall, and a method that constructs a message and sends it in an upcall.
`taprules_stats.ncl`	The classification rules for the accelerator module; in this case, the predefined TCP/IP definitions, and a single rule that passes all packets.

IPPairs

This sample demonstrates the use of sets and searches to keep data. In this case, the set collects source–destination address pairs of all IP packets that come through, and keeps a count of how many additional packets come through for each pair. A search determines whether a current packet has an address pair matching any existing set element. If there is no match, the action function creates an element and adds it to the set. If a matching element exists, the action function increments the counter for that element.

To keep the size of the set manageable, an expiration timeout is set for each element and reset each time a matching packet comes through. If no matching packet is seen within the timeout period, the element expires and the expiration callback deletes it from the set.

The application contains the following source files in the directory `demos/IPPairs/source`:

Source File	Contains
`tellmsg.h`	A data structure definition to use in creating messages.
`IPAppl.cpp`	The main function for the host module. Error constants and subclass definitions for the host module. Object constructors and destructors, and utility functions, which include an upcall handler callback. The constructor for the application object creates the bindings.

Source File	Contains
`IPActions.cpp`	The initialization and action function for the accelerator module; ACE subclass declaration, constructor, and destructor; global variables and utility routines, including the expiration callback and a function that creates a message and sends an upcall. Includes the set header file created with the NCL compiler, and modifies the set element subclass to add data fields to it.
`IPRules.ncl`	The set and search definitions and the classification rules for the accelerator module. A single rule passes all IP packets to the single action function.

Killer

This sample is the same as SimpleApp, except that it starts and stops the Resolver automatically by using the operating system services layer (OSSL). After starting the Resolver process and the application, the main function allows the user to interrupt the application and kill the Resolver by pressing any key.

The OSSL is a utility that allows you to control processes independently of the operating system. It defines a class, `OSSLProcess`, to represent a process. This application shows how to create the Resolver process as an object of this class. You can use the object's methods to manipulate the process. Deleting the object kills the process.

The application contains the following source files in the directory `demos/Killer/source`:

Source File	Contains
`killer.h`	Error constants and subclass definitions for the host module.
`killer.cpp`	The main function for the host module, which starts the Resolver by creating an OSSL process object. When the user presses Return, the main function stops the Resolver process by deleting the process object. Object constructors and destructors and utility functions, which include an upcall handler callback.

Source File	Contains
killeractions.cpp	The initialization and action functions for the accelerator module; ACE subclass declaration, constructor, and destructor. The ACE contains an upcall and a method that counts packets, constructs a message from the count, and sends it in an upcall.
killerrules.ncl	The classification rules for the accelerator module; in this case, several rules that classify the packet according to its type and source or destination address, as well as the final rule that passes all packets to the counter action function.
nbossl.h	Error handling and subclass definitions for the operating system service layer (OSSL).
nbossl.cpp	Function and method definitions for the OSSL, which provide OS-independent process handling.

