PacketPortal

Release 1.03.501

SDK Guide

Customer Support

For JDSU Customer Support, access the following URL:

http://www.jdsu.com/support

PacketPortal Documentation Suite Reference

The following describes the suite of documentation available to PacketPortal customers. Contact your JDSU PacketPortal representative for more information on downloading this documentation.

Document Title	Document #
Hypertext Library	NSDTP-40010
Architecture and Deployment Guide	NSDTP-40020
Installation Guide	NSDTP-40030
System Manager Online Help	NSDTP-40040
SDK Product Guide	NSDTP-40050
PDG User's Guide	NSDTP-40060
SFProbe Configuration Guide	NSDTP-40070
VNIC User's Guide	NSDTP-40080
PacketPortal Computer-based Training	NSDTP-40100
Release Notes	NSDTP-40110

Guide Conventions

This guide uses the following conventions:

Notes, Cautions, and Warnings

Note: Notes include important supplemental information or tips related to the main text.

Cautions apply to software actions. They indicate a situation that could lead to a loss of data or a disruption of software operation if indicated precautions are not taken.

Warnings apply to hardware handling. They indicate a potentially hazardous situation that could result in damage to the unit, serious bodily injury, and/or death, as from electrocution, if indicated precautions are not taken.

Typographical Conventions

Description	Example
Buttons that you click appear in **this typeface**.	Click the **Filter** button.
Code and output messages appear in `this typeface.`	`All results okay.`
Text that you must type exactly as shown appears in **`this typeface.`**	Type: **`a:\set.exe`** in the dialog box.
References to guides, books, and other publications appear in *this typeface.*	Refer to *Newton's Telecom Dictionary.*
A vertical bar | means "or"; only one option can appear in a single command.	**`platform a|b|e`**
Square brackets [ ] indicate optional arguments.	**`login [platform name]`**
Slanted brackets < > indicate variables.	**`<password>`**

A plus sign (+) indicates simultaneous keystrokes.	Press **Ctrl+s**
A comma (,) indicates consecutive key strokes.	Press **Alt+f,s**
A single slanted bracket (>) indicates choosing a submenu from a menu.	On the menu bar, click **Start > Program Files.**

Table of Contents

Chapter 1. Introduction

Software Development Kit Description

SFProbes are small form-factor pluggable (SFP) transceivers that when placed in a network, can be configured to look for specific network packet types without disrupting the original traffic flow. When there is a match, the SFProbe generates a Filter Results Packet (FRP) that contains metadata as well as the original packet (or specified parts of the original packet.) The FRP is forwarded to the PacketPortal system's Packet Routing Engine (PRE). The PRE then forwards the FRP to applications.

Filter Results Packets (FRPs) contain metadata as well as the original packet (or portions of the original packet as specified in the Feed definition.) FRPs are then forwarded to applications via the Packet Routing Engine (PRE). The PRE then forwards the FRPs to applications.

The SFProbes can also be configured to generate metrics results packets (MRPs) which contain key information about various nodes in the network such as packet counts and packet byte counts. The MRPs are forwarded to the PRE, which then forwards them to Ethernet-based applications.

The PacketPortal Software Development Kit (SDK) consists of a set of software libraries, examples, and tools that enable software developers to write applications that analyze the FRPs and MRPs received from the PRE. The software libraries included in the SDK comprise an Application Programming Interface (API) that enables software developers to integrate software with the PacketPortal system. The API allows applications to extract captured network traffic and metadata such as the capture time and the SFProbe identifier.

The Software Development Kit contains all of the elements required to create scripts or programs to process FRPs. The SDK supports software development in both a live and a simulated PacketPortal environment.

The SDK contains the following:

- An API for accessing PacketPortal captured traffic data and metrics feeds
- Source code examples that illustrate API implementation in C++, Java, Perl, and Python
- Documentation about the API
- Tools to simulate FRPs and MRPs for use in debugging newly developed applications

PacketPortal System Overview

PacketPortal is a cloud-based packet acquisition and filtering system. It selectively copies traffic from 1 Gigabit per second (Gbps) SFP optical links in a network and forwards the filtered traffic to Ethernet-based applications for analysis. Through its unique, highly distributed hardware architecture, it provides enhanced network visibility over traditional probe systems. This functionality significantly reduces the cost and time to troubleshoot network issues and monitor network health and performance.

PacketPortal Components

SFProbes: the Core of the PacketPortal System

PacketPortal uses enhanced optical SFP transceivers called SFProbes to filter network traffic and route it to applications and systems for further analysis. By strategically deploying intelligent SFProbes throughout edge, access and core networks, you can obtain real-time access to network, service and application data.

SFProbes selectively copy and forward filtered data to network analysis tools in-line with network traffic. The filtered data is inserted only during idle periods, so there is minimal disruption to customer traffic flow.

Because SFProbes plug into any standard SFP port, replacing standard 1 Gbps optical SFPs, you can collect network traffic from virtually any location, including at the network edge, without the need to be physically present.

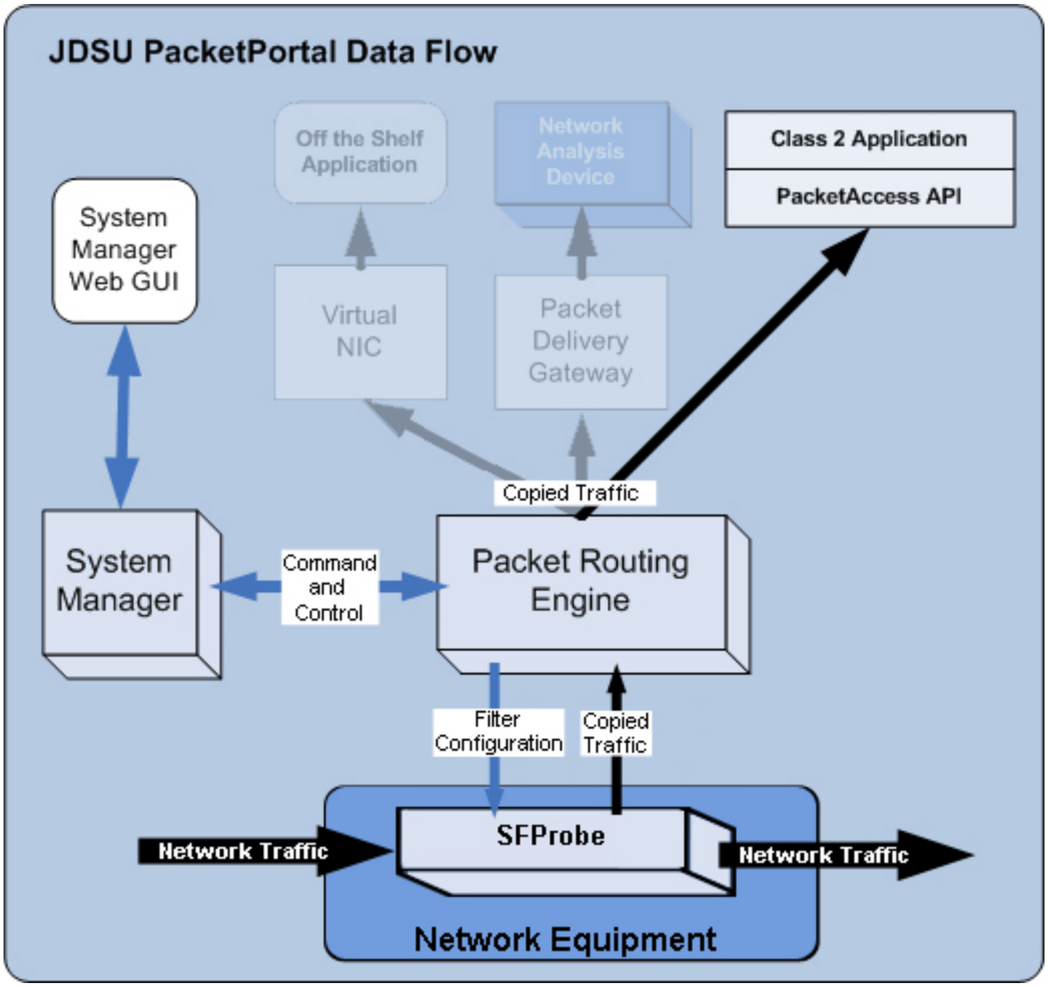
SDK Integration with PacketPortal

The PacketPortal SDK is designed to work independently and in conjunction with the PacketPortal System. The SDK provides tools, such as FilterResultsReplay and MetricsResultsReplay, that allow you to emulate PacketPortal data feeds without requiring a functional PacketPortal System. In addition, PacketPortal contains two utilities that allow an unaltered application to receive network traffic that has been captured and filtered by PacketPortal. The two utilities are the Virtual NIC (VNIC) Manager and the Packet Delivery Gateway (PDG). They are installed and documented separately.

The PacketAccess API gives developers the ability integrate their application into the PacketPortal system and process PacketPortal metrics and data feeds directly.

- Applications that are able to directly access PacketPortal metrics and data-feeds using the PacketAccess API are considered to be *Class 2 integrated applications*.

- Unaltered applications that use the VNIC or PDG (and thus, do not use the PacketAccess API) for access to data feeds are considered to be *Class 1 compatible applications*.



PacketPortal is fundamentally an enhanced, remote network traffic capture tool, not a network traffic analysis tool. The PacketAccess API gives developers access to network packets and some network context information, such as packet direction on the network and time captured. The data stream consists of network traffic that can be collected from multiple network locations simultaneously and then can be delivered to either a single location or to multiple locations. The traffic in the data-stream is pre-selected by creating filters in the PacketPortal System Manager.

By integrating applications with the PacketPortal system using the PacketAccess API, developers can take full advantage of all the data captured by the PacketPortal system and focus development efforts on data presentation and analysis.

Chapter 2. Installing the SDK

Recommended Hardware

The recommended hardware for an application using the PacketAccess API should have the following as a minimum:

- CPU: at least 2 cores
- RAM: at least 1 GB

Each application should determine its own hardware requirements.

Supported Operating Systems

The SDK supports the following operating systems:

Linux:

64-bit SUSE Linux Enterprise Server (SLES) 11 SP2

64-bit Red Hat Enterprise Linux Server 6.3 (RHEL6)

Microsoft Windows:

32-bit Windows 7 SP1

32-bit Windows XP with SP3 or above

64-bit Windows 7 SP1 as native 64-bit or WoW64 application

SDK GUI Installation

The SDK GUI installation is available on both operating systems (Linux and Windows).

Note: *In the following installation process, not all installation screens are common between the two operating systems. Where there are differences, notations are made to differentiate the two installation types.*

To install the PacketPortal SDK Interface software, do the following:

1) Download the latest PacketPortal SDK installation software to your target machine.

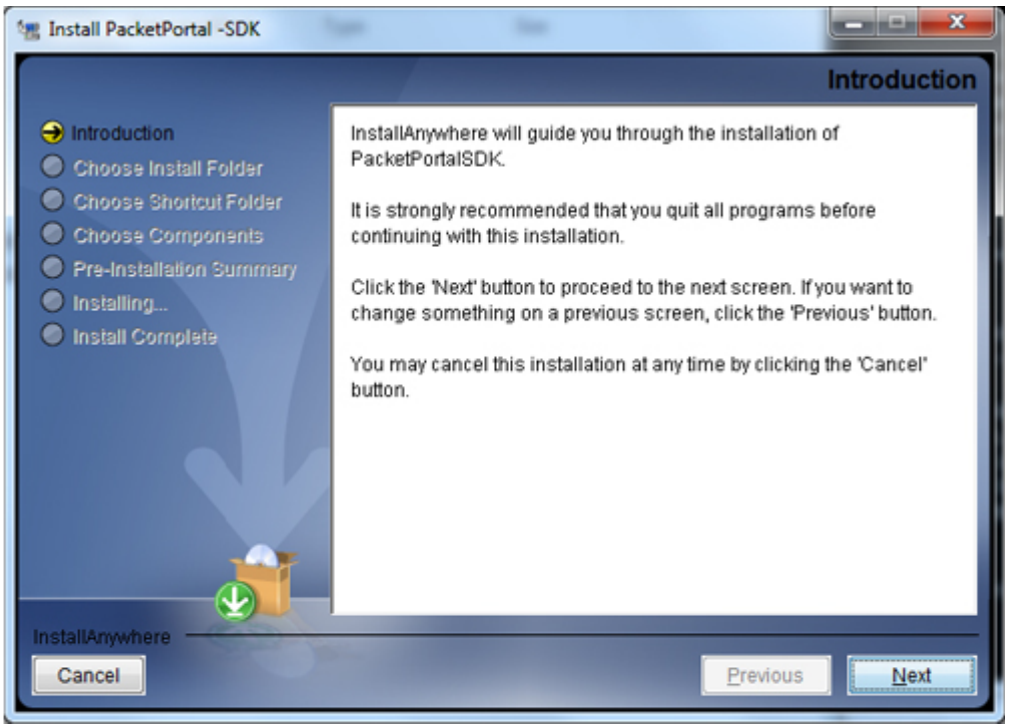
2) Run the installer by either double-clicking on the installer executable or running it from the command line:

   `PacketPortal-Installer-<VERSION>.bin` (Linux)
   - or -
   `PacketPortal-Installer-<VERSION>.exe` (Windows)

   The following is the first screen you will see when the installation begins.

Note: *You may cancel the installation at any time by clicking the **Cancel** button.*

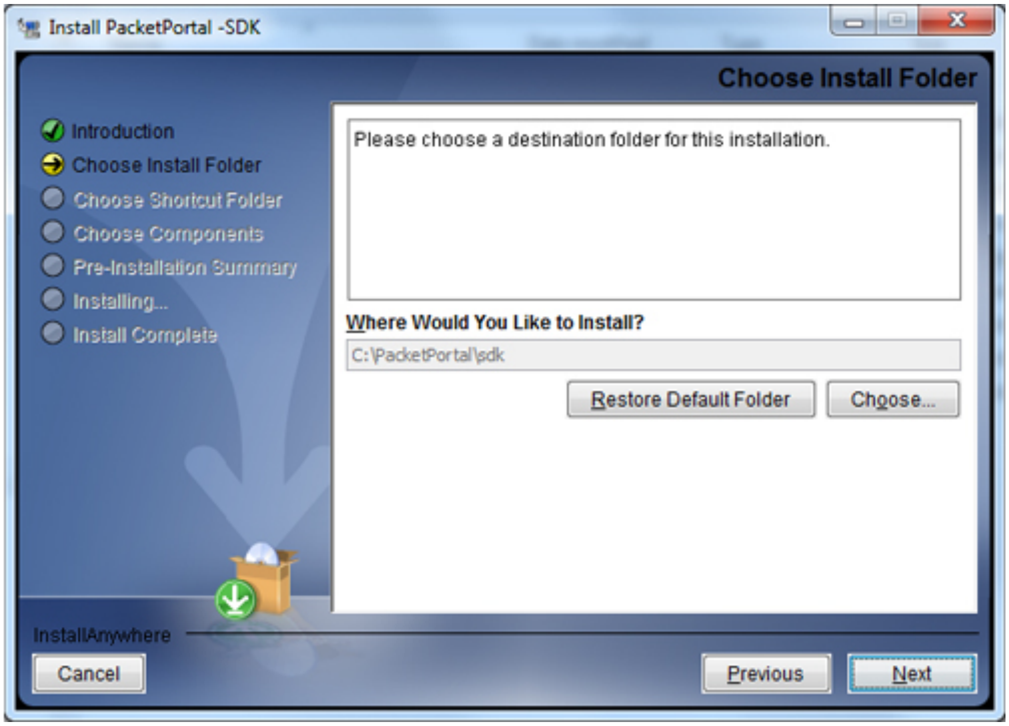
3) Click **Next** to continue the installation.



4) Review the location where the SDK application will be installed. This location is shown in the **Where Would You Like to Install** field. If you want to change the location, select the **Choose…** button and browse to find the desired location (Windows only).

   If you want to return the installation to the default location, select the **Restore Default Folder** button.

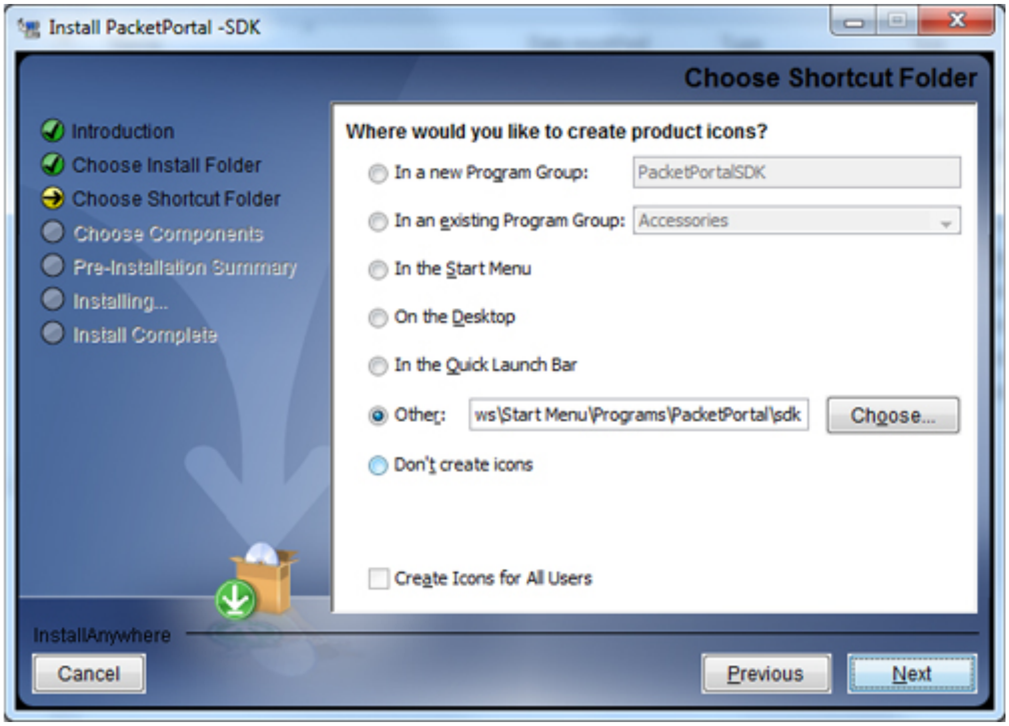
5) Click **Next** to continue the installation (Windows only).



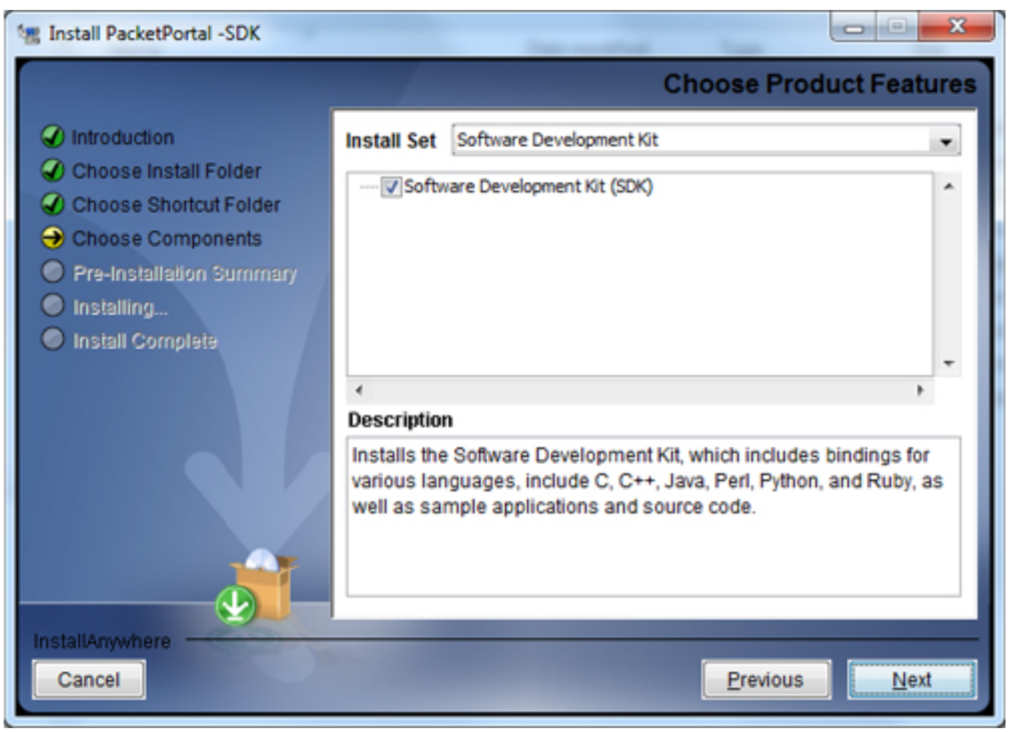
6) Review where you would like to the SDK product icons to be installed. Select the desired icon location (Windows only).

7) Select the **Create Icons for All Users** checkbox if you would like the icons to be created for all users (Windows only).

8) Click **Next** to continue the installation (Windows only).

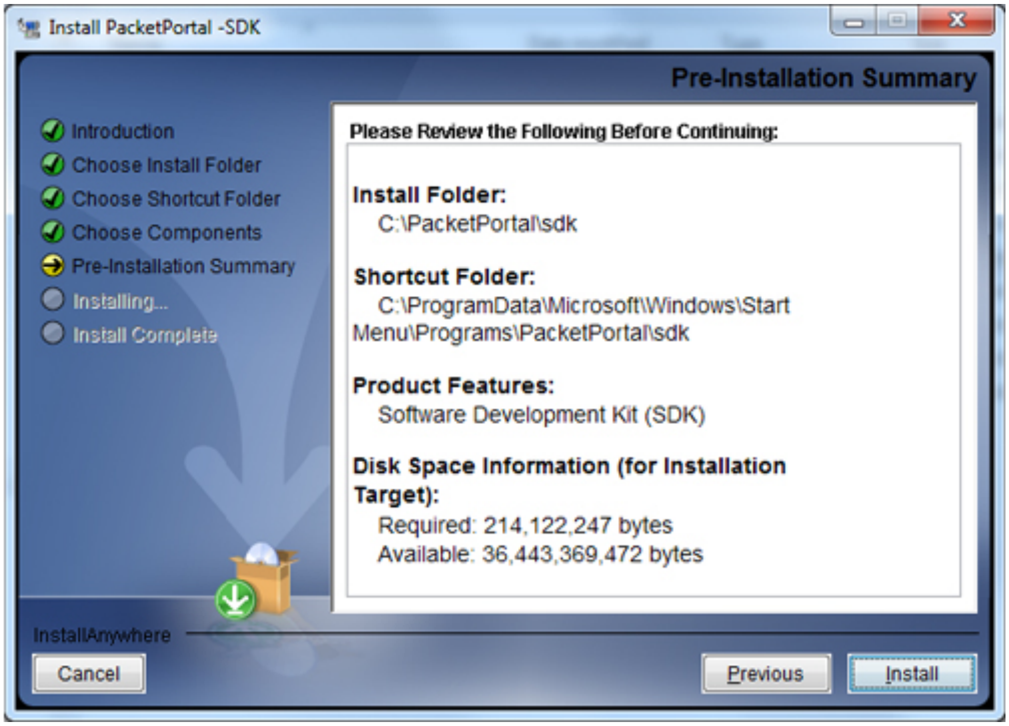


9) Verify the **Software Development Kit (SDK)** checkbox is selected and then click **Next** to continue the installation.

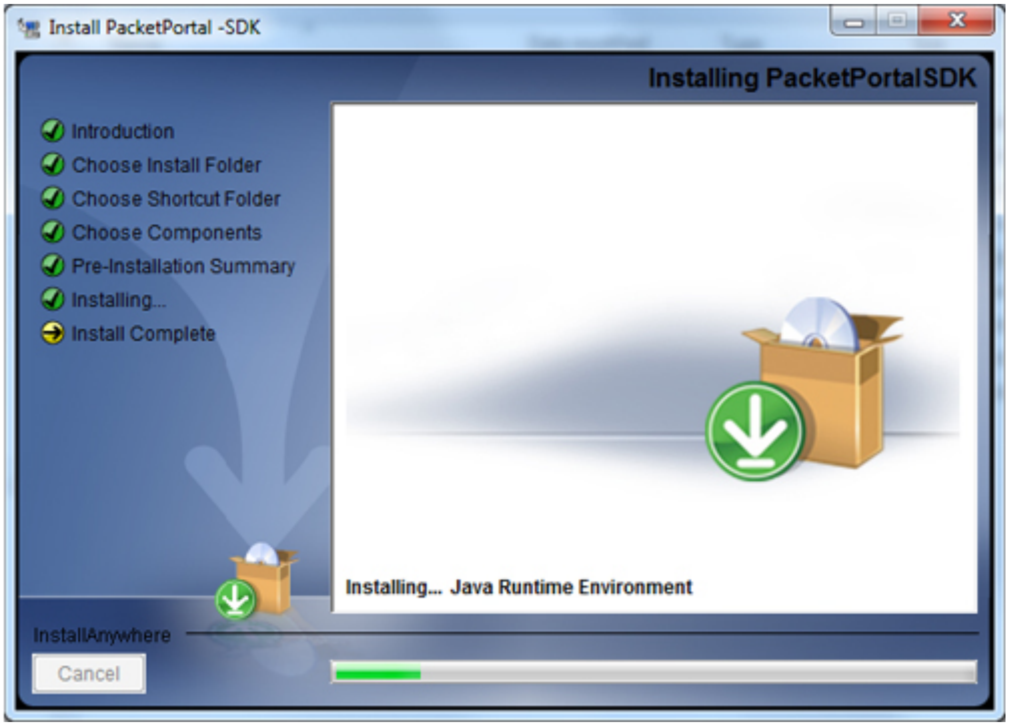


10) Verify the installation configuration before starting the actual installation of the software. If you would like to make changes, click **Previous** to make the desired changes.

11) Click **Next** to start the software installation.

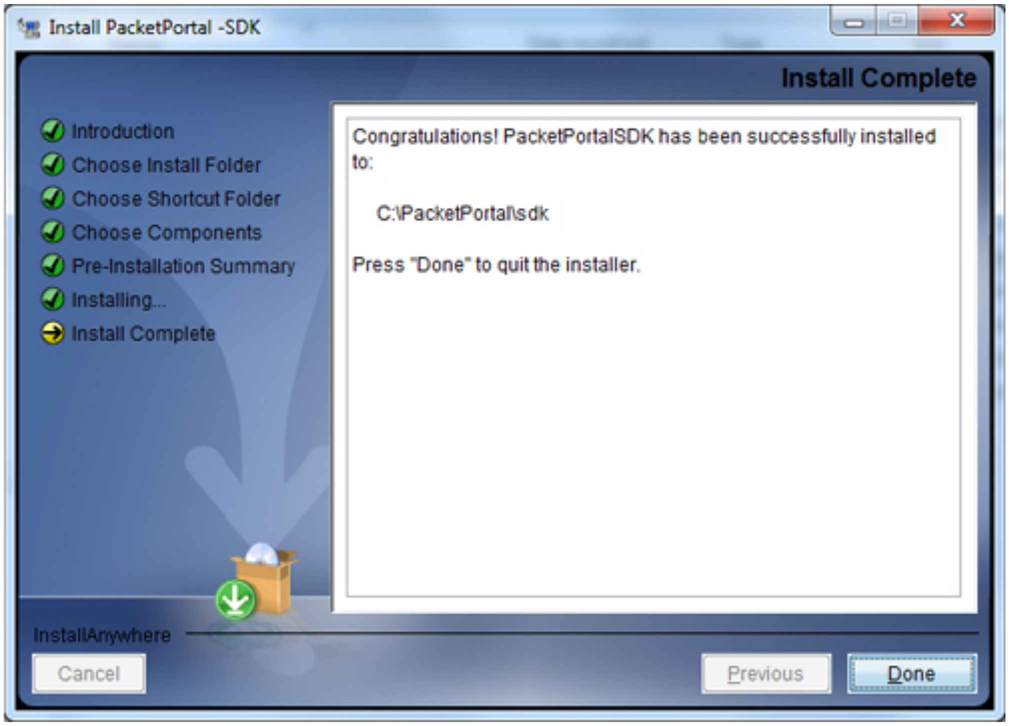


12) Make sure the installation is progressing. The status bar at the bottom of the screen shows the progress of the installation.



13) When the SDK has been installed, review the installation information. The Linux installer also includes a **Details…** button to show what has been installed and a **Refresh** button to show updates.

14) Click **Done** to exit the installer.



The SDK is now ready to use.

SDK Command Line Installation

Use the following actions to install the SDK using the command line, if graphical installation is unavailable or not desired.

> ***Note:*** *If the Windows version is started from the command line, the installation proceeds as shown in the SDK GUI installation procedure. The following shows command line installation on a Linux system.*

Actions	**Descriptions**
Enter: ./PacketPortal-Installer-SDK -<VERSION>.bin -i console then press: <ENTER>	SDK Installation package

Press: <ENTER>	`Preparing to install...`
	
	`Extracting the JRE from the installer archive...`
	
	`Unpacking the JRE...`

<table>
<tr><td></td><td><pre>Extracting the installation resources from the installer archive...

Configuring the installer for this system's environment...

Launching installer...

Graphical installers are not supported by the VM. The console mode will be used
instead...

===

PacketPortalSDK   (created with InstallAnywhere)

Preparing CONSOLE Mode Installation...

Custom code execution Started...

Custom code execution Completed...

===

Introduction

This installer will guide you through the installation of the PacketPortal

software components.

Follow the instructions at the prompts to proceed

Type "back" at any prompt to return to a previous screen

Type "quit" at any prompt to quit the installer

PRESS <ENTER> TO CONTINUE:</pre></td></tr>
<tr><td>Enter option:1

then press:
<ENTER></td><td><pre>===

Choose Install Set

===

Choose Product Features

ENTER A COMMA_SEPARATED LIST OF NUMBERS REPRESENTING THE FEATURES YOU WOULD
LIKE TO SELECT, OR DESELECT. TO VIEW A FEATURE'S DESCRIPTION, ENTER
'?<NUMBER>'. PRESS <RETURN> WHEN YOU ARE DONE:</pre></td></tr>
</table>

	`    1- [X] Software Development Kit (SDK)` `Please choose the Features to be installed by this installer.:`

Press:
<ENTER>
	`===============================================================================`
	`Pre-Installation Summary`
	`------------------------`
	`Please Review the Following Before Continuing:`
	`Install Folder:`
	`    /opt/PacketPortal`
	`Product Features:`
	`    Software Development Kit (SDK)`
	`PRESS <ENTER> TO CONTINUE:`

Enter option: 5
then press:
<ENTER>
	`===============================================================================`
	`Installing...`
	`-------------`
	`[==================|==================|==================|==================]`
	`[------------------`
	`Custom code execution Started...`
	`Custom code execution Completed...`
	`|------------------|------------------|---------`
	`Custom code execution Started...`
	`Custom code execution Completed...`
	`---------`
	`Custom code execution Started...`
	`Custom code execution Completed...`
	`-]`
	`===============================================================================`

	Installation Completed
	
	----------------------- 	
	
	Please review the tasks that were performed by the installer. To see details for a specific task, enter the corresponding task number, or select the last option to exit the installer.
	
	    1- Check Linux Distribution
	
	    2- Uninstall RPMs
	
	    3- Install JRE
	
	    4- Install SDK
	
	    5- Exit the installer

	The installation is complete.

Chapter 3. Using the SDK

API Programming Languages

The PacketPortal SDK supports a variety of compiled and scripted programming languages. The native language API interfaces are provided in the SDK to enable software development in that language. In addition to the supported programming languages called out in the next table, the Runtime Requirements and the Required Build Environments for each programming language and operating system is shown further below.

Supported Programming Languages

The following programming languages are supported on these operating systems:

Programming Language	Supported on these Operating Systems[1]	
	Windows	Linux
C++ (Static Library version)	Yes	Yes
C++ (Dynamic Library version)	Yes	No
Java	Yes	Yes
Perl 5.10	Yes (32 bit)	Yes
Perl 5.14	Yes (64 bit)	No
Python 2.6	No	Yes
Python 2.7	Yes	No

[1] Refer to SDK Supported Operating Systems for the supported operating system versions.

Runtime Requirements

The following runtime applications are required based on the supported programming language and operating systems:

Programming Language	Operating Systems	
	Windows	Linux
C++ (Static Library version)	WinPcap 4.1.2 or above	Libpcap 1.0 or above
C++ (Dynamic Library version)	WinPcap 4.1.2 or above	N/A
Java	WinPcap 4.1.2 or above JRE 6 or above	Libpcap 1.0 or above JRE 6 or above
Perl 5	WinPcap 4.1.2 or above Perl interpreter (e.g. Strawberry Perl)	Libpcap 1.0 or above
Python	WinPcap 4.1.2 or above	Libpcap 1.0 or above

	Python 2.7	Python 2.6

Required Build Environments

The following build environment applications are required based on the supported programming language and operating systems:

Programming Language	Operating Systems	
	Windows	Linux
C++ (Static Library version)	Visual Studio 2010	GCC 4, make
C++ (Dynamic Library version)	Visual Studio 2005 or above	N/A
Java	JDK 6 or above	JDK 6 or above
Perl 5	Perl interpreter (e.g. Strawberry Perl)	Perl is pre-installed in most Linux systems. Run "perl -v" to find out which version is on your system.
Python	Python 2.7	Python 2.6

Basic PacketAccess API Library Usage to Obtain Filter Results

Some concepts to be familiar with are:

- Objects are created, have a lifetime, then must be freed.
- Sources of input to an application using the API are TCP sockets, UDP sockets, and libpcap files captured from network devices.
- The mechanism for retrieving packets by the application is a polling mechanism.

In a typical environment, an application can retrieve filter results by following these steps:

1) Create a FilterResultAccess object by calling the CreateFilterResultAccess function.

2) Specify the source of the Filter Result Packets (FRPs). For example, UDP, TCP, FILE or LIBPCAP by calling the FilterResultAccess::SetSourceType function.

3) Set the appropriate source properties. For example, if the source is UDP, then specify the port number.

4) Call the FilterResultAccess: object's Start function to start monitoring for FRPs.

5) Call the FilterResultAccess: object's Get function to retrieve a filter result, represented by a FilterResult object.

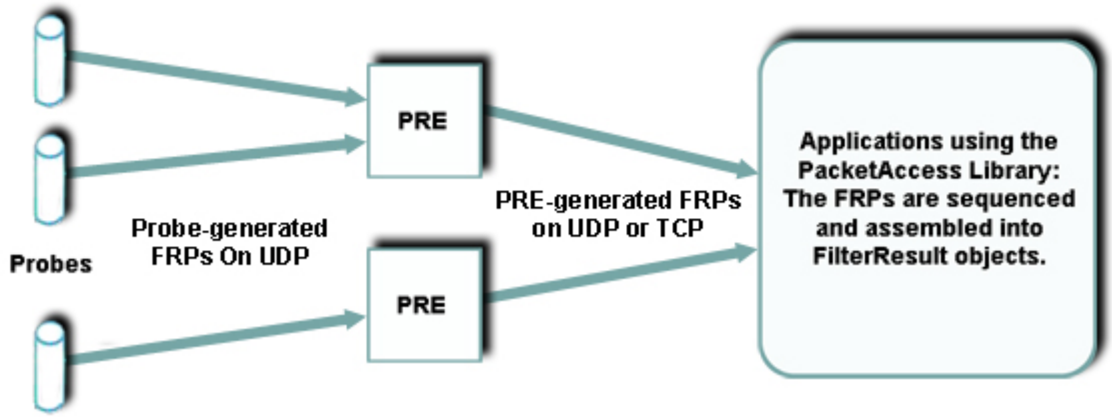
6) When a FilterResult object is no longer in use, call the DeleteFilterResult function to release resources used by the object.

7) To stop monitoring for FRPs, call the FilterResultAccess: object's Stop function.

8) Call the DeleteFilterResultAccess function to release resources used by the FilterResultAcess object.

Accessing Filter Results

SFProbes capture data packets (original captured packets) per their filtering instructions. Metadata (timestamps, SFProbe IDs, time synchronization information) is then added to the original captured packets to create Filter Results Packets (FRPs). The SFProbes forward the FRPs to their PREs and the PREs forward the FRPs to a host machine on UDP or TCP ports. A PacketPortal enabled application can use the PacketAccess API to retrieve these FRPs by instantiating a “FilterResultAccess” object. A FilterResultAccess object can return “FilterResult” objects representing a captured packet and its metadata. Depending on the size of the captured packet and the SFProbe settings, a FilterResult can be created from one or two filter result packets.



Accessing Metrics Results

SFProbes monitor the network packets and send metrics information to the PRE via Metrics Results Packets (MRPs).

Information in an MRP includes:

- Packet counts
- Byte counts
- Aggregate filtered counts
- Timing information
- SFProbe temperature
- SFProbe voltage
- SFProbe bias current

- SFProbe transmitted and received power

FilterResultAccess and MetricsResultsAccess Sources

A FilterResultAccess instance can be configured to retrieve FRPs or a MetricsResultsAccess instance can be configured to retrieve MRPs from one of the following sources:

- One or more UDP ports

User Datagram Protocol (UDP) uses a simple transmission model not requiring handshaking messages between the transmitter and the receiver. It does not require prior communications to set up transmission channels or paths. Error checking and correction is either assumed not to be necessary or it is performed in the application which avoids the additional overhead of processing. With UDP, the reduction of latency is emphasized over the reliability of the transmission which is important to time-sensitive applications. UDP does not allow multiple applications to listen to the same ports.

- One or more TCP ports

Transmission Control Protocol (TCP) emphasizes accurate delivery over timely delivery, thus providing reliable, ordered delivery of data. The reliability of the TCP transmission ensures that lost data or duplication does not occur when data is sent from one host to another. Relatively long delays can sometimes occur while waiting for packets that arrive out-of-order or retransmissions of lost packets. TCP does not allow multiple applications to listen to the same ports.

- One or more network interface cards (NICs) in promiscuous mode

When capturing using this source, the PacketAccess API uses libpcap (for Linux) or WinPcap (for Windows) to capture network traffic coming across NICs that are being monitored, bypassing the protocol stack. The PREs must be configured to send the FRPs using UDP for this source type to work.

- A file in pcap format

A captured file that is saved in the pcap format may be used as an input. This allows packets that have been captured and saved previously to be used. The PREs must have been originally configured to send the FRPs or MRPs using UDP for this source type to work.