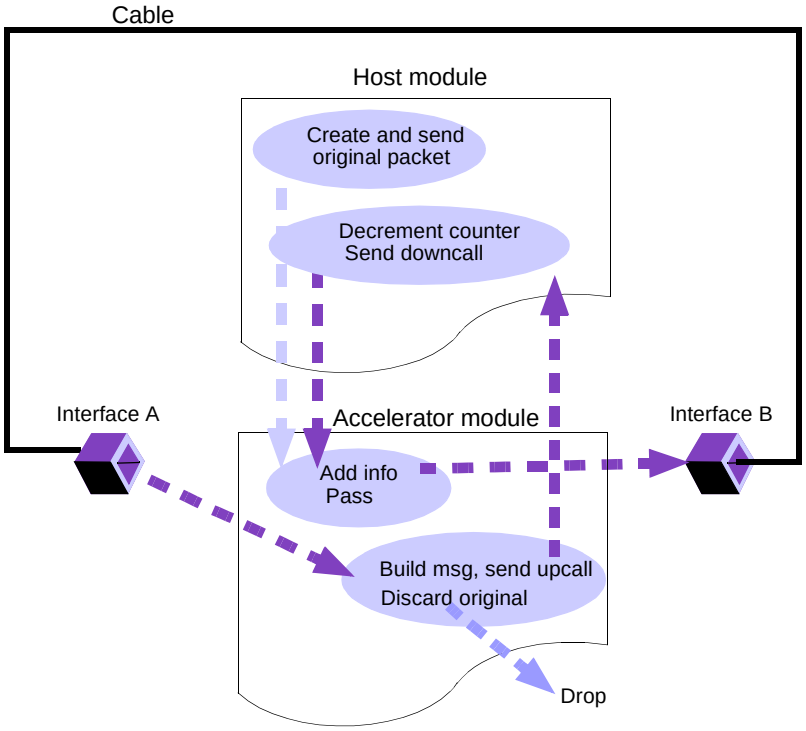
LoopApp

This sample runs a simple test to show that the IX-API SDK operating environment is configured correctly. To use this test, you must connect a 10/100 BT crossover cable between the two interfaces of the Policy Accelerator.

When the `NBappl` object is constructed, it creates a packet that contains a counter and sends it to the ACE in a downcall. Each time the ACE receives the packet in a downcall, it adds some information and forwards the packet to the default pass target, the B interface. Because the interfaces are connected by a cable, this causes the packet to be received again by the accelerator module, on the A interface.

When the ACE receives a packet from interface A, it creates a message from it, which it sends back to the host module in an upcall. It then forwards the original packet to the default drop target.

Each time the host module receives the packet back in an upcall, it decrements the counter in the packet and sends it back to the ACE in a downcall, where it starts on the loop again. When the counter reaches 0 (after 255 repetitions), it displays a message that the test is successful.



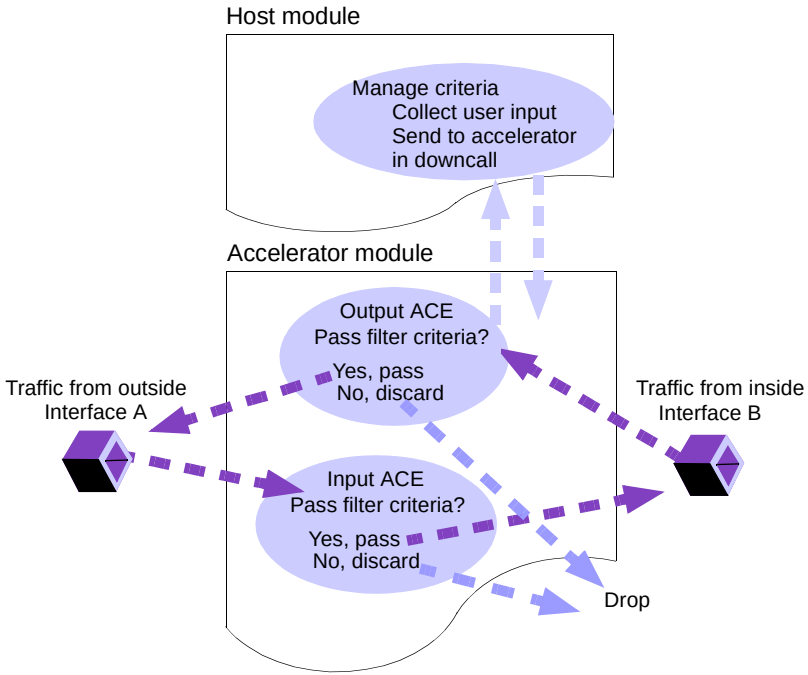
The application contains the following source files in the directory `demos/LoopApp/source`:

Source File	Contains
`loopapp.cpp`	The main function for the host module, subclass definitions, object constructors and destructors, and the upcall handler call-back that decrements a counter in the packet and sends it in a downcall.
`loopact.cpp`	The initialization and action function for the accelerator module; ACE subclass declaration, constructor and destructor; and the downcall handler callback that adds information to the packet it receives and forwards it to the pass target. The action function constructs a message from a packet received on the interface and sends it in an upcall.
`loopncl.ncl`	The classification rule for the accelerator module. A single rule passes all packets to the single action function.

Firewall

This more complex and complete sample is an example of how to construct a firewall application.

The application contains two ACEs, Input and Output. Each ACE intercepts traffic in one direction between two interfaces, one connected to the external network and the other to the internal network. The ACEs accept or reject packets based on whether their port or address information matches values in the `ipdeny` and `tcpallow` data sets.



The host module allows the user to manage the criteria used by the ACEs to accept or reject packets. The main function runs a loop (`DoMenu`) that displays a menu of options for the user, looks for the user's keyboard input, and sends the result to the accelerator module in a downcall.

A downcall handler callback in each ACE interprets the command received in the downcall and responds accordingly. Depending on the command, it can:

- Adjust the contents of the data sets of allowed and denied addresses and ports.

- Send an upcall to the host module with requested information. In the host module, an upcall handler unpacks and displays the message.

The example menu allows the user to issue the following commands:

Command	Action
f	Read rules from a file (an example file, `rules.txt`, is provided)
i	Show input firewall rules
a	Add IP address to denied incoming hosts
d	Delete IP address from denied incoming hosts
p	Add port to list of accepted incoming ports
r	Remove port from list of accepted incoming ports
o	Add host IP address to list of denied outside hosts
n	Remove host IP address from list of denied outside hosts
q	Quit

The application contains the following source files in the directory `demos/Firewall/source`:

Source File	Contains
`firewallAppl.h`	The data structure defining the commands the user can issue.
`firewallAppl.cpp`	The main function for the host module, subclass definitions, object constructors and destructors, and method definitions. Each ACE manager contains an upcall handler with its call-back method, a downcall, and a method to send a downcall. The constructor for the application object runs a loop that displays a menu, waits for keyboard input, then interprets and acts on the command received. The main function creates the application object, whose constructor runs the menu loop.

Source File	Contains
`InputACT.cpp`	The initialization and action functions for the accelerator module of the Input ACE; subclass declarations, constructors and destructors; and method definitions. The ACE object contains a downcall handler with its callback and a method to send an upcall. The downcall handler responds to the commands it receives by modifying the sets of addresses to be accepted or rejected. The two action functions simply pass or drop packets.
`InputNCL.ncl`	The classification rules for the accelerator module of the Input ACE. The rules determine whether to accept (pass) or reject (drop) packets, based on whether their port and address values match members of the `ipdeny` or `tcpallow` sets. They also reject packets that exhibit various error conditions; however, this example does not handle all cases.
`OutputACT.cpp`	The initialization and action functions for the accelerator module of the Output ACE; subclass declarations, constructors, and destructors; and method definitions. The ACE object contains a downcall handler with its callback and a method to send an upcall. The downcall handler responds to the commands it receives by modifying the sets of addresses to be accepted or rejected. The two action functions simply pass or drop packets.
`OutputNCL.ncl`	The classification rules for the accelerator module of the Output ACE. The rules determine whether to accept (pass) or reject (drop) packets, based on whether the destination address values match members of the `ipdeny` set. They also reject packets that exhibit various error conditions; however, this example does not handle all cases.
`rules.txt`	An example of a file containing a set of firewall rules that the application can read in directly.

Crosscall

This sample demonstrates the use of crosscalls for communication between ACEs, using the same basic format and bindings as the Firewall application.

The host module allows the user to initiate a crosscall from either ACE with a keystroke. The main function runs a loop (`DoMenu`) that displays a menu of options for the user, looks for the user's keyboard input, and sends the result to the accelerator module in a downcall.