Different sources cannot be mixed in one FilterResultAccess or MetricsResultsAccess instance. For example, if UDP is the selected source, then TCP, libpcap/WinPcap, and File are stopped and not allowed to be used as a source. However, the application may instantiate multiple instances of FilterResultAccess or MetricsResultsAccess at the same time. Note that transmission from the SFProbe to the PRE is always UDP. Only the transmission from the PRE to the application can be specified as listed above.

Basic PacketAccess API Library Usage to Obtain Metrics Results

Some concepts to be familiar with are:

- Objects are created, have a lifetime, then must be freed.

- Sources of input to an application using the API are TCP sockets, UDP sockets, and libpcap files captured from network devices.

- The mechanism for retrieving packets by the application is a polling mechanism.

In a typical environment, an application can retrieve metrics results by following these steps:

1) Create a MetricsResultAccess object by calling the CreateMetricsResultAccess function.

2) Specify the source of the Metrics Results Packets (MRPs). For example, UDP, TCP, FILE or LIBPCAP by calling the MetricsResultAccess::SetSourceType function.

3) Set the appropriate source properties. For example, if the source is UDP, then specify the port number.

4) Call the MetricsResultAccess object's Start function to start monitoring for MRPs.

5) Call the MetricsResultAccess object's Get function to retrieve a metrics result, represented by a MetricsResult object.

6) When a MetricsResult object is no longer in use, call the DeleteMetricsResult function to release resources used by the object.

7) To stop monitoring for MRPs, call the MetricsResultAccess object's Stop function.

8) Call the DeleteMetricsResultAccess function to release resources used by the MetricsResultAcess object.

SDK Gadgets and Tools

SDK Gadgets and Tools

Gadgets

SDK gadgets demonstrate the use of the PacketAccess API in the supported programming languages. They are also useful applications that can be used in test environments for software development.

Tools

SDK tools are provided to support development of PacketPortal applications. They are used to emulate and process the PacketPortal Filter Results Packets (FRPs) and Metrics Results Packets (MRPs).

Gadget: FilterResults

Purpose

This gadget is a very simple Filter Result Packet (FRP) consumer application. It demonstrates how to use PacketAccess API to receive FRPs from a data source and produce FilterResults (FRs). It can capture FRPs from a variety of sources including TCP, UDP or a PCAP file. When capturing from TCP or UDP, the PacketAccess API is configured with sequencing ON so packets are re-ordered based on time and sequence numbers (refer to Sequencing). The source code of this gadget is available to PacketPortal SDK customers.

Requirements

Refer to SDK Supported Operating Systems.

Examples

There are four versions of the FilterResults gadget, each written in a different programming language to illustrate the usage of the PacketAccess API in different programming environments.

C++ (Dynamic Linked VersionStatic Linked Version)

Usage: filterResults [UDP | TCP | filename] <port #> [output.pcap]

Captures filter result packets from a TCP port, a UDP port or a PCAP file. Optionally, this writes the payload of the filter result packets to a PCAP file.

Examples:

```
filterResults UDP 5120
```

Capture filter result packets from UDP port 5120 and prints result summary to standard output.

```
filterResults test.pcap
```

Read from test.pcap and emulate each packet in the file as a filter result packet.

```
filterResults TCP 5000 output.pcap
```

Capture filter result packets from TCP port 5000 and write the payload of the filter result packets to output.pcap

Java

Usage: java -jar filterResults.jar [UDP | TCP | filename] <port #>

Captures filter result packets from a TCP port, a UDP port or a PCAP file. Write result summary to standard output.

Perl

Usage: perl filterResults.pl [-p <port #> | -f <file.pcap>] [-e]

Captures filter result packets from a UDP port or a PCAP file. Optionally emulates the filter result packets.

Examples:

```
perl filterResults.pl -p 5000
```

Capture filter result packets from UDP port 5000 and prints result summary to standard output.

```
perl filterResults.pl -f test.pcap -e
```

Read from test.pcap and emulate each packet in the file as a filter result packet.

Python

Usage: `python  filterResults.py [-p <port #> | -f <file.pcap> ] [-e]`

Captures filter result packets from a UDP port or a PCAP file. Optionally emulates the filter result packets.

Examples:

```
python filterResults.py -p 5000
```

Capture filter result packets from UDP port 5000 and prints result summary to standard output.

```
python filterResults.py -f test.pcap -e
```

Read from test.pcap and emulate each packet in the file as a filter result packet.

Gadget: MetricsResults

Purpose

This gadget is a very simple Metrics Results Packet (MRP) consumer application. It demonstrates how to use PacketAccess API to receive MRPs from a data source and produce MetricsResults (MRs). It can capture MRPs from a variety of sources including UDP or a PCAP file. The source code of this gadget is available to PacketPortal SDK customers.

Requirements

Refer to SDK Supported Operating Systems.

Examples

There are four versions of the MetricsResults gadget, each written in a different programming language to illustrate the usage of the PacketAccess API in different programming environments.

C++ (Dynamic Linked Version or Static Linked Version)

Usage: `metricsResults [UDP | TCP | filename] <port #>`

Captures metrics result packets from a TCP port, a UDP port or a PCAP file.

Examples:

```
metricsResults UDP 5120
```

Capture metrics result packets from UDP port 5120 and prints result summary to standard output.

```
metricsResults test.pcap
```

Read from test.pcap and emulate each packet in the file as a metrics result packet.

Java

Usage: `java -jar metricsResults.jar [UDP | TCP | filename] <port #>`

Captures metrics result packets from a TCP port, a UDP port or a PCAP file. Write result summary to standard output.

Perl

Usage: `perl  metricsResults.pl [-p <port #> | -f <file.pcap> ] [-e]`

Captures metrics result packets from a UDP port or a PCAP file. Optionally emulates the metrics result packets. PacketAccess.pm and PacketAccess.so (linux) or PacketAccess.dll (Windows) should be in the same directory as the metricsResults.pl file. 32-bit Windows programs and Linux programs require Perl version 5.10.X. 64-bit Windows programs require Perl version 5.14.X.

Examples:

```
perl metricsResults.pl -p 5000
```

Capture metrics result packets from UDP port 5000 and prints result summary to standard output.

```
perl metricsResults.pl -f test.pcap -e
```

Read from test.pcap and emulate each packet in the file as a metrics result packet.

Python

Usage: `python  metricsResults.py [-p <port #> | -f <file.pcap> ] [-e]`

Captures metrics result packets from a UDP port or a PCAP file. Optionally emulates the metrics result packets. PacketAccess.pyd and PacketAccess.so (linux) or PacketAccess.dll (Windows) should be in the same directory as the metricsResults.pl file. 32-bit Windows programs and Linux programs require a Python version 2.6.X. 64-bit Windows programs require Python version 2.7.X.

Examples:

```
python metricsResults.py -p 5000
```

Capture metrics result packets from UDP port 5000 and prints result summary to standard output.

```
python metricsResults.py -f test.pcap -e
```

Read from test.pcap and emulate each packet in the file as a metrics result packet.

Tool: FilterResultsReplay

Purpose

This tool generates Filter Results Packets (FRPs) and transmits them using either TCP or UDP. The generated FRPs can be based from a PCAP file, captured from an Ethernet device, or internally-generated. FilterResultsReplay is a tool that you can use to test your PacketPortal-enabled application without having access to a live PacketPortal system. FilterResultsReplay attaches metadata (similar to the metadata that SFProbes attach to captured packets) to packets that are received from a live network. The system time is used as a timestamp and random probe IDs are assigned based on the number of probes that you input from the command line.

Requirements

Refer to SDK Supported Operating Systems.

Description

Usage:

```
filterResultsReplay -port <port#> -h <address> -source [file | custom | device]
```

Where:

-version	display version information
-port	send packets to this port, e.g. 8081
-host	send packets to this host, e.g. 10.10.2.3, localhost
-tcp	send packets using TCP (default is UDP)
-probes	number of probes to emulate (optional, default=1)
-t sec	stop after some number of seconds
-verbose	print more status
-quiet	print less status
-source file -filename	<file.pcap> [-loop <n>] [-realTime <1|0>]
-source custom	[-size <size> ...] [-total <filter results>]
-source libpcap	-device <device>

Examples:

Example #1: Generates FRPs from a file

```
filterResultsReplay -port 2001 -probes 5 -source file -fileName example.pcap -loop 100
-realTime 0
```

Generates FRPs based on packets in the example.pcap file. Each packet in the example.pcap file is emulated with one of five randomly-generated SFProbe IDs.

Depending on the size of the packet, each packet may be represented by one or two FRPs.

The file will be looped 100 times. If the "-realTime 0" is specified (default), then the first time, the timestamps for each packets in the PCAP file are used in the FRPs as probe timestamps. For subsequent iterations, the timestamps will be increased by the elapsed time between the last and the first packets in the file so that the filter results packets will appear to be progressing.

If the "-realtime -1" is specified instead, then the timestamps for the packets in the PCAP file are not used. Instead, the current system time is used as each FRP is generated.

Example #2: Replay FRPs from a file with filter results packets

```
filterResultsReplay -tcp -h 10.10.1.1 -port 2001 -probes 0 -source file -fileName
frp.pcap
```

Sends the filter results packets in the frp.pcap file to host 10.10.10.1 on TCP port 2001. Packets that are not filter results packets are ignored. Please note that the filter results packets in this file must have been originally sent by the PRE using UDP.

Example #3: Generates FRPs from libpcap interface

```
filterResultsReplay -h 10.10.10.1 -port 2001 -source libpcap -device eth0
```

Generates FRPs based on packets captured on eth0 interface and sends them to 10.10.10.1 on UDP port 2001. This assumes 10.10.10.1 is reachable through an interface other than eth0. Otherwise, if the filter results packets are sent to 10.10.10.1 through eth0, then an infinite number of packets will be generated since each outgoing filter results packet will be captured back and re-send out again.

For this to work properly, you need to have more than one interface. The output interface cannot be the same as the input interface. In this example, we are using eth0 as the input and eth1 as the output even though it is not apparent from the syntax.

Example #4: Generates FRPs internally

```
filterResultsReplay -h 10.10.10.1 -port 2001 -source custom -size 1500 -size 60 -total
1000
```

Generates FRPs with random bytes of specified sizes and sends them to 10.10.10.1 on UDP port 2001. Multiple sizes are allowed. When multiple sizes are provided, FRPs will be generated with sizes taken from the pool of provided values in a round-robin fashion. For packet sizes larger than 1418, two FRP packets will be generated.

If the “-total” parameter is specified, then the tool will generate that many FRs (between total and 2 times (2x) the total FRPs, depending on whether fragmentation is occurring) and exits.

Tool: MetricsResultsReplay

Purpose

This tool generates Metrics Results Packets (MRPs) and transmits them using either TCP or UDP. The generated MRPs can be based from a PCAP file or internally-generated. MetricsResultsReplay is a tool you can use to test your PacketPortal-enabled application without having access to a live PacketPortal system. When MetricsResultsReplay generates the MRPs, the system time is used as a timestamp and random probe IDs are assigned based on the number of probes that you input from the command line.

Requirements

Refer to SDK Supported Operating Systems.

Description

```
Usage:
metricsResultsReplay -port <port#> -host <address> [ -file <file.pcap> | -probes <n> ]
```

Where:

-version	display version information
-port	send packets to this port, e.g. 8081
-host	send packets to this host, e.g. 10.10.2.3, localhost
-tcp	send packets using TCP (default is UDP)
-max	send up to this number of packets and terminate (default is no max)

To replay metrics packets from a file:

-file	name of file containing metrics results packets

To emulate metrics packets (default mode):

-probes	number of probes to emulate (default=1)
-t	frequency of emulation in seconds (default = 1 second per probe)
-ts	timestamp interval used in packets generated (default = 0, use wall clock time)

Examples:

Example #1: Emulates 2 probes, sending packets in mrp.pcap to local host/2001 and ignoring non-metrics results packets

```
metricsResultsReplay -port 2001 -probes 2 -file mrp.pcap
```

Example #2: Emulates 2 probes, sending 1 packet per probe to 10.10.1.12/2001 every 2 seconds

```
metricsResultsReplay -host 10.10.1.12 -port 2001 -probes 2 -t 2
```

Testing FRPs in a Simulated Environment

If you are integrating the PacketAccess API in a custom application, you may not have access to SFProbes to capture live traffic. In that case, several methods are available to emulate SFProbes. The easiest method available is to use the PacketAccess API in emulation mode. In that case, the PacketAccess API will treat each non-FRP packet as if they are FRP packets.

There are many alternatives available using the PacketAccess API in emulation mode. Here are some examples:

- You may use one of the filter result trace files provided by PacketPortal. These PCAP files contain filter results packets (FRPs) of payload that represent real traffic, e.g. IP, RTP, VOIP, etc.

  Example code in C++:

  ```
  FilterResultAccess *pa = CreateFilterResultAccess();

  pa->SetSourceType("file");

  pa->SetSourceProperty("filename", "frp_trace.pcap");

  // ...

  int timeout = 1000; // 1 second

  FilterResult *pResult = pa->Get(timeout);

  // ...
  ```

- You may use a regular PCAP file with traffic suitable for your application, and turn on the API's emulation mode.

  Example code:

```
FilterResultAccess *pa = CreateFilterResultAccess();

pa->SetSourceType("file");

pa->SetSourceProperty("filename", "my_trace.pcap");

pa->Emulate(true);

// ...

int timeout = 1000; // 1 second

FilterResult *pResult = pa->Get(timeout);

// ...
```

- You may listen on an Ethernet interface that has traffic you want to use, and turn on the API's emulation mode.

Example code:

```
FilterResultAccess *pa = CreateFilterResultAccess();

pa->SetSourceType("libpcap");

pa->SetSourceProperty("device", "eth0");

pa->Emulate(true);

// ...

int timeout = 1000; // 1 second

FilterResult *pResult = pa->Get(timeout);

// ...
```

The SDK also contains some sample tools like filterResults or filterResultsReplay that are built on top of the PacketAccess API using the same concepts. As an example, you may use one as an FRP generator (such as filterResultsReplay) and another one (such as filterResults) as an FRP consumer. You could install them on different computers or the same computer and configure them to communicate using either TCP or UDP.

An example of this is shown below:

```
filterResults UDP 50000

filterResultsReplay –h localhost –port 50000 –source file –fileName trace.pcap
```

In this example,

- filterResultsReplay emulates only a single SFProbe. If you want to emulate multiple SFProbes you can use multiple instances with separate trace files.

- filterResults time sequences packets from multiple SFProbes. So when using multiple incoming feeds from multiple SFProbes, the systems they are sent from must be time synchronized.

If you are running filterResults and filterResultsReplay on separate computers, FRPs will go through many buffers (such as, switches, routers, NICs, TCP/UDP), so they are susceptible to being dropped. Even when running both tools on the same computer, FRPs will go through some local buffers. For example, when replaying a trace locally, it is common to overrun the local TCP/UDP receive buffers.

A possible path for testing the PacketAccess API with a custom application might be by:

- Testing in emulation mode with local traces emulating a single SFProbe

- Replaying local traces with filterResultReplay through a local TCP/UDP port emulating a single SFProbe

- Replaying from a remote computer traces with filterResultsReplay through a TCP/UDP/Libpcap source emulating a single SFProbe

- Replaying from a remote computer traces with filterResultsReplay through a TCP/UDP/Libpcap source emulating multiple SFProbes

Testing MRPs in a Simulated Environment

If you are integrating the PacketAccess API in a custom application, you may not have access to SFProbes to capture live traffic. In that case, several methods are available to emulate SFProbes. The easiest method available is to use the PacketAccess API in emulation mode.

- In addition, you may use one of the metrics results trace files provided by PacketPortal.

  Example code in C++:

```
MetricsResultAccess *pa = CreateMetricsResultAccess();

pa->SetSourceType("file");

pa->SetSourceProperty("filename", "mrp_trace.pcap");

// ...

int timeout = 1000; // 1 second

MetricsResult *pResult = pa->Get(timeout);

// ...
```

The SDK also contains some sample tools like metricsResults or metricsResultsReplay that are built on top of the PacketAccess API using the same concepts. As an example, you may use one as an MRP generator (such as metricsResultsReplay) and another one (such as metricsResults) as an MRP consumer. You could install them on different computers or the same computer and configure them to communicate using either TCP or UDP.

An example of this is shown below:

```
metricsResults UDP 50000

metricsResultsReplay –h localhost –port 50000 -probes 1
```

Chapter 4. PacketAccess API

PacketAccess API

The API can be directly invoked from C++, Java, Perl and Python. The API is nearly identical in all the language bindings.

A Filter Results Packet (FRP) is a custom PacketPortal packet that is used to transfer the original captured packet and its associated metadata to its final destination. The FRPs are generated by the SFProbes and forwarded to the application via the Packet Routing Engine (PRE). Depending on the filter setup, the entire original captured packet or just the header information of the original captured packet is contained in the FRP.

A Metrics Results Packet (MRP) is a custom PacketPortal packet that is used to transport metrics information to the application. The MRPs are generated by the SFProbes and forwarded to the application via the Packet Routing Engine (PRE). The System Manager can be used to configure how often these packets are generated by the SFProbes and where the destination should be.

The PREs can be configured to send the FRPs and MRPs to an application via TCP or UDP transport. A PacketAccess-enabled application can process FRPs and MRPs from one or more PREs in real time (or from a PCAP file.) If the PREs are using a UDP transport to forward the FRPs, the PacketAccess-enabled application can retrieve the FRPs directly from a network interface. A PacketAccess-enabled application can also process FRPs from a PCAP file.

The MRPs and FRPs may be forwarded to the same application, but not to the same port. (For example, you cannot set MRPs and FRPs to be forwarded to the same machine on the same UDP port. There will be no errors; instead either all the MRPs or all the FRPs will be dropped by the PacketAccess API).

The packet data and metadata can be retrieved through the API by querying a FilterResult (FR) object. The FR is returned by calling the FilterResultAccess object's Get function.

PacketAccess C++ (Dynamic Linked Version) API

C++ (Dynamic Linked Version, Windows Only) API Library

This section describes the API Library for the C++ Dynamic Linked version. This version is run only on Microsoft Windows.

Header Files

The following header files provide access to the C++ Dynamic Linked (Windows Only) Version of the PacketAccess API classes and functions.

Header File	Library
PacketAccessDLL.h	packetAccessDLL.lib (import library) packetAccessDLL.dll (the dynamic library)

Overview

The following are available in the dynamic-linked version of the C++ API library. This version is used only with the Microsoft Windows operating system. Each of these classes is described briefly in the table below and in detail later in this section.

Global Functions	Global functions allow you to access version information, create and delete PacketAccess objects, etc.
PAString	PAString represents a sequence of characters. The primary purpose of this class is to pass string parameters between the application and the PacketAccess library.
PAStrings	PAStrings represents an ordered collection of strings. The primary purpose of this class is to pass a collection of string objects between the application and the PacketAccess library.
FilterResult	The FilterResult class represents an original captured packet and its meta-data. A pointer to this object is obtained through the FilterResultAccess class.
MetricsResult	The MetricsResult class represents information contained in a metrics packet. A pointer to this object is obtained through the MetricsResultAccess class, or created from previously obtained metric data.
PacketSourceTypeInfo	PacketSourceTypeInfo provides information on a packet source supported by the PacketAccess Library.
PacketSourceTypeInfoList	PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling the global function CreatePacketSourceTypeInfoList.

PacketResultAccess	The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.
FilterResultAccess	The FilterResultAccess class retrieves FilterResult objects.
MetricsResultAccess	The MetricsResultsAccess class retrieves MetricsResult objects.