When the accelerator module receives the downcall, the ACE's downcall handler sends a response in an upcall. In the host module, the ACE manager's upcall handler unpacks and displays the message. If the ACE receives the proper code, it also sends a crosscall to the other ACE. The other ACE receives the crosscall and its crosscall handler unpacks and displays the message.

The application contains the following source files in the directory `demos/Crosscall/source`:

Source File	Contains
`crosscallAppl.h`	The data structure used for sending messages between the host and accelerator modules.
`crosscallAppl.cpp`	The main function for the host module, subclass definitions, object constructors and destructors, and method definitions. Each ACE manager contains a downcall, crosscall manager, and crosscall handler manager object, as well as an upcall handler with its callback method and a method to send a downcall. In addition to creating the bindings, the application object constructor creates the crosscall links, connecting each crosscall to the crosscall handler in the other ACE. The main function runs a menu loop.
`InputACT.cpp`	The initialization and action functions for the accelerator module of the Input ACE; subclass declarations, constructors, and destructors; and method definitions. The ACE contains a downcall handler with its callback and a method to send an upcall. The ACE also contains a crosscall, a method to send the crosscall, and a crosscall handler with its callback. The two action functions simply pass or drop packets.

Source File	Contains
`InputNCL.ncl`	The classification rules for the accelerator module of the Input ACE. The single rule passes all packets.
`OutputACT.cpp`	The initialization and action functions for the accelerator module of the Output ACE; subclass declarations, constructors, and destructors; and method definitions. The ACE contains a downcall handler with its callback and a method to send an upcall. The ACE also contains a crosscall, a method to send the crosscall, and a crosscall handler with its callback. The two action functions simply pass or drop packets.
`OutputNCL.ncl`	The classification rules for the accelerator module of the Output ACE. The single rule passes all packets.

StringSearch

This sample demonstrates how to search for strings within packets, using a format similar to the Firewall application.

The host module allows the user to add or remove strings for which to search. The main function runs a loop (`DoMenu`) that displays a menu of options for the user, looks for the user's keyboard input, and sends the result to the accelerator module in a downcall.

When the accelerator module receives the downcall, the ACE's downcall handler interprets the command received and responds accordingly. Depending on the command, it takes one of the following actions:

- Sends an acknowledgement to the host module in an upcall. In the host module, an upcall handler unpacks and displays the message.
- Initializes the search context and engine objects, using the string passed in the downcall.

A single rule passes all packets to the `action_pass` action function. Once a search has been set up for a particular string, the action function passes each incoming buffer to the string search before disposing of it. In this example, the results of the search do not affect the disposition of the packet; all packets are sent to the pass target.

The accelerator module defines two string search callbacks:

- The per-match callback, `GotString`, is executed each time a search finds a matching string. In this example, the callback displays information about the matching string, and instructs the search to terminate for the current buffer.
- The per-buffer callback, `AfterBuffer`, is executed when the search is complete for the buffer, whether or not it found a matching string. In this example, the callback simply displays a message. In a real application, it might make a decision about what to do with the buffer based on the result of the search and transmit that decision back to the calling action function.

The application contains the following source files in the directory `demos/StringSearch/source`:

Source File	Contains
`strsrchAppl.cpp`	The main function for the host module, subclass definitions, object constructors and destructors. The ACE manager contains an upcall handler with its call-back and a method that sends a downcall. The main function runs a menu loop.
`strsrchACT.cpp`	The initialization and action functions for the accelerator module; subclass declarations, constructors and destructors; and method definitions. The ACE contains a downcall handler with its callback, and a method to send an upcall. The ACE also contains string search engine and context objects, and a method to initialize them. The context object contains the search response callbacks. The `action_pass` action function sends each buffer to the string search before sending it to the pass target.
`strsrchNCL.ncl`	The classification rules for the accelerator module. In this case, the single rule passes all packets to the `action_pass` function.

.

Appendix B
Packet-Counting Application

This appendix includes the complete source for the packet-counting application created in Chapter 2, “Tutorial: Creating a Simple Application.” You can also view the code online.

Host Module Header (CountApp.h)

```
// include system interface header files

#include “NBswap.h”
#include “NBapi/nbappl.h”

// user-defined error codes

#define NBERROR_COUNT_BASE (NBERROR_USER_BASE + 0x1000)
#define NBERROR_COUNT_ERRCODE (x) (NBERROR_COUNT_BASE + (x))

#define NBERROR_COUNT_CANNOTCREATEGROUP NBERROR_COUNT_ERRCODE(1)
#define NBERROR_COUNT_CANNOTCREATEACE NBERROR_COUNT_ERRCODE(2)
#define NBERROR_COUNT_CANNOTCREATEUPCALL
NBERROR_COUNT_ERRCODE(3)
#define NBERROR_COUNT_CANNOTLOADCODE NBERROR_COUNT_ERRCODE(4)

// AceManager subclass declaration
class CountAceManager: public AceManager
{
 public:
   CountAceManager (NBAppl* appl, AceGroup* acegroup, char*
name);
   ~CountAceManager();
   // showPacketCount method
   void showPacketCount (Message* m);
   // pointer to UpcallHandler object
   UpcallHandler* showPacketCountUpcallHandler;
}
```

```
// AceGroup subclass declaration
class CountAceGroup: public AceGroup {
   public:
   CountAceGroup (NBAppl* appl, NBFactory* nbf, char* name,
NBStringList* list);
   ~CountAceGroup();
   //       pointer to AceManager subclass object
   protected:
    CountAceManager* countAceManager;
}


// NBAppl subclass declaration

class CountAppl: public NBAppl {
public:
   CountAppl (char* name, char* curdir, char* cmdLine);
   ~CountAppl();
   // pointer to AceGroup subclass object
   protected:
    CountAceGroup* countAceGroup;
};
```

Host Module (CountApp.cpp)

```
#ifdef Win32
   #include <iostream.h>
   #include <direct.h>
#else
   #include <unistd.h>
   #include <stdlib.h>
#endif

#include CountApp.h

// NBAppl subclass definition

CountAppl::CountAppl (char* name, char* cwd, char* cmd):
NBAppl (name, cwd, cmd)
   {
   // AceGroup subclass instantiation
   try {
      countAceGroup = new CountAceGroup (this, NULL,
“CountAceGroup”, NULL);
```

```
   }
   catch (NBError&) {
      throw NBError (NB_ERROR
(NBERROR_COUNT_CANNOTCREATEGROUP));
   }
   // Network interface bindings
   uint32 rval;
   rval = bind
(“/nbhwpe0/FromInterface:nbhwpe0A/Interface/pass”,
         “/CountPackets/CountAceGroup/CountAce”);
   if (rval != NB_SUCCESS) {
      NB_ABORT(rval);
   }
   rval = bind (“/CountPackets/CountAceGroup/CountAce/pass”,
         “/nbhwpe0/ToInterface:nbhwpe0B/Interface”);
   if (rval != NB_SUCCESS) {
      NB_ABORT(rval);
   }
   }
CountAppl::~CountAppl()
   {
   // delete AceGroup subclass
   delete countAceGroup;
   }


// AceGroup subclass definition

CountAceGroup::CountAceGroup (NBAppl* appl, NBFactory* nbf,
char* name, NBStringList* list) :
AceGroup (appl, nbf, name, list)
   // AceManager subclass instantiation
   {
      try {
         countAceManager = new CountAceManager (appl, this,
“CountAce”);
      }
      catch (NBError&) {
         throw
NBError(NB_ERROR(NBERROR_COUNT_CANNOTCREATEACE));
      }
   }

CountAceGroup::~CountAceGroup()
{
   // delete AceManager subclass
   delete countAceManager;
}
```

```
// AceManager subclass definition

CountAceManager::CountAceManager (NBAppl* appl, AceGroup*
acegroup, char* name):
// constructor
AceManager (appl, acegroup, name)
{
   // UpcallHandler class instantiation
   try {
   #ifdef WIN32
    showPacketCountUpcallHandler =
      new UpcallHandler (appl, acegroup, this,
“showPacketCount”,
                       (UpcallFp) showPacketCount);
   #else
    showPacketCountUpcallHandler =
      new UpcallHandler (appl, acegroup, this,
“showPacketCount”,
                          (UpcallFp)&showPacketCount);
   #endif
   }
   catch (NBError&) {
      throw NBError (NB_ERROR
(NBERROR_COUNT_CANNOTCREATEUPCALL));
   }

   // load accelerator module into Policy Accelerator
   int errorcode;
   if ((errorcode = load (“CountRules”,    // NCL (.ncl) file
            “CountActions”)) // action (.nbo) code
      != NB_SUCCESS)
   {
      throw NBError (NB_ERROR(NBERROR_COUNT_CANNOTLOADCODE));
   }
}
// showPacketCount method definition
void CountAceManager::showPacketCount (Message* m)
{
   NB_ASSERT (m->getLen1() == sizeof(nuint32));
   printf (“Packet Counter: 0x%08x\n”, ntohl (*(nuint32 *)m-
>getBuffer1()));
   releaseMessage (m); // Message passed here, dispose of it.
}
// destructor
CountAceManager::~CountAceManager()
{
   // delete UpcallHandler
   delete showPacketCountUpcallHandler;
```

```
}

// application main function

void main (int argc, char** argv) {
   nb_trace_verbose(1);

   // NBAppl subclass instantiation
   CountAppl* appl;
      try {
           appl =  new CountAppl (“CountPackets”, NULL, NULL);
      }
      catch (NBError&) {
           fprintf(stderr, “CountAppl creation failed!\n”);
           exit(2);
      }

   // main loop
   while(1)
      #ifdef Win32
           _sleep(999999);
      #else
           sched_yield();
      #endif
   }
```