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Global Functions

Global Functions allow you to access version information, create and delete PacketPortal objects, etc.

`int GetMajorVersion()`

Description: This function returns the major version number.

Parameters: None

Return Value: Returns the major version number

`int GetMinorVersion()`

Description: This function returns the minor version number.

Parameters: None

Return Value: Returns the minor version number

`int GetPatchVersion()`

Description: This function returns the patch version number.

Parameters: None

Return Value: Returns the patch version number

`int GetBuildVersion()`

Description: This function returns the build version number.

Parameters: None

Return Value: Returns the build version number

`unsigned int GetBuildTime()`

Description: This function returns the build time.

Parameters: None

Return Value: Returns the build time in number of seconds since January 01, 1970, 00:00:00.

`void GetVersion(PAString& version)`

Description: This function returns a version string representing the version information.

Parameters: version
Type: PAString&
The version information is stored to the PAString reference.

Return Value: None

Example:

```
PAString ver;

GetVersion(ver);

printf("\nPacketPortal Library Version: %s\n", ver.c_str());
```

`FilterResultAccess *CreateFilterResultAccess()`

Description: This function creates a FilterResultAccess object.

Parameters: None

Return Value: Returns a pointer to the FilterResultAccess object

Remarks: Call DeleteFilterResultAccess to release resources used by the FilterResultAccess object.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

// …

DeleteFilterResultAccess(pAccess);
```

`MetricsResultAccess *CreateMetricsResultAccess()`

Description: This function creates a MetricsResultAccess object.

Parameters: None

Return Value: Returns a pointer to the MetricsResultAccess object

Remarks: Call DeleteMetricsResultAccess to release resources used by the MetricsResultAccess object.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

// …

DeleteMetricsResultAccess(pAccess);
```

`void DeleteFilterResultAccess(FilterResultAccess *p)`

Description: This function deletes a FilterResultAccess object.

Parameters: p

Type: FilterResultAccess *
A pointer to a valid FilterResultAccess object.

Return Value: None

Remarks: A FilterResultAccess object is created by using CreateFilterResultAccess.

Example: See the example in CreateFilterResultAccess

`void DeleteMetricsResultAccess(MetricsResultAccess *p)`

Description: This function deletes a MetricsResultAccess object.

Parameters: p
Type: MetricsResultAccess *
A pointer to a valid MetricsResultAccess object.

Return Value: None

Remarks: A MetricsResultAccess object is created by using CreateMetricsResultAccess.

Example: See the example in CreateMetricsResultAccess

`void DeleteFilterResult(FilterResult *p)`

Description: This function deletes a FilterResult object.

Parameters: p
Type: FilterResult *
A pointer to a valid FilterResult object.

Return Value: None

Remarks: A pointer to a FilterResult object is returned by calling FilterResultAccess object's Get functions, when filter result becomes available from the FilterResultAccess object.
Call DeleteFilterResult to release resources used by the FilterResult object.

`MetricsResult *CreateMetricsResult(const void * buffer, int bufferSize)`

Description: This function creates a MetricsResult object from data stored in the buffer.

Parameters: buffer
Type: const void * buffer
A pointer to metrics data. Metrics data can be returned by calling the MetricsData function from an existing MetricsResult instance. These two functions allow an application to store and retrieve the metrics data as a byte array.

Return Value: Returns a pointer to a MetricsResult.

Remarks: This function always returns a MetricsResult pointer, even if the buffer may point to invalid data. You may receive invalid data when you call functions in MetricsResult if the buffer contains invalid data or of insufficient size.

An application should release the memory used by this MetricsResult by calling DeleteMetricsResult() after it is no longer needed.

`void DeleteMetricsResult(MetricsResult *p)`

Description: This function deletes a MetricsResult object.

Parameters: p
Type: MetricsResult *
A pointer to a valid MetricsResult object.

Return Value: None

Remarks: A pointer to a MetricsResult object is returned by calling MetricsResultAccess object's Get functions, when Metrics result becomes available from the MetricsResultAccess object.
Call DeleteMetricsResult to release resources used by the MetricsResult object.

`PacketSourceTypeInfoList *CreatePacketSourceTypeInfoList()`

Description: This function creates a PacketSourceTypeInfoList object.

Parameters: None

Return Value: Returns a pointer to PacketSourceTypeInfoList object

Remarks: Call DeletePacketSourceTypeInfoList to release resources used by the object

Example:

```
PacketSourceTypeInfoList *pList = CreatePacketSourceTypeInfoList();

GetPacketSourceTypeInfo(pList);

for (int i = 0; i < pList->Size(); i++)

{

   PacketSourceTypeInfo *pInfo = pList->Get(i);
```

```
        PAString type;
        PAString desc;
        PAString help;
        pInfo->Name(type);
        pInfo->Description(desc);
        pInfo->HelpText(help);
        printf("\n%s (%s):\n%s\n", type.c_str(), desc.c_str(), help.c_str());
    }
    DeletePacketSourceTypeInfoList(pList);
```

void GetPacketSourceTypeInfo(PacketSourceTypeInfoList *pList)

Description: This function fills in the PacketSourceTypeInfoList object with all the source types supported by the PacketAccess Library.

Parameters: pList
Type: GetPacketSourceTypeInfo *
The packet source type information is stored in pList.

Return Value: None

Remarks: The application can use this function to dynamically find out all the packet source types supported by the PacketAccess library. The name of the source type can be passed to the PacketResultAccess object's SetSourceType function.

Example: See example in CreatePacketSourceTypeInfoList

void DeletePacketSourceTypeInfoList(PacketSourceTypeInfoList *pList)

Description: This function deletes the PacketSourceTypeInfoList object.

Parameters: pList
Type: PacketSourceTypeInfoList *
A pointer to the PacketSourceTypeInfoList.

Return Value: None

Remarks: A pointer to a FilterResult object is returned by calling CreatePacketSourceTypeInfoList.

Example: See example in CreatePacketSourceTypeInfoList

Class: PAString

PAString represents a sequence of characters. The primary purpose of this class is to pass string parameters between the application and the PacketPortal library.

Methods

Each of the PAString class methods is described below.

`PAString()`

Description: Content is initialized to an empty string.

Parameters: None

`PAString(const char *value)`

Description: Content is initialized to the value pointed to by the character array.

Parameters: value
Type: const char *
A pointer to a null-terminated array of characters.

Remarks: The length of the character array is determined by the first occurrence of a null character.

`PAString(const char *value, size_t length)`

Description: Content is initialized to the first length characters in the array of characters pointed to by value.

Parameters: value
Type: const char *
A pointer to character array used to initialize the object.

length
Type: size_t
The number of characters to use for the array.

`PAString(const PAString& s)`

Description: Content is initialized to a copy of the PAString object.

Parameters: s
Type: const PAString&
The PAString reference used to initialize the object.

`PAString& operator=(const PAString& rhs)`

Description: Content is set to the value of the PAString.

Parameters: rhs
Type: const PAString&
The PAString reference used for the content of the object.

Return Value: The string that is being assigned.

`void assign(const char *value)`

Description: Content is set to the value pointed to by the character array

Parameters: valueconst
Type: char *
A pointer to a null-terminated array of characters.

Remarks: The length of the character array is determined by the first occurrence of a null

`void assign(const char *value, size_t length)`

Description: Content is set to the first length characters in the array of characters pointed to by value.

Parameters: value
Type: const char *
A pointer to character array used to initialize the object.

length
Type:size_t
A number of characters to use for the array.

Return Value: None

```
const char *data() const
```

Description: Returns a pointer to an array of characters with the same content as the string.

Parameters: None

Return Value: A pointer to an array of characters.

```
const char *c_str() const
```

Description: Generates and returns a null-terminated sequence of characters with the same content as the string.

Parameters: None

Return Value: A pointer to an array of characters.

```
size_t length() const
```

Description: Returns a count of the number of characters in the string.

Parameters: None

Return Value: A count of the number of characters in the string.

`void clear()`

Description: Assign the string to an empty string.

Parameters: None

Return Value: None

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: PAStrings

PAStrings represents an ordered collection of strings. The primary purpose of this class is to pass a collection of string objects between the application and the PacketPortal library.

Methods

Each of the PAStrings class methods is described in the table below.

`PAStrings()`

Description: Initialize the content to an empty collection.

Parameters: None

Return Value: None

`PAStrings(const PAStrings& rhs)`

Description: Initialize the content to a copy of the strings contained in the PAStrings object.

Parameters: rhs
Type: const PAStrings&
The PAStrings reference used to initialize the object.

Return Value: None

`PAStrings& operator=(const PAStrings& s)`

Description: Set the content to a copy of the strings contained in the PAStrings object.

Parameters: s
  Type: const PAStrings&
  The PAStrings reference used for the content of the object.

Return Value: The PAStrings that is being assigned.

`void push_back(const char *value)`

Description: Add a new string initialized to the value pointed to by the character array to the end of the collection.

Parameters: value
  Type: const char *
  A pointer to a null-terminated array of characters used for the new string object.

Return Value: None

Remarks: The new string is created using value. The length of the new string is determined by the first occurrence of a null character.

`void push_back(const char *value, size_t length)`

Description: Add a new string initialized to the value pointed to by the character array with the specified length to the end of the collection.

Parameters: value
  Type: const char *
  The pointer to character array used for the new string object.

length
  Type: size_t
  The number of characters to use for the array.

Return Value: None

Remarks: The new string is created using *value* and *length*.

`void push_back(const PAString& s)`

Description: Add a new string initialized to the value of the PAString to the end of the collection.

Parameters: s
Type: const PAString&
PAString reference used for the new string object.

Return Value: None

Remarks: The new string is created using s and added to the end of the collection.

`void clear()`

Description: All the objects in the collection are removed.

Parameters: None

Return Value: None

Remarks: The size of the collection is zero after clear is called.

`void at(size_t index, PAString& s) const`

Description: Copy the content of the object at position index to s.

Parameters: index
Type: size_t
The position of the object to be copied.

s
Type:PAString&
The PAString reference that is used to store the string content.

Return Value: None

Remarks: Index positions start at zero. If index is out of range, then s is set to an empty string.

`size_t size() const`

Description: Returns the number of string objects in the collection.

Parameters: None

Return Value: The number of string objects in the collection

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: FilterResult

The FilterResult class represents an original captured packet and its meta-data. A pointer to this object is obtained through the FilterResultAccess class.

Methods

Each of the FilterResult class methods is described below.

`int Version() const`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Library version.

`void ProbeId(PAString& s) const`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: s

Type: PAString&
The probe ID is stored in the PAString.

Return Value: None

Remarks: A probe ID is not null-terminated. Applications should use PAString's length function to return the length of the probe ID string.

`unsigned int Seconds() const`

Description: The "Seconds" portion of the timestamp. This value may or may not be the same as the "ProbeSeconds" value depending on sequencing rules. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: If sequencing is turned on, a FilterResult's timestamp may be adjusted. See the Sequencing section in the Understanding Filter Results chapter for more information.

`unsigned int NSeconds() const`

Description: Returns the "nanoseconds" portion of the timestamp. This value may or may not be the same as the "ProbeNSeconds" value depending on sequencing rules.

Parameters: None

Return Value: Returns the "nanoseconds" portion of the timestamp.

`unsigned int Sequence() const`

Description: Returns an unsigned 32-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a FilterResult is missing.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();


//... setup source and source properties
bool bNewSequence = true;
unsigned int lastSequence = 0;
FilterResult *pResult;


const int timeout = 1000; // 1 second
while ((pResult = pAccess->Get(timeout)) != NULL)
{
  // for this example, let’s assume that all
  // FilterResults comes from the same probe.
  if (bNewSequence)
  {
    lastSequence = pResult->Sequence();
    bNewSequence = false;
  }
  else
  {
    unsigned int currentSequence = pResult->Sequence();
    if (lastSequence + 1 != currentSequence)
    {
       // one or more FilterResult is missing …
       // the above expression works with
       // sequence number wrapping since it is
       // an unsigned value.
    }
    lastSequence = currentSequence;
  }
```

```
    // ...

    DeleteFilterResult(pResult);

  }

  pAccess->Stop();

  DeleteFilterResultAccess(pAccess);
```

`unsigned int FilterMatchBits() const`

Description: Returns a value that represents which filters are matched

Parameters: None

Return Value: A value that represents which filters are matched.

`unsigned int CongestionCount() const`

Description: The number of packets that has matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow.

Parameters: None

Return Value: Returns a 29-bit value representing packets that matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow. This counter only applies to the side for this filtered packet.

Remarks: Only 29-bits of this value are valid. There are two congestion counters, one for equipment side and one for network side. When the SFProbe is unable to process a packet due to buffer overflow, it increments this counter for the side of this filtered packet.

Since the sequence number is not incremented in this situation, the application may receive filter results with consecutive sequence numbers, when in fact there are missing filtered packets. When the packets that the SFProbe are unable to process are on the same side as the next successfully injected filter result packets, application can check the CongestionCount for potential packet loss.

`unsigned int InjectedCount() const`

Description: Returns the number of captured packets that the SFProbe has successfully injected.

Parameters: None

Return Value: The number of captured packets that the SFProbe has successfully injected.

`bool IsBadFCS() const`

Description: Returns whether the original captured packet has a bad FCS.

Parameters: None

Return Value: Returns true if the original captured packet has a bad FCS.

`bool IsHeaderOnly() const`

Description: Returns whether the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Remarks: If the original capture packet matches more than 1 filter, and not all of them has the "headers only" setting, then the captured payload may contain more than the protocol headers.

`bool IsInjectNet() const`

Description: Returns whether the filter result was injected on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the filter result was injected on the network side of the SFProbe, otherwise returns false.

Remarks: This flag is not related to whether the original captured packet is on the network or equipment side of the SFProbe.

`bool IsLate() const`

Description: Returns whether the filter result is considered late.

Parameters: None

Return Value: Returns true if the filter result is considered late. Otherwise returns false.

Remarks: A filter result is considered late if the application has already retrieved a filter result with a more recent timestamp. The application can optionally discard these filter results by calling FilterResultsAccess object's DiscardLate(false) function.

`bool IsNet() const`

Description: Returns whether the original captured packet is on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the original captured packet was captured on the network side of the SFProbe. Otherwise returns false, indicating the original captured packet was captured on the equipment side.

`bool IsNewSequence() const`

Description: Returns whether the filter result indicates a new sequence.

Parameters: None

Return Value: Returns true if the filter result indicates a new sequence. Otherwise returns false.

Remarks: A filter result is considered a new sequence depending on sequencing rules. A filter result can be considered a new sequence if it is the first filter result from a particular SFProbe; a filter result that has a sequence number that is sufficiently far away from the previous filter result's sequence number from the same SFProbe; or if this filter result arrives a long time after other filter results. The SequenceBreakpoint and Timebreakpoint functions of FilterResultAccess can be used to adjust the breakpoint values.

`bool IsOnlyRoute() const`

Description: Returns whether this machine and port is the only recipient of this FilterResult.

Parameters: None

Return Value: Returns true if this machine and port is the only recipient of this FilterResult, otherwise returns false.

Remarks: If this machine and port is the only recipient of this FilterResult, then a missing sequence in the filter result indicates that a filter result is unable to reach the application. If multiple machine and ports can be the intended recipients of this FilterResult, then a missing sequence number only indicates that a filter result may be missing.

`bool IsSliced() const`

Description: Returns whether the filter expression requested the SFProbe to slice the payload of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to slice the payload of the original captured packet. If slicing was not been requested, a false is returned.

Remarks: This value indicates the setting of the filter used to capture the original captured packet. The captured payload may not necessarily be truncated. If the original packet is too big, the captured payload may be truncated even if the filter does not specify truncation.

To determine if the Filter Result object is sliced, you can compare the "real packet length" and "payload length" of the Filter Result object. The payload is sliced if either of the two following conditions are true:

- the real packet length is zero
- the payload length is less than the real packet length

The "real packet length" and "payload length" are methods documented later in the FilterResult class.

`bool IsTimingLock() const`

Description: Returns whether the filter result is captured when the SFProbe is time synchronized with the PRE.

Parameters: None

Return Value: Returns true if the filter result is captured when the SFProbe is time synchronized with the PRE. Otherwise returns false.

Remarks: If the SFProbe is not time synchronized with the PRE and original captured packets are expected to be captured by multiple SFProbes, then the timestamps may not reflect the true packet order. In that case, the application may consider either turning off sequencing, or setup time synchronization for the SFProbes involved.

`bool WasFragmented() const`

Description: Returns whether the filter result was assembled from two filter result packets.

Parameters: None

Return Value: Returns true if the filter result was assembled from two filter result packets. Otherwise returns false.

Remarks: If the captured payload is over a specified limit (usually around the MTU of the network), then two filter result packets are needed to carry the metadata and the original captured packet as payload. This function is useful if the application wants to identify this situation.

`int RealPacketLength() const`

Description: Returns the original packet length in bytes, if known. This length does not include the 4 byte FCS of the original packet.

Parameters: None

Return Value: Returns the original packet length in number of bytes.

Remarks: The maximum number of bytes counted by the probe depends on its configuration and network encapsulation. Typically, the maximum number is around the maximum MTU size, or up to around 2000 bytes. When the actual number of bytes in the original packet is not known, the function returns 0.
The application can determine if the payload returned for the filter result is sliced by comparing the return value of PayloadLength function and RealPacketLength function. The payload for the filter result is sliced if either

of the following conditions exist:

- If RealPacketLength returns zero
- If PayloadLength is less than RealPacketLength

`int PayloadLength() const`

Description: Returns the length of the payload captured.

Parameters: None

Return Value: Returns number of bytes of the payload captured.

`void Payload(PAString& s) const`

Description: Copies the payload to a PAString.

Parameters: s
Type: PAString&
The payload is stored in the PAString

Return Value: None

Remarks: The captured payload maybe truncated by the probe depending on probe configuration. The payload does not include the 4 byte FCS.

`int Payload(void *buffer, int bufferSize) const`

Description: Copies the payload to a buffer up to a specified size.

Parameters: buffer
Type: void *
The payload is copied to the buffer up to bufferSize bytes.

bufferSize
Type: int
Specifies the maximum number of bytes to copy.

Return Value: Returns the number of bytes copied.

Remarks: The application is responsible to provide a valid buffer of the specified size. The captured payload maybe truncated by the probe depending on probe configuration. The payload does not include the 4 byte FCS.

`unsigned int ProbeSeconds() const`

Description: Returns the "seconds" portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The "seconds" portion of the real time that the original packet is captured by the SFProbe.

`unsigned int ProbeNSeconds() const`

Description: Returns the "nanoseconds" portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The "nanoseconds" portion of the real time that the original packet is captured by the SFProbe.

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: MetricsResult

The MetricsResult class represents metrics result packets generated by the SFProbe. A pointer to this object is obtained through the MetricsResultAccess class, or the global function CreateMetricsResult.

Example

```
#include <AtlBase.h>

#include <AtlConv.h>

#include <tchar.h>
```

```
#include <stdio.h>

#include <stdlib.h>

#include "../PacketAccessDLL.h"

#include <exception>


using namespace PacketAccessDLL_NS;


#include <Windows.h>

...

void PrintSourceInfo(MetricsResultAccess *pa)

{

   USES_CONVERSION;


   PAStrings propNames;

   pa->GetSourcePropertyNames(propNames);

   for (size_t i = 0; i < propNames.size(); i++)

   {

      PAString name;

      propNames.at(i, name);


      PAStrings propValues;

      pa->GetSourceProperties(name.c_str(), propValues);

      for (size_t j = 0; j < propValues.size(); j++)