PE Module (CountActions.cpp)

```
// include system interface header files

#include “NBaction/NBAction.h”

// Ace subclass declaration

class CountAce: public Ace {
public:
   CountAce (ModuleId id, char* name, image* obj);
   // showPacketCount method
   void showPacketCount (void);
   // packet counter declaration
   int packetCounter;
   // packet counter snapshot for message
   nuint32 countSnapshot;
   // upcall declaration
   protected:
    Upcall showPacketCountUpcall;
```

```
}



// Ace subclass definition

CountAce::CountAce (ModuleId id, char* name, Image* obj) :
Ace (id, name, obj), // note ending comma (declaration
incomplete)

   // Upcall subclass instantiation
   showPacketCountUpcall (id, this, “showPacketCount”)
   // initialize packet counter
{
   packetCounter = 0;
};
// showPacketCount method definition

void CountAce::showPacketCount (void)
{
   // increment counter
   packetCounter++;
   if (!((packetCounter-1)%0x20)) // Don’t do upcall for every
packet;
                           // could overwhelm the driver

   {
   // take snapshot of count with known byte order
   countSnapshot = htonl(packetCounter);
   // create message
   MessageBlock b ((char *) &countSnapshot, sizeof
(countSnapshot));
   Message msg (mb);
   // invoke upcall
   if (showPacketCountUpcall.call(&msg) != 0 )
      printf(“Upcall failed.\n”);
   }
}


// action function entry point (sends packet count)

ACTNF action_all (Buffer* buf, CountAce* ace)
{
   ace->showPacketCount ();
   return RULE_CONT;
}

// main-equivalent (init_actions)
```

```
INITF init_actions (void* id, char* name, Image* obj)
{
   // instantiate and return Ace subclass
   return new CountAce (id, name, obj);
}
```

Rules (CountRules.ncl)

```
// rule to invoke action for all packets

rule all_packets { 1 } { action_all() }
```

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

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

Revision 2.3, May 2000