      {

         PAString value;


         propNames.at(i, name);

         propValues.at(j, value);
```

```
            _tprintf(TEXT("%s: %s\n"), A2T(name.c_str()), A2T(value.c_str()));
        }
    }
}

void PrintResult(MetricsResult *pResult)
{
    if (_bVerbose)
    {
        PAString probeId;
        pResult->ProbeId(probeId);

        unsigned long seconds = pResult->Seconds();
        unsigned long nSeconds = pResult->NSeconds();
        unsigned int  seq = pResult->Sequence();
            unsigned int  netPacketCount = pResult->NetPacketCount();
            unsigned int  eqtPacketCount = pResult->EqtPacketCount();

        _tprintf(TEXT("Probe ["));
        PrintBinary(probeId.data(), probeId.length());
            _tprintf(TEXT("] %09lu.%09lu: seq [%lu] net packets: %10d, eqt
packets: %10d\n"),
                    seconds,
                    nSeconds,
                    seq,
                          netPacketCount,
                          eqtPacketCount);
    }
```

```
    else
    {
        static unsigned long long count = 0;
        count++;
        if ((count % 60)== 0)
            _tprintf(TEXT("\n"));
        else
            _tprintf(TEXT("."));
    }
}

void PrintBinary(const char *s, size_t length)
{
         // xx:xx:xx:xx:xx:xx format
    for (size_t i = 0; i < length; i++)
    {
        _tprintf(TEXT("%02x"), (unsigned int) (s[i] & 0xFF));
        if (i < length - 1)
            _tprintf(TEXT(":"));
    }
}
```

Methods

Each of the MetricsResult class methods is described below.

`bool IsPRETimeSync() const`

Description: Indicates whether the PRE is time synced with the wall clock.

Parameters: None

Return Value: Returns true if the PRE is time sync with the wall clock.

Remarks:

The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off.

When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

`unsigned int PRETimeSyncLossCount() const`

Description: Indicates the number of times the PRE has lost time sync with the wall clock.

Parameters: None

Return Value: Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

Remarks: If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

`int Version() const`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the metrics result packet format version associated with this object. It is not related to the PacketAccess Library version.

`void ProbeId(PAString& s) const`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: s
  Type: PAString&
  The probe ID is stored in the PAString.

Return Value: None

Remarks: A probe ID is not null-terminated. Applications should use PAString's length function to return the length of the probe ID string.

`unsigned int Seconds() const`

Description: Returns the “Seconds” portion of the timestamp. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: MetricsResults are returned to the application in a first-in, first-out manner. The application may receive a MetricsResult with an earlier timestamp than the previous MetricsResult it receives. This is very common if there are multiple probes in different parts of the network sending MetricsResults to the same MetricsResultsAccess object, or if the probes are not time synchronized with the PRE.

`unsigned int NSeconds() const`

Description: Returns the “nanoseconds” portion of the timestamp.

Parameters: None

Return Value: Returns the “nanoseconds” portion of the timestamp. See remarks on Seconds().

`unsigned int Sequence() const`

Description: Returns an unsigned 16-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a MetricsResult is lost. A MetricsResult can be lost in transit, or due to buffer overflow.

A gap in the sequence number indicates that an intended MetricsResult for that probe was not delivered. In most cases, an application can safely ignore this situation.

There are two situations an application may want to re-baseline its counters if there is a skipped MetricsResult:

1. If an application is interested in the filter byte counters. The filter byte counter may become invalid if a jumbo packet (more than about 2000 bytes) has been filtered. In that case, the filter byte counter invalid flag for that filter slot will be set. This flag is cleared for each Metrics Result request. If there is a lost Metrics Result, the application may not be aware that the filter byte counter invalid flag has been reset.

2. Under rare situations, if there are too many missed sequence numbers, then the counters may rollover more than once. Different counters rollover at different rates, depending on the counter's capacity and network traffic volume,. The application should decide when the number of missed sequences may cause a double rollover.

For example, the theoretical maximum number of Ethernet frames per second on a 1G network is around 1.4 million frames per second, and the 29-bit total packet count can count up to around 536 million packets. So packet counter may rollover around every 6 minutes. If the metrics result request interval (configurable through System Manager) is every 5 minute, then missing two consecutive Metrics Results may cause a double rollover.

`unsigned int ResetCount() const`

Description: Returns a value that represents how many times the application should treat this metrics result as a new baseline.

Parameters: None

Return Value: None

Example:

```
MetricsResultAccess *pAccess =
CreateMetricsResultAccess();

// …

const long timeout = 1000; // one second

bool isFirst = true;

unsigned short lastResetCount = 0;

unsigned short lastSequence = 0;

std::string reason;
```

```
MetricsResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

    bool shouldbaseline = false;

    if (isFirst)

    {

        shouldbaseline = true;

        isFirst = false;

        reason = "first metrics result";

    }

    else if (lastResetCount != (unsigned short) pResult-
>ResetCount())

    {

       shouldbaseline = true;

        reason = "new reset count, the metrics feed may
be stopped and restarted.";

    }

    else if ((unsigned short)((lastSequence + 1)&0xffff)
!= (unsigned short) pResult->Sequence())

    {

        // skipping one sequence number may not affect
the counters you are interested in.


       shouldbaseline = true;

        reason = "sequence number is skipped, some
counters may have double rollover.";

    }


    // find out if there are any invalid bits in the
filter byte counter
```

```
    for (int i = 0; i < 8; i++)

    {

        if (pResult->NetFilterByteCountInvalid(i) ||
pResult->EqtFilterByteCountInvalid(i))

        {

           shouldbaseline = true;

            reason = "Invalid filter byte counter";

        }

    }


    lastResetCount = (unsigned short) pResult-
>ResetCount();

    lastSequence = (unsigned short) pResult->Sequence();


    if (shouldbaseline)

    {

        // log the situation and reset counters that may
affect your measurement

    }


    // ... more processing

}
```

`unsigned int RetryCount() const`

Description: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Parameters: None

Return Value: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Remarks: This value may be important to applications that want to determine if the missing sequence number is due to the PRE unable to transmit or receive metrics results to and from the SFProbe.

`int SFFTemperature() const`

Description: Temperature of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit signed integer in increments of 1/256 °C

`unsigned int SFFVcc() const`

Description: Supply voltage of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit unsigned integer in increments of 100 μV.

`unsigned int SFFTxBias() const`

Description: Laser bias current of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit unsigned integer in increments of 2 μV.

`unsigned int SFFTxPower() const`

Description: Transmitted average optical power of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit unsigned integer in increments of 0.1 μW.

```
unsigned int SFFRxPower() const
```

Description: Received average optical power of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit unsigned integer in increments of 0.1 μW.

```
int  M2SAverageNSecond() const
```

Description: The average time needed for a packet to travel from PRE to SFProbe.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the PRE to the SFProbe.

```
int S2MAverageNSecond() const
```

Description: The average time in nanoseconds needed for a packet to travel from SFProbe to PRE.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the SFProbe to the PRE.

```
int TimingOffset() const
```

Description: M2SAverageNSecond minus the average of M2SAverageNSecond and S2MAverageNSecond. M2S - (M2S + S2M) / 2

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average round trip latency between the PRE to the SFProbe.

`bool IsTimingValid() const`

Description: Indicates whether the IsTimingLock return value is valid.

Parameters: None

Return Value: Boolean value indicating "true" if the time IsTimingLock is valid.

Remarks: This value should be used in conjunction with the IsTimingLock.

`bool IsTimingLock() const`

Description: Indicates whether the SFProbe is in time synchronization with the PRE at the time this packet is generated.

Parameters: None

Return Value: Boolean value indicating "true" if the time is synchronized between the PRE and the API.

Remarks: None

`unsigned long long EqtByteCount() const`

Description: Total number of bytes on the EQT side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the EQT side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`unsigned long long NetByteCount() const`

Description: Total number of bytes on the NET side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the NET side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`unsigned int EqtPacketsFiltered() const`

Description: Total number of packets filtered on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered on the EQT side.

Remarks: None

`unsigned int EqtPacketsInjected() const`

Description: Total number of packets injected by the SFProbe on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

`unsigned int NetPacketsFiltered() const`

Description: Total number of packets filtered by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered by the SFProbe on the NET side.

Remarks: None

```
unsigned int  NetPacketsInjected() const
```

Description: Total number of packets injected by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the NET side.

Remarks:

```
unsigned int  EqtPacketCount() const
```

Description: Number of packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of packets filtered by the SFProbe on the equipment side.

Remarks: None

```
unsigned int  EqtIPv4Count() const
```

Description: Number of IPv4 packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

```
unsigned int  EqtIPv4MulticastCount() const
```

Description: Number of IPv4 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of “1110” (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

`unsigned int EqtIPv4BroadcastCount() const`

Description: Number of IPv4 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has the IPv4 destination address of 255.255.255.255.

`unsigned int EqtIPv6Count() const`

Description: Number of IPv6 packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`unsigned int EqtIPv6MulticastCount() const`

Description: Number of IPv6 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of multicast packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`unsigned int EqtIPv6BroadcastCount() const`

Description: Number of IPv6 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit unsigned integer representing the total number of broadcast packets filtered by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`unsigned int EqtTCPCount() const`

Description: Number of TCP packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the TCP header.

`unsigned int EqtUDPCount() const`

Description: Number of UDP packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the UDP header.

`unsigned int EqtSCTPCount() const`

Description: Number of SCTP packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`unsigned int EqtICMPCount() const`

Description: Number of ICMP packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the ICMP header.

`unsigned int Eqt63OrLessCount() const`

Description: Number of packets on the EQT side that have less than 64 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is less than 64 bytes.

`unsigned int Eqt64To127Count() const`

Description: Number of packets on the EQT side that are between 64 and 127 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 64 and 127 bytes.

```
unsigned int  Eqt128To255Count() const
```

Description: Number of packets on the EQT side that are between 128 and 255 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 128 and 255 bytes.

```
unsigned int  Eqt256To511Count() const
```

Description: Number of packets on the EQT side that are between 256 and 511 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 256 and 511 bytes.

```
unsigned int  Eqt512To1023Count() const
```

Description: Number of packets on the EQT side that are between 512 and 1023 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

```
unsigned int  Eqt1024To1500Count() const
```

Description: Number of packets on the EQT side that are between 1024 and 1500 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 1024 and 1500 bytes.

```
unsigned int  Eqt1501OrMoreCount() const
```

Description: Number of packets on the EQT side that are 1501 or more bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is 1501 or more bytes.

```
unsigned int  EqtMisalignedCount() const
```

Description: Number of packets that are misaligned on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: None

```
unsigned int  NetPacketCount() const
```

Description: Number of packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: None

```
unsigned int  NetIPv4Count() const
```

Description: Number of IPv4 packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

`unsigned int NetIPv4MulticastCount() const`

Description: Number of IPv4 multicast packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of “1110” (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

`unsigned int NetIPv4BroadcastCount() const`

Description: Number of IPv4 broadcast packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the NET side has the IPv4 destination address of 255.255.255.255 (or another addresses if the net mask is set appropriately).

`unsigned int NetIPv6Count() const`

Description: Number of IPv6 packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`unsigned int NetIPv6MulticastCount() const`

Description: Number of IPv6 multicast packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

1. The first byte of the address is 0xFF.
2. The last 2 bytes are not equal to 0x0001.
3. The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`unsigned int NetIPv6BroadcastCount() const`

Description: Number of IPv6 broadcast packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the NET side has an IPv6 destination address that meets all of the following criteria:

1. The first byte of the address is 0xFF.
2. The last 2 bytes are equal to 0x0001.
3. The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`unsigned int NetTCPCount() const`

Description: Number of TCP packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the TCP header.

`unsigned int NetUDPCount() const`

Description: Number of UDP packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the UDP header.

`unsigned int NetSCTPCount() const`

Description: Number of SCTP packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the SCTP header.

`unsigned int NetICMPCount() const`

Description: Number of ICMP packets on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the ICMP header.

`unsigned int Net63OrLessCount() const`

Description: Number of packets on the NET side that have less than 64 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is less than 64 bytes.

```
unsigned int  Net64To127Count() const
```

Description: Number of packets on the NET side that are between than 64 and 127 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 64 and 127 bytes.

```
unsigned int  Net128To255Count() const
```

Description: Number of packets on the NET side that are between than 128 and 255 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 128 and 255 bytes.

```
unsigned int  Net256To511Count() const
```

Description: Number of packets on the NET side that are between than 256 and 511 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 256 and 511 bytes.

```
unsigned int  Net512To1023Count() const
```

Description: Number of packets on the NET side that are between than 512 and 1023 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`unsigned int  Net1024To1500Count() const`

Description: Number of packets on the NET side that are between than 1024 and 1500 bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 1024 and 1500 bytes.

`unsigned int  Net1501OrMoreCount() const`

Description: Number of packets on the NET side that are 1501 or more bytes.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is 1501 or more bytes.

`unsigned int  NetMisalignedCount() const`

Description: Number of packets that are misaligned on the NET side.

Parameters: None

Return Value: 29-bit

Remarks: None

`unsigned int EqtFilterPacketCount(int index) const`

Description: Return the number of filtered packets on EQT side for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 29-bit

Remarks: None

`unsigned long long EqtFilterByteCount(int index) const`

Description: Return the number of filtered bytes for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 36-bit

Remarks: This counter may not be valid if the EqtFilterByteCountInvalid of the same filter slot returns true.

`bool EqtFilterByteCountInvalid(int index) const`

Description: Return whether the EqtFilterByteCount of the same filter slot has a valid value.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: Return true if the EqtFilterByteCount of the same filter slot has a valid value.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`unsigned int NetFilterPacketCount(int index) const`

Description: Number of filtered packets on the NET side for a filter slot.

Parameters: index
Type: int

A number between 0 and 15.

Return Value 29-bit value returned

Remarks: None

`unsigned long long NetFilterByteCount(int index) const`

Description: Number of filtered bytes on the NET side for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 29-bit value returned.

Remarks: None

`bool NetFilterByteCountInvalid(int index) const`

Description: Indicates whether the filtered byte count on the NET side is valid for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 29-bit value returned.

`int MetricsDataLength() const`

Description: Return the size of the MetricsData object

Parameters: None

Return Value: Return the number of bytes needed to store a MetricsResult object

Remarks: In some cases, an application may want to store the entire MetricsResult object away for analysis at a later time. Application can allocate a buffer of size returned by the MetricsDataLength function.
Note: MetricsResult object size is the same for the same MetricsResult object version.

`int  MetricsData(void *buffer, int length) const`

Description: Copies the content of the MetricsResult object into a byte array.

Parameters: buffer:
Type: pointer to a byte array
pointer to a buffer of size “length”.

length:
Type: int
size in bytes of the buffer.

Return Value: Returns the number of bytes copied.

Remarks: If “length” is greater than the number returned by MetricsDataLength, then only “MetricsDataLength” bytes are copied.

If “length” is less than MetricsDataLength, then only “length” bytes are copied. In that case, if you use this buffer to obtain a MetricsResult object using CreateMetricsResult function, the values returned by the MetricsResult’s functions are invalid.

`void MetricsData(PAString& s) const`

Description: Copies the content of the MetricsResult object into the PAString object.

Parameters: s:
Type: a reference to PAString

Return Value: None

Remarks: An application can use CreateMetricsResult function and the value in s to recreate a MetricsResult object.

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: PacketSourceTypeInfo

PacketSourceTypeInfo provides information on a packet source supported by the PacketPortal Library.

Methods

Each of the PacketSourceTypeInfo class methods is described below.

`void Name(PAString& s) const`

Description: Returns the name of the packet source in the PAString.

Parameters: s
Type: PAString&
The name of the packet source will be stored in s.

Return Value: None

Remarks: The name of the packet source can be passed to PacketResultAccess's SetSourceType function. This value is not case-sensitive.

```
void Description(PAString& s) const
```

Description: Returns the description of the packet source.

Parameters: s
Type: PAString&
The description of the packet source will be stored in s.

Return Value: None

```
void HelpText(PAString& s) const
```

Description: Returns more information on packet source properties.

Parameters: s
Type: PAString&
The help text of the packet source will be stored in s.

Return Value: None

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: PacketSourceTypeInfoList

PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling the global function CreatePacketSourceTypeInfoList.

Methods

Each of the PacketSourceTypeInfoList class methods is described below.

`int Size()`

Description: Returns the number of objects in the collection.

Parameters: None

Return Value: Returns the number of objects in the collection.

`PacketSourceTypeInfo *Get(int index)`

Description: Returns a pointer to a PacketSourceTypeInfo object by its position in the collection.

Parameters: index
Type: int
The position of the object to be retrieved. Positions start at 0.

Return Value: If index is valid, then returns a pointer to a PacketSourceTypeInfo object. Otherwise, returns null.

Example: See the example in the CreatePacketSourceTypeInfoList function in Global Functions.

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

Class: PacketResultAccess

The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.

Methods

Each of the PacketResultsAccess class methods is described below.

`bool SetSourceType(const char *sourceType)`

Description: Specifies the packet source.

Parameters: sourceType

Type:const char *

Set the packet source to one of the following: TCP, UDP, Libpcap, and File. This parameter is not case sensitive.

Note: Additional detailed information for each parameter is shown in the sections that immediately follow this section.

Return Value: Returns true if the sourceType parameter specified a supported source. Otherwise returns false. Call LastError for extended information.

Remarks: If this function is called after Start, then the running instance is stopped before the sourceType is applied. All counter information is lost.

Additional Parameter Information (Describing UDP, TCP, File, and Libpcap parameters in detail):

UDP retrieves PacketPortal packets using the UDP protocol. Use the UDP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using UDP. Since UDP provides unreliable data service, there may be packet loss between the PRE and the PacketAccess API application.

Property Name	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("UDP");

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error
```

```
}

if (!pAccess->Start())

{

  // handle error

}

// add another listening port during the run.

if (!pAccess->SetSourceProperty("port", 10002))

{

  // handle error

}

// remove a listening port during the run.

if (!pAccess->SetSourceProperty("removePort", 10001))

{

  // handle error

}
```

TCP retrieves PacketPortal packets using TCP protocol. Use the TCP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using TCP. Since TCP provides a reliable data service, there may be minimal packet loss between the PRE and the PacketAccess API application. However, TCP adds some overhead and may cause more delays in the PacketAccess API.

Property Name:	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default
MaxConnections:	The maximum number of pending TCP connections. This value is passed to the system	Positive integer	No	64

	call listen, and is subject to the limit set by the operating system.			

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("TCP");

// optionally set default socket buffer size for

// listening TCP ports

if (!pAccess->SetSourceProperty("socketBufferSize", 1024 * 64))

{

}

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pAccess->Start())

{

  // handle error

}
```

File retrieves PacketPortal packets from a PCAP capture file. The File source can be used for post processing of captured filter results packets, or used in the emulation mode with a regular PCAP file. TimeBreakpoint is ignored when using the File source. Inter-packet gap is also ignored, as the FilterResultAccess object returns the filter results (or emulated filter results) to the application as fast as it can read and sequence the packets.

Property Name	Description	Type	Allow Multiple?	Defaults
FileName:	Use packets from this PCAP file. Must set this property before Start. If file does not exist or user does not have sufficient permission to read it, then Start returns false. Setting of this property after a Start will be ignored until the next Start.	Pointer to a null-terminated char array	No	None

Loop:	The number of times the file is looped.	Integer. 0 means loop forever	No	1
IdleTime:	Idle this many milliseconds every “idleInterval” number of packets.	Positive integer	No	0
IdleInterval:	Idle the number milliseconds specified by “IdleTime” every this many number of packets.	Positive integer	No	100

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->SetSourceProperty("loop", 2);

if (!pAccess->Start())

{

  // Error situations:

  //    File does not exist;

  //    The application does not have sufficient

  //        permission to open the file;

  //    The file is not a valid PCAP file.

}
```

Libpcap retrieves PacketPortal packets from an Ethernet device in promiscuous mode. When the PacketPortal system is configured to send the filter results packets to the PacketAccess API using UDP, the application can choose to use the “libpcap” mode instead of the TCP mode. One advantage of using the libpcap source instead of the UDP source is that it can limit receiving packets by a device; the libpcap source also allows the application to receive filter results packets from any UDP port.

Property Name	Description	Type	Allow Multiple?	Defaults
Device:	Monitor this device (network interface name). This property may be set before or after Start, and it will take effect immediately if the device is successfully	Pointer to a null-terminated	Yes	None

	opened.	char array		
RemoveDevice:	Stop monitoring this device.	Pointer to a null-terminated char array	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
// find libpcap devices on the host by calling pcap_findalldevs

// this example uses the first device found by the libpcap library

char error[PCAP_ERRBUF_SIZE + 1];

pcap_if_t *alldevs = NULL;

if (pcap_findalldevs(&alldevs, error) == 0 && alldevs != NULL)

{

   FilterResultAccess *pAccess = CreateFilterResultAccess();

   pAccess->SetSourceType("libpcap");

   pAccess->SetSourceProperty("device", t->name);

   if (!pAccess->Start())

   {

     // handle error

     PAString s;

     pAccess->LastError(s);

     printf("Error: %s\n", s.c_str());

   }

}
```

`bool SetSourceProperty (const char *name, const char *value)`

Description: Sets or adds a value to a source property.

Parameters: name
Type: const char *
Specifies the property name.

value
Type: const char *
Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

If the value of the property name is a numeric type, this function will convert the value string to the numeric value automatically.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->SetSourceProperty("loop", "2");

if (!pAccess->Start())

{

  // handle error

  PAString error;

  pAccess->LastError(error);

}
```

void SetSourceProperty (const char *name, int value)

Description: Sets or adds a value to a source property.

Parameters: name
Type: const char *
Specifies the property name.

value

Type: const char *
Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

Example:

```
FilterResultAccess * pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("UDP");

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pAccess->SetSourceProperty("port", 10002))

{

  // handle error

}
```

void GetSourceType(PAString& s)

Description: Returns the currently specified source type

Parameters: s
Type: PAString&
The source type will be saved to the reference to PAString

Return Value: None

Remarks: Returns an empty string if the source is unspecified

void GetSourcePropertyNames(PAStrings& v)

Description: Gets all the valid property names of the packet source associated with the object.

Parameters: v
Type: PAStrings&
All the property names for the packet source associated with the object is stored to a PAString collection.

Return Value: None

Remarks: The object should have a valid packet source before calling GetSourcePropertyNames.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("UDP"))

{

  // handle error

}

PAStrings names;

pAccess->GetSourcePropertyNames(names);

for (size_t i = 0; i < names.size(); i++)

{

  PAString name;

  names.at(i, name);

  printf("property: %s", name.c_str());

}
```

void GetSourceProperty(const char *name, PAString& value)

Description: Gets the first value associated with the property name.

Parameters: name
Type: const char*
Specifies the property name

value
Type: PAString&

Returns the value associated with the specified name in the PAString reference.

Return Value: None

Remarks: If there is no value associated with this property, GetSourceProperty returns an empty string. If there is more than one value associated with this property, then the first value is returned.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("TCP"))

{

  // handle error

}

PAString portValue;

pAccess->GetSourceProperty("port", portValue);

if (portValue.length() == 0)

{

  // no port value set

}
```

`void GetSourceProperties(const char *name, PAStrings& values)`

Description: Gets all the values associated with the property name.

Parameters: name

Type: const char*
Specifies the property name.

values

Type: PAStrings&
Store all the values associated with the property to the reference to the PAString.

Return Value: None

`void Emulate(bool b)`

Description: Turn on or off emulation mode.

Parameters: b

Type: boolean

When b is true, turns on emulation mode, otherwise, turns off emulation mode.

Return Value: None

Remarks: When emulation mode is turned on, every network packet retrieved from the specified source is emulated as one or two filter result packets. The payload of the filter result packet will be the original network packet, and the values in the filter result packet header fields will be set sensibly.

This function must be called prior to Start. Once Start is called, the emulation mode of the running instance cannot be changed until the next Start.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("file"))

{

  // handle error

}

if (!pAccess->SetSourceProperty("fileName", "test.pcap"))

{

  // handle error

}

pAccess->Emulate(true);  // turn on emulation

if (!pAccess->Start())

{

  // handle error

}

FilterResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{
```

```
  // Packets in the test.pcap file is emulated as FRPs.
  // The payload is the original network packet.
  PAString payload;
  pResult->Payload(payload);
  // ...
  DeleteFilterResult(pResult);
}
pAccess->Stop();
DeleteFilterResultAccess(pAccess);
```

`bool Emulate() const`

Description: Returns the current state of emulation

Parameters: None

Return Value: Returns true if emulation is on, otherwise returns false.

`void LastError(PAString& error) const`

Description: Returns the last error.

Parameters: error
Type: PAString&
The error string is returned in the reference to PAString

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();
if (!pAccess->Start())
{
   PAString error;
   pAccess->LastError(error);
```

```
    printf("Error: %s\n", error.c_str());
}
pAccess->ClearError();
```

void ClearError()

Description: Clears the last error.

Parameters: None

Return Value: None

Example: See example in LastError

void BufferSize(long size)

Description: Specifies the maximum number of objects stored in the internal buffer.

Parameters: size
Type: long
Specifies the maximum number of objects.

Return Value: None

Remarks: Application should adjust the buffer size based on memory available for use with the API, how fast the PacketPortal packets are arriving and if there are high latencies among PacketPortal packets routed from multiple SFProbes.

The memory usage is roughly equal to (size * 2000) + (N * 2000) where N = number of actual objects in the buffer.

long BufferSize() const

Description: Returns the current setting of the buffer size.

Parameters: None

Return Value: Returns the maximum number of objects stored in the internal buffer.

C++ (Dynamic Linked Version, Windows only)

Namespace: PacketAccessDLL_NS

Class: FilterResultAccess

The FilterResultAccess class retrieves FilterResult objects.

Methods

Each of the FilterResultAccess class methods is described below.

`bool Start()`

Description: Start processing filter result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a FilterResultAccess object by using CreateFilterResultAccess. After setting the appropriate source properties, the application typically calls Start. All counters are reset to zero at start. The application can then call Get to retrieve available filter results. When the application decides to stop processing filter results, it should call Stop and then DeleteFilterResultAccess to free up resources used by FilterResultAccess.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("udp"))

{

  // handle error

}

if (!pAccess->SetSourceProperty("port", 25000))

{

  // handle error

}

if (!pAccess->Start())
```

```
{
  // handle error
}

const int timeout = 1000; // 1 second

FilterResult *pResult;
while ((pResult = pAccess->Get(timeout)) != NULL)
{
  // handle FilterResult
  // ...
  DeleteFilterResult(pResult);
}
pAccess->Stop();
DeleteFilterResultAccess(pAccess);
```

`bool Stop()`

Description: Stop processing filter result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: No more filter results are available to the application after Stop is called. All counters are still valid until DeleteFilterResultAccess or another Start is called.

Example: See **Start** example provided earlier in this section.

`bool LoadSettings(const char *s)`

Description: Configure the FilterResultAccess object according to the settings string. The setting string can be obtained by calling SaveSettings on an existing FilterResultAccess object. Application should not alter the string returned by SaveSettings.

Parameters: s

Type: const char *
A pointer to a NULL terminated character array containing the configuration settings of a FilterResultAccess object.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: Calling LoadSettings will stop a running instance of FilterResultAccess and all counter information will be lost. The function will not automatically restart the FilterResultAccess object.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->BufferSize(50000);

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->Sequencing(false);

PAString s;

pAccess->SaveSettings(s);


// s can be treated as an opaque NULL-terminated string

// stored with other application settings.

// ...

FilterResultAccess *pAccess2 = CreateFilterResultAccess();

if (!pAccess2->LoadSettings(s.c_str()))

{

  // handle error

}

if (!pAccess2->Start())

{
```

```
    // handle error
  }
```

`void SaveSettings(PAString& s)`

Description: Save the current configuration of the FilterResultAccess object to a string. This function does not affect the current state of the FilterResultAccess object.

Parameters: s
Type: PAString&
The settings will be saved to this object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Return Value: None

Example: See **LoadSettings** example provided earlier in this section.

`void Sequencing(bool b)`

Description: This function sets sequencing on or off. When sequencing is turned on, the FilterResultAccess object will return filter results to the application according to a set of sequencing rules. If sequencing is turned off, filter results are made available to the application immediately on a first-in, first-out basis.

Parameters: b: bool

When b is true, turns on sequencing, otherwise, turns off sequencing.

Remarks: An application using the FilterResultAccess object with sequencing turned on is subject to the following rules for sequencing:

- The timestamp for a FilterResult will be the same or later than the previous FilterResult provided to the application.
- Each FilterResult will be held for a minimum of the specified MinBufferTime before it is available to the application.
- Each FilterResult will be held for a maximum of the specified MaxBufferTime before it is available to the application.
- FilterResults of the same SFProbe are ordered by sequence numbers.
- FilterResults of different SFProbes are ordered by timestamps.
- For FilterResults of the same SFProbe, if a FilterResult with an earlier sequence has a later timestamp than another FilterResult, then the FilterResult with the later sequence will have its timestamp adjusted to be a

later time than the FilterResult with an earlier sequence.

- For FilterResults from different SFProbes and have the same timestamp, a FilterResult that arrived earlier is provided to the application before a FilterResult that arrived later.

Example: See **Start** example provided earlier in this section.

`bool Sequencing() const`

Description: Returns the sequencing setting.

Parameters: None

Return Value: Returns true if sequencing is turned on; otherwise, false is returned.

`void DiscardLate(bool b)`

Description: This sets the application to discard Filter Results that are received late. A filter result is considered "late" if the application has already retrieved a filter result with a more recent timestamp.

Parameters: b:

Type: bool
When b is true, discard late filter results, otherwise, late filter results' timestamp will be adjusted to the timestamp that is most recently provided to the application, and then sequence accordingly.

Return Value: None

Remarks: DiscardLate takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`bool DiscardLate() const`

Description: Returns the DiscardLate setting.

Parameters: None

Return Value: Returns true if the object is set to discard late filter results, otherwise, false is returned.

`void MinBufferTime(long timeout)`

Description: This specifies the time period (the minimum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This allows filter results from multiple SFProbes with different latencies to be time ordered. The MinBufferTime should typically be set to the maximum expected delta in the latency among feeds.

Parameters: timeout
  Type: long
  Specifies in millisecond the minimum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: MinBufferTime takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`long MinBufferTime() const`

Description: Returns the minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

Parameters: None

Return Value: The minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`void MaxBufferTime(long timeout)`

Description: This specifies the time period (the maximum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This is used when there are sequence number gaps in the Filter Results and the application is allowing extra time for the missing Filter Results to arrive.

Parameters: timeout
  Type: long
  Specifies in millisecond the maximum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: If sequencing is turned off, this parameter is ignored for sequencing purposes, but is used to determine when an unmatched truncated or fragmented filter result packet will be discarded from the buffer.

```
long MaxBufferTime() const
```

Description: Returns the maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

Parameters: None

Return Value: The maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

```
void SequenceBreakpoint(long breakpoint)
```

Description: This sets the number of missing sequence numbers before FilterResultAccess treats the filter result as a new feed. A filter result sequence number specifies the order of the original captured packets. If there is a large gap in sequence numbers between two filter results from the same SFProbe, this may indicate that a feed has been stopped and restarted.

Parameters: breakpoint
Type: long
Specifies the number of missing sequence number.

Return Value: None

Remarks: An application should set sequence breakpoint to a large number (e.g. the same as the buffer size) if the FilterResultAccess object is expected to capture a feed that does not stop and restart. Conversely, if the application anticipates that the feed often stops and restarts during a running instance of the FilterResultAccess object, then it should set the sequence breakpoint to a relatively small number. If sequencing is turned off, this parameter is ignored.

```
long SequenceBreakpoint() const
```

Description: Returns the sequence breakpoint value.

Parameters: None

Return Value: The sequence breakpoint value

`void TimeBreakpoint(long millisecond)`

Description: This sets the time period that the application waits to receive the next Filter Result Packet in the sequence. If a new Filter Result Packet has not arrived within the number of milliseconds specified by TimeBreakpoint value, then any Filter Result Packet that arrives after that will be treated as a new feed. This allows feeds to be stopped and restarted, and be sequenced correctly within the same running instance of FilterResultAccess.

Parameters: millisecond
Type: long
Specifies the time breakpoint in milliseconds. If the value is 0, then time breakpoint is not used.

Return Value: None

Remarks: Time breakpoint should be set to shortest expected time delay between stopping a feed and starting a feed. If sequencing is turned off, this parameter is ignored.

`long TimeBreakpoint() const`

Description: Returns the time breakpoint value.

Parameters: None

Return Value: The time breakpoint value

`unsigned int NumProbes()`

Description: This is the number of unique SFProbes that the application has retrieved filter results from.

Parameters: None

Return Value: Number of unique SFProbes that the application has retrieved filter results from

Remarks: This counter does not count filter results that are in the FilterResultAccess buffer, but not yet retrieved by the application. This number is only valid when sequencing is turned on.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

// ... set source type and properties

FilterResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

  // ...

  DeleteFilterResult(pResult);

}

pAccess->Stop();

printf("Number of probes: %lu\n", pAccess->NumProbes());

printf("Number of input filter result packets: %llu\n", pAccess-
>InputFRPCount());

DeleteFilterResultAccess(pAccess)
```

`unsigned long long LostCount()`

Description: This number represents the number of missing filter results, by sequence number, for all SFProbes. This number is only valid when sequencing is turned on.

Parameters: None

Return Value: The number of missing filter results of all SFProbes by sequence number

Remarks: LostCount can be affected by TimeBreakpoint and SequenceBreakpoint values.

For example, filter results from the same probe ID with the following sequence numbers arrive:

Filter Result 1
<5 ms gap>

Filter Result 3
<20 ms gap>
Filter Result 15

Sequence Breakpoint	Time Breakpoint	Lost Count	Description
10	0	1	The filter result with sequence number 2 is considered lost, and the filter result with sequence number 15 is considered the start of a new sequence
20	0	12	The filter result with sequence number 2 and the filter results with sequence numbers 4 through 14 are all considered lost.
20	10	1	The filter result with sequence number 2 is considered lost because filter result 3 arrives less than 10 ms after filter result 1. Filter results with sequence 4 through 14 are not considered lost because filter results 15 arrives more than 10 ms later, even though filter results 15 is less than sequenceBreakpoint away from filter results 3.

Since LostCount counts all the gaps in filter results, it may not reflect the loss of filter results if the loss happens before the first filter results of a particular probe, or if the loss happens after the last filter result was processed by the application.

The following examples illustrate how LostCount and DiscardCount are related.

Scenario 1: Multiple Probes with Late Filter Results

In this scenario, the results from Probe B are all "late" and FilterResultAccess is configured to discard late packets. The following is the filter result packets arrival order (where Probe A results are: A1, A2, A3, and A4 -and- Probe B results are: B1 and B2):

A1, A2, B1, B2, A3, A4

FilterResultAccess will discard B1 and B2 because they are considered late, therefore, the application receives four filter results: A1 - A4.

In this case, LostCount is 0 since there are no gaps in sequence numbers for probe A
and DiscardCount is 2.

Scenario 2: Filter Results discarded towards the end of a run

In this scenario, the following packets arrived from probe A: A1, A2, A3, A5, A6, A7, A8, A9, A10.

Assume that A8, A9 and A10 are discarded by FilterResultAccess because of buffer overflow.

In this case, after the application receives A7, the LostCount is 1 (because A4 is missing), and the DiscardCount is 3 (because A8, A9 and A10 are discarded).

`unsigned long long DiscardCount()`

Description: This is the number of filter results discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The number of filter results discarded

Remarks: This is valid whether sequencing is turned on or not.

`unsigned long long DiscardDuplicateCount()`

Description: This is the number of filter results discarded because the filter result is considered a duplicate.

Parameters: None

Return Value: The number of filter results discarded because they are duplicates

Remarks: This is a relatively rare occasion, and usually occurs when the Filter Result Packets are duplicated by the network due to incorrect network configuration.

`unsigned long long DiscardLateCount()`

Description: This is the number of filter results discarded because the filter result is considered to be late.

Parameters: None

Return Value: The number of filter results discarded because they are considered to be late.

Remarks: There are several reasons that a filter result can be considered late. For example:

- The SFProbes are not time synchronized with the PRE.
- Multiple PREs are not time synchronized with one another.
- The MinBufferTime value is not set high enough to accommodate the difference in network latencies among the Filter Result Packets.

If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

`unsigned long long DiscardOutOfSequenceCount()`

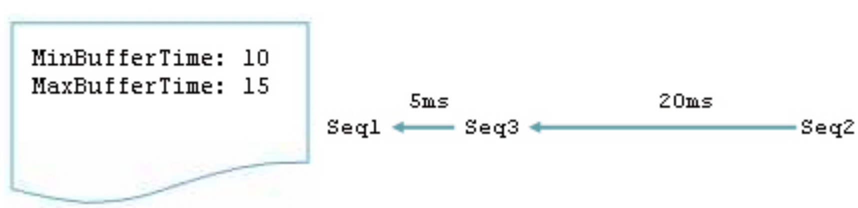
Description: This is the number of filter results discarded because the filter result is considered out of sequence. A filter result is considered out of sequence if the application is provided with a filter result of the same probe ID and a later sequence number.

Parameters: None

Return Value: The number of filter results discarded because the filter result is considered out of sequence

Remarks: This count can be affected by MinBufferTime and MaxBufferTime.

For example, filter results from the same probe ID arrive with the following sequence numbers and time gaps:



T is the time when the filter result with sequence number 1(Seq1) arrives.

T + 0 Seq 1 arrives

T + 5 Seq 3 arrives

T + 10 Seq 1 has been held for MinBufferTime, so it can be provided to application

T + 15 Seq 3 has been held for MinBufferTime, but it will wait 5 more ms to MaxBufferTime because a sequence is missing

T + 20 Seq 3 has been held MaxBufferTime, so it will be provided to application

T + 25 Seq 2 arrives

T + 35 Seq 2 is ready for the application, but it is discarded because it has an earlier sequence number than Seq 3

`unsigned long long DiscardOverflowCount()`

Description: This is the number of filter results discarded because the FilterResultAccess buffer is too full. Filter results are not inserted in the buffer once the number of filter results in the buffer reaches the BufferSize value.

Parameters: None

Return Value: The number of filter results discarded because the FilterResultAccess buffer is too full

Remarks: Changing the BufferSize value can affect the number of filter results that are discarded.

`unsigned long long InputFRPCount()`

Description: This is the number of filter result packets retrieved from the FilterResultAccess source. Only packets that appear to contain a legitimate FilterResults header are counted.

Parameters: None

Return Value: The number of filter result packets retrieved from the FilterResultAccess source

Remarks: This count is valid whether sequencing is turned on or not.

`unsigned long long DiscardFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The total number of Filter Result Packets discarded

Remarks: This count is valid whether sequencing is turned on or not.

`unsigned long long DiscardDuplicateFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates

`unsigned long long DiscardFragmentedFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Remarks: An unmatched filter result packets is discarded once it is kept in the buffer for a period set in MaxBufferTime. Changing the MaxBufferTime value can affect the number of fragmented Filter Result Packets that are discarded.

`unsigned long long DiscardLateFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be late.

Parameters: None

Return Value: The number of filter result packets discarded because the filter result packets are considered to be late

Remarks: If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

`unsigned long long DiscardOutOfSequenceFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be out of sequence.

Parameters: None

Return Value: The number of filter result packets discarded because they are considered to be out of sequence

Remarks: See DisardOutOfSequenceCount.

`unsigned long long DiscardOverflowFRPCount()`

Description: This is the number of filter result packets discarded because the FilterResultAccess buffer is too full.

Parameters: None

Return Value: The number of filter result packets discarded because the FilterResultAccess buffer is too full

Remarks: See DisardOverflowCount.

`FilterResult *Get()`

Description: This retrieves the next filter result from the FilterResultAccess buffer that is available at this moment. If no filter result is currently available, then a NULL is returned.

Parameters: None

Return Value: Returns a pointer to a FilterResult object or NULL

Remarks: When the pointer to the FilterResult object is no longer needed, call DeleteFilterResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains filter result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`FilterResult *Get(long timeout)`

Description: This retrieves the next filter result from the FilterResultAccess buffer, waiting the specified time for an available filter result. If no filter result is available at the end of the specified time, then a NULL is returned.

Parameters: timeout

Type: long

Specifies the maximum time in millisecond before this function returns. If timeout is 0, then this function behaves the same as Get().

Return Value: Returns a pointer to a FilterResult object or NULL

Remarks: When the pointer to the FilterResult object is no longer needed, call DeleteFilterResult to release resources used by the object.

`unsigned int NumResultsInBuffer()`

Description: Retrieves the number of filter results in the buffer.

Parameters: None.

Return Value: Number of filter results in the buffer.

Remarks: Allows an application to detect how full the PA-API internal buffer is.

C++ (Dynamic Linked Version, Windows only)

Namespace: PacketAccessDLL_NS

Class: MetricsResultAccess

The MetricResultsAccess class retrieves MetricsResult objects.

Methods

Each of the MetricsResultAccess class methods is described below.

`bool Start()`

Description: Start processing metrics result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a MetricsResultAccess object by using CreateMetricsResultAccess. After setting the appropriate source properties, the application typically calls Start. The application can then call Get to retrieve available metrics results. When the application decides to stop processing metrics results, it should call Stop and then DeleteMetricsResultAccess to free up resources used by MetricsResultAccess.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

if (!pAccess->SetSourceType("udp"))

{

  // handle error

}

if (!pAccess->SetSourceProperty("port", 25000))

{

  // handle error

}

if (!pAccess->Start())

{

  // handle error
```

```
}

const int timeout = 1000; // 1 second

MetricsResult *pResult;
while ((pResult = pAccess->Get(timeout)) != NULL)
{
  // handle MetricsResult
  // ...
  DeleteMetricsResult(pResult);
}
pAccess->Stop();
DeleteMetricsResultAccess(pAccess);
```

`bool Stop()`

Description: Stop processing metrics result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: No more metrics results are available to the application after Stop is called.

Example: See **Start** example provided earlier in this section.

`bool LoadSettings(const char *s)`

Description: Configure the MetricsResultAccess object according to the settings string. The setting string can be obtained by calling SaveSettings on an existing MetricsResultAccess object. Application should not alter the string returned by SaveSettings.

Parameters: s
Type: const char *

A pointer to a NULL terminated character array containing the configuration settings of a MetricsResultAccess object.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: Calling LoadSettings will stop a running instance of MetricsResultAccess and all counter information will be lost. The function will not automatically restart the MetricsResultAccess object.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

pAccess->BufferSize(50000);

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->Sequencing(false);

PAString s;

pAccess->SaveSettings(s);


// s can be treated as an opaque NULL-terminated string

// stored with other application settings.

// ...

MetricsResultAccess *pAccess2 = CreateMetricsResultAccess();

if (!pAccess2->LoadSettings(s.c_str()))

{

  // handle error

}

if (!pAccess2->Start())

{

  // handle error

}
```

`void SaveSettings(PAString& s)`

Description: Save the current configuration of the MetricsResultAccess object to a string. This function does not affect the current state of the MetricsResultAccess object.

Parameters: s
Type: PAString&
The settings will be saved to this object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Return Value: None

Example: See **LoadSettings** example provided earlier in this section.

`MetricsResult *Get()`

Description: This retrieves the next metrics result from the MetricsResultAccess buffer that is available at this moment. If no metrics result is currently available, then a NULL is returned.

Parameters: None

Return Value: Returns a pointer to a MetricsResult object or NULL

Remarks: When the pointer to the MetricsResult object is no longer needed, call DeleteMetricsResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains metrics result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`MetricsResult *Get(long timeout)`

Description: This retrieves the next metrics result from the MetricsResultAccess buffer, waiting the specified time for an available metrics result. If no metrics result is available at the end of the specified time, then a NULL is returned.

Parameters: timeout
Type: long
Specifies the maximum time in millisecond before this function returns. If timeout is 0, then this function behaves the same as Get().

Return Value: Returns a pointer to a MetricsResult object or NULL

Remarks: When the pointer to the MetricsResult object is no longer needed, call DeleteMetricsResult to release resources used by the object.

Probe Grouping (C++ Dynamic Linked Version only)

The filter results from multiple SFProbes can be mapped to a single probe ID. This is useful in situations where a group of SFProbes are part of a link aggregation group (LAG), where you want Filter Results from multiple SFProbes to appear as a single stream of Filter Results. This feature is only available when using the C++ Dynamic Linked Version of the SDK.

If time sequencing is turned on, then the FRPs for the same group of SFProbes will be re-sequenced to represent the same sequence number ordering. For example, if SFProbe A and SFProbe B are mapped to SFProbe ID C, and the filter results prior to being mapped are as follows:

4. SFProbe A, sequence 10
5. SFProbe B, sequence 200
6. SFProbe A, sequence 11
7. SFProbe A, sequence 13
8. SFProbe B, sequence 201

Then the mapped result will be:

1. SFProbe C, sequence 1
2. SFProbe C, sequence 2
3. SFProbe C sequence 3
4. SFProbe C, sequence 5
5. SFPRobe C, sequence 6

To use probe grouping, you must set a system environment variable called PP_FILTERRESULT_PROBEMAP. It is important that you set this as a system environment variable, not a user environment variable. The value of this environment variable will be a file path such as “C:\ProbeGrouping.xml” pointing to an XML file which will be used to specify the probe grouping. If this environment variable is changed while the SDK is in use, the change will not take effect until the next time the application using the SDK is started. Up to 10 groupings of 16 probes each are supported.

The XML file specifies mapping from a probe ID to another probe ID. The destination probe ID can be an existing probe ID or a new ID.

The following XML example maps four probes to one probe ID:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<probes>
  <probe from="87348472CD57" to="6F421F792126" />
  <probe from="6F421F792126" to="6F421F792126" />
  <probe from="E7576F74F36B" to="6F421F792126" />
  <probe from="9709542DD71A" to="6F421F792126" />
</probes>
```

The following XML example maps four probes to two different probe IDs:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<probes>
  <probe from="87348472CD57" to="000000000001" />
  <probe from="6F421F792126" to="000000000001" />
  <probe from="E7576F74F36B" to="000000000002" />
  <probe from="9709542DD71A" to="000000000002" />
</probes>
```

C++ (Dynamic Linked Version , Windows only)

Namespace: PacketAccessDLL_NS

PacketAccess C++ (Static Version) API

C++ (Static Version) API Library

This section describes the API Library for the C++ Static version.

Header Files

The following header files provide access to the C++ Static Version of the PacketAccess API classes and functions.

Header File	Library
PacketAccess.h	packetAccess.lib (Windows) libpacketAccess.a (Linux)

Overview

The following are available in the static version of the C++ API library. This version can be used with both the Linux and the Microsoft Windows operating systems. Each of these classes is described briefly in the table below and in detail later in this section.

The API library uses the C++ Standard Template Library (STL).

Global Functions	Global Functions allow you to access version information, create and

	delete PacketAccess objects, etc.
FilterResult	The FilterResult class represents an original captured packet and its meta-data. A pointer to this object is obtained through the FilterResultAccess class.
MetricsResult	The MetricsResult class represents information contained in a metrics packet. A pointer to this object is obtained through the MetricsResultAccess class or created from previously obtained metric data.
PacketSourceTypeInfo	PacketSourceTypeInfo provides information on a packet source supported by the PacketAccess Library.
PacketSourceTypeInfoList	PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling the global function CreatePacketSourceTypeInfoList.
PacketResultAccess	The PacketResultAccess class provides the base implementation for accessing packets generated by the PacketPortal system.
FilterResultAccess	The FilterResultAccess class retrieves FilterResult objects.
MetricsResultAccess	The MetricsResultAccess class retrieves MetricsResult objects.

C++ Static Version

Namespace: PacketAccessNS

Global Functions

Global Functions allow you to access version information, create and delete PacketPortal objects, etc.

```
int GetMajorVersion()
```

Description: This function returns the major version number.

Parameters: None

Return Value: Returns the major version number

```
int GetMinorVersion()
```

Description: This function returns the minor version number.

Parameters: None

Return Value: Returns the minor version number

`int GetPatchVersion()`

Description: This function returns the patch version number.

Parameters: None

Return Value: Returns the patch version number

`int GetBuildVersion()`

Description: This function returns the build version number.

Parameters: None

Return Value: Returns the build version number

`unsigned int GetBuildTime()`

Description: This function returns the build time.

Parameters: None

Return Value: Returns the build time in number of seconds since January 01, 1970, 00:00:00.

`std::string GetVersion()`

Description: This function returns a version string representing the version information.

Parameters: None

Return Value: Returns a string representing the version.

Example:

```
std::string ver = GetVersion();

printf("PacketPortal Library Version: %s\n", ver.c_str());
```

`FilterResultAccess *CreateFilterResultAccess()`

Description: This function creates a FilterResultAccess object.

Parameters: None

Return Value: Returns a pointer to the FilterResultAccess object

Remarks: Call DeleteFilterResultAccess to release resources used by the FilterResultAccess object.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

// …

DeleteFilterResultAccess(pAccess);
```

`MetricsResultAccess *CreateMetricsResultAccess()`

Description: This function creates a MetricsResultAccess object.

Parameters: None

Return Value: Returns a pointer to the MetricsResultAccess object

Remarks: Call DeleteMetricsResultAccess to release resources used by the MetricsResultAccess object.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

// …

DeleteMetricsResultAccess(pAccess);
```

`void DeleteFilterResultAccess(FilterResultAccess *p)`

Description: This function deletes a FilterResultAccess object.

Parameters: p
Type: FilterResultAccess *
A pointer to a valid FilterResultAccess object.

Return Value: None

Remarks: A FilterResultAccess object is created by using CreateFilterResultAccess.

Example: See the example in CreateFilterResultAccess

`void DeleteMetricsResultAccess(MetricsResultAccess *p)`

Description: This function deletes a MetricsResultAccess object.

Parameters: p
Type: MetricsResultAccess *
A pointer to a valid MetricsResultAccess object.

Return Value: None

Remarks: A MetricsResultAccess object is created by using CreateMetricsResultAccess.

Example: See the example in CreateMetricsResultAccess

`FilterResult *CreateFilterResult(const void *header, int headerSize)`

Description: Create a FilterResult object from a previous stored header.

Parameters: header

Type: const void *

A pointer to the header buffer.

headerSize

Type: int

Number of bytes in the header buffer.

Return Value: A FitlerResult object

Remarks: The buffer used to create the FilterResult can be obtained by calling the Header() or HeaderPointer() function of a previously retrieved FilterResult.

This allows a FilterResult to be stored away and recreated for later processing. This version of the CreateFitlerResult() does not recreate the payload portion of the FilterResult.

The following information is lost for this re-created FilterResult:

* IsLate() - will always return false

* IsNewSequence() - will always return false

* Seconds() - will always be the same as ProbeSeconds()

* NSeconds() - will always be the same as ProbeNSeconds()

The reason the above information is lost for the recreated FilterResult is because the above functions is affected by the state of FitlerResultAccess (if sequencing is turned on), and not part of the data stored in the FilterResult header.

```
FilterResult *CreateFilterResult(const void *header, int headerSize, const
void *payload, int payloadSize)
```

Description: Create a FilterResult object from a previous stored header.

Parameters: header

Type: const void *

A pointer to the header buffer.

headerSize

Type: int

Number of bytes in the header buffer.

payload

Type: const void *

A pointer to the payload buffer.

payloadSize

Type: int

Number of bytes in the payload buffer.

Return Value: A FitlerResult object.

Remarks: See remarks for the other version of CreateFilterResult. This version of CreateFilterResult also restored the payload portion of the filter result. The payload buffer pointer can be obtained by calling the Payload() function of a previously retrieved FilterResult

void DeleteFilterResult(FilterResult *p)

Description: This function deletes a FilterResult object.

Parameters: p
Type: FilterResult *
A pointer to a valid FilterResult object.

Return Value: None

Remarks: A pointer to a FilterResult object is returned by calling FilterResultAccess object's Get functions, when filter result becomes available from the FilterResultAccess object.
Call DeleteFilterResult to release resources used by the FilterResult object.

MetricsResult *CreateMetricsResult(const void * buffer, int bufferSize)

Description: This function creates a MetricsResult object from data stored in the buffer.

Parameters: buffer
Type: const void * buffer
A pointer to metrics data. Metrics data can be returned by calling the MetricsData function from an existing MetricsResult instance. These two functions allow an application to store and retrieve the metrics data as a byte array.

Return Value: Returns a pointer to a MetricsResult.

Remarks: This function always returns a MetricsResult pointer, even if the buffer may point to invalid data. You may receive invalid data when you call functions in MetricsResult if the buffer contains invalid data or of insufficient size.

An application should release the memory used by this MetricsResult by calling DeleteMetricsResult() after it is no longer needed.

void DeleteMetricsResult(MetricsResult *p)

Description: This function deletes a MetricsResult object.

Parameters: p
Type: MetricsResult *
A pointer to a valid MetricsResult object.

Return Value: None

Remarks: A pointer to a MetricsResult object is returned by calling MetricsResultAccess object's Get functions, when Metrics result becomes available from the MetricsResultAccess object.
Call DeleteMetricsResult to release resources used by the MetricsResult object.

`PacketSourceTypeInfoList *CreatePacketSourceTypeInfoList()`

Description: This function creates a PacketSourceTypeInfoList object.

Parameters: None

Return Value: Returns a pointer to PacketSourceTypeInfoList object

Remarks: Call DeletePacketSourceTypeInfoList to release resources used by the object

Example:

```
PacketSourceTypeInfoList *pList =
    CreatePacketSourceTypeInfoList();


GetPacketSourceTypeInfo(pList);
for (size_t i = 0; i < pList->Size(); i++)
{
   PacketSourceTypeInfo *pInfo = pList->Get(i);
   std::string type = pInfo->Name();
   std::string desc = pInfo->Description();
   std::string help = pInfo->HelpText();
   printf("%s (%s): %s\n",
           type.c_str(), desc.c_str(), help.c_str());
}
DeletePacketSourceTypeInfoList(pList);
```

`void GetPacketSourceTypeInfo(PacketSourceTypeInfoList *pList)`

Description: This function fills in the PacketSourceTypeInfoList object with all the source types supported by the PacketAccess Library.

Parameters: pList
Type: GetPacketSourceTypeInfo *
The packet source type information is stored in pList.

Return Value: None

Remarks: The application can use this function to dynamically find out all the packet source types supported by the PacketAccess library. The name of the source type can be passed to the PacketResultAccess object's SetSourceType function.

Example: See example in CreatePacketSourceTypeInfoList

`void DeletePacketSourceTypeInfoList(PacketSourceTypeInfoList *pList)`

Description: This function deletes the PacketSourceTypeInfoList object.

Parameters: pList
Type: PacketSourceTypeInfoList *
A pointer to the PacketSourceTypeInfoList.

Return Value: None

Remarks: A pointer to a FilterResult object is returned by calling CreatePacketSourceTypeInfoList.

Example: See example in CreatePacketSourceTypeInfoList

C++ Static Version

Namespace: PacketAccessNS

Class: FilterResult

The FilterResult class represents an original captured packet and its meta-data. A pointer to this object is obtained through the FilterResultAccess class.

Methods

Each of the FilterResult class methods is described below.

`int Version() const`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Library version.

`std::string ProbeId() const`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: None

Return Value: Returns the probe ID.

Remarks: A probe ID is not null-terminated. Applications should use the returned string's length function to return the length of the probe ID string.

`unsigned int Seconds() const`

Description: Returns the “Seconds” portion of the timestamp. This value may or may not be the same as the “ProbeSeconds” value depending on sequencing rules. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: If sequencing is turned on, a FilterResult's timestamp may be adjusted. See the Sequencing section in the Understanding Filter Results chapter for more information.

`unsigned int NSeconds() const`

Description: Returns the "nanoseconds" portion of the timestamp. This value may or may not be the same as the "ProbeNSeconds" value depending on sequencing rules.

Parameters: None

Return Value: Returns the "nanoseconds" portion of the timestamp.

`unsigned int Sequence() const`

Description: Returns an unsigned 32-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a FilterResult is missing.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

//... setup source and source properties

bool bNewSequence = true;

unsigned int lastSequence = 0;

FilterResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

  // for this example, let's assume that all

  // FilterResults comes from the same probe.

  if (bNewSequence)

  {

    lastSequence = pResult->Sequence();
```

```
      bNewSequence = false;
    }
    else
    {
      unsigned int currentSequence = pResult->Sequence();
      if (lastSequence + 1 != currentSequence)
      {
         // one or more FilterResult is missing …
         // the above expression works with
         // sequence number wrapping since it is
         // an unsigned value.
      }
      lastSequence = currentSequence;
    }
    // ...
    DeleteFilterResult(pResult);
  }
  pAccess->Stop();
  DeleteFilterResultAccess(pAccess);
```

unsigned int FilterMatchBits() const

Description: Returns a value that represents which filters are matched

Parameters: None

Return Value: A value that represents which filters are matched.

unsigned int CongestionCount() const

Description: The number of packets that has matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow.

Parameters: None

Return Value: Returns a 29-bit value representing packets that matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow. This counter only applies to the side for this filtered packet.

Remarks: Only 29-bits of this value are valid. There are two congestion counters, one for equipment side and one for network side. When the SFProbe is unable to process a packet due to buffer overflow, it increments this counter for the side of this filtered packet.

Since the sequence number is not incremented in this situation, the application may receive filter results with consecutive sequence numbers, when in fact there are missing filtered packets. When the packets that the SFProbe are unable to process are on the same side as the next successfully injected filter result packets, application can check the CongestionCount for potential packet loss.

`unsigned int InjectedCount() const`

Description: Returns the number of captured packets that the SFProbe has successfully injected.

Parameters: None

Return Value: The number of captured packets that the SFProbe has successfully injected.

`bool IsBadFCS() const`

Description: Returns whether the original captured packet has a bad FCS.

Parameters: None

Return Value: Returns true if the original captured packet has a bad FCS.

`bool IsHeaderOnly() const`

Description: Returns whether the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Remarks: If the original capture packet matches more than 1 filter, and not all of them has the "headers only" setting, then the captured payload may contain more than the protocol headers.

`bool IsInjectNet() const`

Description: Returns whether the filter result was injected on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the filter result was injected on the network side of the SFProbe, otherwise returns false.

Remarks: This flag is not related to whether the original captured packet is on the network or equipment side of the SFProbe.

`bool IsLate() const`

Description: Returns whether the filter result is considered late.

Parameters: None

Return Value: Returns true if the filter result is considered late. Otherwise returns false.

Remarks: A filter result is considered late if the application has already retrieved a filter result with a more recent timestamp. The application can optionally discard these filter results by calling FilterResultAccess object's DiscardLate(false) function.

`bool IsNet() const`

Description: Returns whether the original captured packet is on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the original captured packet was captured on the network side of the SFProbe. Otherwise returns false, indicating the original captured packet was captured on the equipment side.

`bool IsNewSequence() const`

Description: Returns whether the filter result indicates a new sequence.

Parameters: None

Return Value: Returns true if the filter result indicates a new sequence. Otherwise returns false.

Remarks: A filter result is considered a new sequence depending on sequencing rules. A filter result can be considered a new sequence if it is the first filter result from a particular SFProbe; a filter result that has a sequence number that is sufficiently far away from the previous filter result's sequence number from the same SFProbe; or if this filter result arrives a long time after other filter results. The SequenceBreakpoint and Timebreakpoint functions of FilterResultAccess can be used to adjust the breakpoint values.

`bool IsOnlyRoute() const`

Description: Returns whether this machine and port is the only recipient of this FilterResult.

Parameters: None

Return Value: Returns true if this machine and port is the only recipient of this FilterResult, otherwise returns false.

Remarks: If this machine and port is the only recipient of this FilterResult, then a missing sequence in the filter result indicates that a filter result is unable to reach the application. If multiple machine and ports can be the intended recipients of this FilterResult, then a missing sequence number only indicates that a filter result may be missing.

`bool IsSliced() const`

Description: Returns whether the filter expression requested the SFProbe to slice the payload of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to slice the payload of the original captured packet. If slicing was not been requested, a false is returned.

Remarks: This value indicates the setting of the filter used to capture the original captured packet. The captured payload may not necessarily be truncated. If the original packet is too big, the captured payload may be truncated even if the filter does not specify truncation.

To determine if the Filter Result object is sliced, you can compare the "real packet length" and "payload length" of the Filter Result object. The payload is sliced if either of the two following conditions are true:

- the real packet length is zero
- the payload length is less than the real packet length

The "real packet length" and "payload length" are methods documented later in the FilterResult class.

`bool IsTimingLock() const`

Description: Returns whether the filter result is captured when the SFProbe is time synchronized with the PRE.

Parameters: None

Return Value: Returns true if the filter result is captured when the SFProbe is time synchronized with the PRE. Otherwise returns false.

Remarks: If the SFProbe is not time synchronized with the PRE and original captured packets are expected to be captured by multiple SFProbes, then the timestamps may not reflect the true packet order. In that case, the application may consider either turning off sequencing, or setup time synchronization for the SFProbes involved.

`bool WasFragmented() const`

Description: Returns whether the filter result was assembled from two filter result packets.

Parameters: None

Return Value: Returns true if the filter result was assembled from two filter result packets.

Otherwise returns false.

Remarks: If the captured payload is over a specified limit (usually around the MTU of the network), then two filter result packets are needed to carry the metadata and the original captured packet as payload. This function is useful if the application wants to identify this situation.

`int RealPacketLength() const`

Description: Returns the original packet length in bytes, if known. This length does not include the 4 byte FCS of the original packet.

Parameters: None

Return Value: Returns the original packet length in number of bytes.

Remarks: The maximum number of bytes counted by the probe depends on its configuration and network encapsulation. Typically, the maximum number is around the maximum MTU size, or up to around 2000 bytes. When the actual number of bytes in the original packet is not known, the function returns 0.
The application can determine if the payload returned for the filter result is sliced by comparing the return value of PayloadLength function and RealPacketLength function. The payload for the filter result is sliced if either of the following conditions exist:

- If RealPacketLength returns zero
- If PayloadLength is less than RealPacketLength

`int PayloadLength() const`

Description: Returns the length of the payload captured.

Parameters: None

Return Value: Returns number of bytes of the payload captured.

`std::string Payload() const`

Description: Returns the payload.

Parameters: None

Return Value: Returns the captured packet.

Remarks: The captured payload maybe truncated by the probe depending on probe configuration. The payload does not include the 4 byte FCS.

`unsigned int ProbeSeconds() const`

Description: Returns the “seconds” portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The “seconds” portion of the real time that the original packet is captured by the SFProbe.

`unsigned int ProbeNSeconds() const`

Description: Returns the “nanoseconds” portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The “nanoseconds” portion of the real time that the original packet is captured by the SFProbe.

`int HeaderLength() const`

Description: Returns the length of the header.

Parameters: None

Return Value: Number of bytes in the FilterResult header.

`int Header(void *buffer, int bufferSize) const`

Description: Copies the FilterResult header to the buffer.

Parameters:

buffer

Type: void *

A pointer to a buffer for the header.

bufferSize

Type: int

Size of the buffer in bytes.

Return Value: Return value: Number of bytes copied to the buffer.

Remarks: If the buffer size is too small for the whole Filter Result header, only the bufferSize number of bytes are copied.

`std::string Header() const`

Description: Returns a string containing the FilterResult header.

Parameters: None

Return Value: None

`const char   *HeaderPointer() const`

Description: Returns a pointer to the first byte of the FilterResult header.

Parameters: None

Return Value: None

Remarks: The pointer will become invalid after the FilterResult is freed (by calling DeleteFilterResult).

C++ Static Version

Class: MetricsResult

The MetricsResult class represents metrics result packets generated by the SFProbe. A pointer to this object is obtained through the MetricsResultAccess class, or the global function CreateMetricsResult.

Example

```
// Use Packet Access API to read metrics packets from a PCAP file

int FromFile(const std::string& s)

{

    PacketAccessNS::MetricsResultAccess *pMetrics =
PacketAccessNS::CreateMetricsResultAccess();

    if (!pMetrics->SetSourceType("file"))

    {

        std::string error = pMetrics->LastError();

        printf("Error create metrics access source: %s\n", error.c_str());

        PacketAccessNS::DeleteMetricsResultAccess(pMetrics);

        return -1;

    }


    pMetrics->SetSourceProperty("filename", s);

    pMetrics->Emulate(true); // turn on emulation


    if (!pMetrics->Start())

    {

        std::string error = pMetrics->LastError();

        printf("Error starting metrics access: %s\n", error.c_str());

        PacketAccessNS::DeleteMetricsResultAccess(pMetrics);

        return -1;

    }
```

```
    PacketAccessNS::MetricsResult *pResult;

    const long timeout = 100;

    while ((pResult = pMetrics->Get(timeout)) != NULL)

    {

        PrintResult(pResult);

        PacketAccessNS::DeleteMetricsResult(pResult);

    }


    PacketAccessNS::DeleteMetricsResultAccess(pMetrics);

    return 0;

}


void PrintResult(PacketAccessNS::MetricsResult *pResult)

{

    std::string probeId = pResult->ProbeId();


    printf("  Probe: %s %u.%u\n",

        BinaryToText(probeId).c_str(),

        pResult->Seconds(),

        pResult->NSeconds());

        printf("  Version                  : %u\n", pResult->Version());

        printf("  Sequence                 : %u\n", pResult->Sequence());

        printf("  ResetCount               : %u\n", pResult->ResetCount());

        printf("  RetryCount               : %u\n", pResult->RetryCount());

        printf("  SFFTemperature           : %u\n", pResult-
>SFFTemperature());

        printf("  SFFVcc                   : %u\n", pResult->SFFVcc());

        printf("  SFFTxBias                : %u\n", pResult->SFFTxBias());
```

```
        printf("  SFFTxPower              : %u\n", pResult->SFFTxPower());

        printf("  SFFRxPower              : %u\n", pResult->SFFRxPower());

        printf("  TimingOffset            : %d\n", pResult->TimingOffset());

        printf("  M2SAverageNSecond       : %d\n", pResult-
>M2SAverageNSecond());

        printf("  S2MAverageNSecond       : %d\n", pResult-
>S2MAverageNSecond());

        printf("  IsTimingValid           : %s\n", pResult-
>IsTimingValid()?"true":"false");

        printf("  IsTimingLock            : %s\n", pResult-
>IsTimingLock()?"true":"false");

        printf("  EqtByteCount            : %u\n", pResult->EqtByteCount());

        printf("  NetByteCount            : %u\n", pResult->NetByteCount());

        printf("  EqtPacketsFiltered      : %u\n", pResult-
>EqtPacketsFiltered());

        printf("  EqtPacketsInjected      : %u\n", pResult-
>EqtPacketsInjected());

        printf("  NetPacketsFiltered      : %u\n", pResult-
>NetPacketsFiltered());

        printf("  NetPacketsInjected      : %u\n", pResult-
>NetPacketsInjected());

        printf("  EqtPacketCount          : %u\n", pResult-
>EqtPacketCount());

        printf("  EqtIPv4Count            : %u\n", pResult->EqtIPv4Count());

        printf("  EqtIPv4MulticastCount   : %u\n", pResult-
>EqtIPv4MulticastCount());

        printf("  EqtIPv4BroadcastCount   : %u\n", pResult-
>EqtIPv4BroadcastCount());

        printf("  EqtIPv6Count            : %u\n", pResult->EqtIPv6Count());

        printf("  EqtIPv6MulticastCount   : %u\n", pResult-
>EqtIPv6MulticastCount());

        printf("  EqtIPv6BroadcastCount   : %u\n", pResult-
>EqtIPv6BroadcastCount());

        printf("  EqtTCPCount             : %u\n", pResult->EqtTCPCount());
```

```
        printf("  EqtUDPCount             : %u\n", pResult->EqtUDPCount());

        printf("  EqtSCTPCount            : %u\n", pResult->EqtSCTPCount());

        printf("  EqtICMPCount            : %u\n", pResult->EqtICMPCount());

        printf("  Eqt63OrLessCount        : %u\n", pResult-
>Eqt63OrLessCount());

        printf("  Eqt64To127Count         : %u\n", pResult-
>Eqt64To127Count());

        printf("  Eqt128To255Count        : %u\n", pResult-
>Eqt128To255Count());

        printf("  Eqt256To511Count        : %u\n", pResult-
>Eqt256To511Count());

        printf("  Eqt512To1023Count       : %u\n", pResult-
>Eqt512To1023Count());

        printf("  Eqt1024To1500Count      : %u\n", pResult-
>Eqt1024To1500Count());

        printf("  Eqt1501OrMoreCount      : %u\n", pResult-
>Eqt1501OrMoreCount());

        printf("  EqtMisalignedCount      : %u\n", pResult-
>EqtMisalignedCount());

        printf("  NetPacketCount          : %u\n", pResult-
>NetPacketCount());

        printf("  NetIPv4Count            : %u\n", pResult->NetIPv4Count());

        printf("  NetIPv4MulticastCount   : %u\n", pResult-
>NetIPv4MulticastCount());

        printf("  NetIPv4BroadcastCount   : %u\n", pResult-
>NetIPv4BroadcastCount());

        printf("  NetIPv6Count            : %u\n", pResult->NetIPv6Count());

        printf("  NetIPv6MulticastCount   : %u\n", pResult-
>NetIPv6MulticastCount());

        printf("  NetIPv6BroadcastCount   : %u\n", pResult-
>NetIPv6BroadcastCount());

        printf("  NetTCPCount             : %u\n", pResult->NetTCPCount());

        printf("  NetUDPCount             : %u\n", pResult->NetUDPCount());

        printf("  NetSCTPCount            : %u\n", pResult->NetSCTPCount());
```

```
        printf("  NetICMPCount            : %u\n", pResult->NetICMPCount());

        printf("  Net63OrLessCount        : %u\n", pResult-
>Net63OrLessCount());

        printf("  Net64To127Count         : %u\n", pResult-
>Net64To127Count());

        printf("  Net128To255Count        : %u\n", pResult-
>Net128To255Count());

        printf("  Net256To511Count        : %u\n", pResult-
>Net256To511Count());

        printf("  Net512To1023Count       : %u\n", pResult-
>Net512To1023Count());

        printf("  Net1024To1500Count      : %u\n", pResult-
>Net1024To1500Count());

        printf("  Net1501OrMoreCount      : %u\n", pResult-
>Net1501OrMoreCount());

        printf("  NetMisalignedCount      : %u\n", pResult-
>NetMisalignedCount());

        printf("  EqtFilterPacketCount    : %u\n", pResult-
>EqtFilterPacketCount(0));

        printf("  EqtFilterByteCount      : %u\n", pResult-
>EqtFilterByteCount(0));

        printf("  EqtFilterByteCountInvalid: %u\n", pResult-
>EqtFilterByteCountInvalid(0));

        printf("  NetFilterPacketCount    : %u\n", pResult-
>NetFilterPacketCount(0));

        printf("  NetFilterByteCount      : %u\n", pResult-
>NetFilterByteCount(0));

        printf("  NetFilterByteCountInvalid: %s\n", pResult-
>NetFilterByteCountInvalid(0)?"true":"false");

        printf("  PRETimeSync             : %s\n", (!pResult-
>IsPRETimeSync() && pResult->PRETimeSyncLossCount() == 0) ? "Off" : "On");

        printf("  PRETimeSyncLostCount    : %u\n", pResult-
>PRETimeSyncLossCount());

        printf("\n\n");

}
```

```
std::string BinaryToText(const std::string& s)
{
    // xx:xx:xx:xx:xx:xx format
    std::string h;

    char buf[12];
    memset(buf, 0, sizeof(buf));

    for (size_t i = 0; i < s.size(); i++)
    {
        unsigned int c = (unsigned int) s[i] & 0xFF;
        sprintf(buf, "%02x", c);
        h += std::string(buf, 2);

        if (i < s.size() -1)
            h += ":";
    }
    return h;
}
```

Methods

Each of the MetricsResult class methods is described below.

`bool IsPRETimeSync() const`

Description:	Indicates whether the PRE is time synced with the wall clock.
Parameters:	None
Return Value:	Returns true if the PRE is time sync with the wall clock.
Remarks:	The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off.

When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

`unsigned int PRETimeSyncLossCount() const`

Description: Indicates the number of times the PRE has lost time sync with the wall clock.

Parameters: None

Return Value: Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

Remarks: If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

`int Version() const`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the metrics results packet format version associated with this object. It is not related to the PacketAccess Library version.

`void std:string ProbeId() const`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: None

Return Value: None

Remarks: A probe ID is not null-terminated. Applications should use PAString's length function to return the length of the probe ID string.

`unsigned int Seconds() const`

Description: Returns the “Seconds” portion of the timestamp. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: MetricsResults are returned to the application in a first-in, first-out manner. The application may receive a MetricsResult with an earlier timestamp than the previous MetricsResult it receives. This is very common if there are multiple probes in different parts of the network sending MetricsResults to the same MetricsResultsAccess object, or if the probes are not time synchronized with the PRE.

`unsigned int NSeconds() const`

Description: Returns the “nanoseconds” portion of the timestamp.

Parameters: None

Return Value: Returns the “nanoseconds” portion of the timestamp.

`unsigned int Sequence() const`

Description: Returns an unsigned 16-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a MetricsResult is lost. A MetricsResult can be lost in transit, or due to buffer overflow.

A gap in the sequence number indicates that an intended MetricsResult for that probe was not delivered. In most cases, an application can safely ignore this situation.

There are two situations an application may want to re-baseline its counters if there is a skipped MetricsResult:

1. If an application is interested in the filter byte counters. The filter byte counter may become invalid if a jumbo packet (more than about 2000

bytes) has been filtered. In that case, the filter byte counter invalid flag for that filter slot will be set. This flag is cleared for each Metrics Result request. If there is a lost Metrics Result, the application may not be aware that the filter byte counter invalid flag has been reset.

2. Under rare situations, if there are too many missed sequence numbers, then the counters may rollover more than once. Different counters rollover at different rates, depending on the counter's capacity and network traffic volume,. The application should decide when the number of missed sequences may cause a double rollover.

   For example, the theoretical maximum number of Ethernet frames per second on a 1G network is around 1.4 million frames per second, and the 29-bit total packet count can count up to around 536 million packets. So packet counter may rollover around every 6 minutes. If the metrics result request interval (configurable through System Manager) is every 5 minutes, then missing two consecutive Metrics Results may cause a double rollover.

Example:

```
MetricsResultAccess *pAccess =
CreateMetricsResultAccess();

//... setup source and source properties

bool bNewSequence = true;

unsigned short lastSequence = 0;

MetricsResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

  // for this example, let's assume that all

  // MetricsResults comes from the same probe.

  if (bNewSequence)

  {

    lastSequence = (unsigned short) pResult->Sequence();

    bNewSequence = false;

  }

  else

  {
```

```
    unsigned short currentSequence = pResult->Sequence();

    if ((unsigned short)(lastSequence + 1) !=
currentSequence)

    {

       // one or more MetricsResult is missing …

       // the above expression works with

       // sequence number wrapping since it is

       // an unsigned value.

    }

    lastSequence = currentSequence;

  }

  // ...

  DeleteMetricsResult(pResult);

}

pAccess->Stop();

DeleteMetricsResultAccess(pAccess);
```

`unsigned int ResetCount() const`

Description: Returns a value that represents how many times the application should treat this metrics result as a new baseline.

Parameters: None

Return Value: None

Example:

```
MetricsResultAccess *pAccess =
CreateMetricsResultAccess();

// …

const long timeout = 1000; // one second
```

```
bool isFirst = true;

unsigned short lastResetCount = 0;

unsigned short lastSequence = 0;

std::string reason;


MetricsResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

    bool shouldbaseline = false;

    if (isFirst)

    {

        shouldbaseline = true;

        isFirst = false;

        reason = "first metrics result";

    }

    else if (lastResetCount != (unsigned short) pResult-
>ResetCount())

    {

       shouldbaseline = true;

        reason = "new reset count, the metrics feed may
be stopped and restarted.";

    }

    else if ((lastSequence + 1)&0xffff != (unsigned
short) pResult->Sequence())

    {

        // skipping one sequence number may not affect
the counters you are interested in.


       shouldbaseline = true;

        reason = "sequence number is skipped, some
```

```
counters may have double rollover.";

    }



    // find out if there are any invalid bits in the
filter byte counter

    for (int i = 0; i < 8; i++)

    {

        if (pResult->NetFilterByteCountInvalid(i) ||
pResult->EqtFilterByteCountInvalid(i))

        {

           shouldbaseline = true;

            reason = "Invalid filter byte counter";

        }

    }



    lastResetCount = (unsigned short) pResult-
>ResetCount();

    lastSequence = (unsigned short) pResult->Sequence();


    if (shouldbaseline)

    {

        // log the situation and reset counters that may
affect your measurement

    }


    // ... more processing

}
```

unsigned int RetryCount() const

Description: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Parameters: None

Return Value: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Remarks: This value may be important to applications that want to determine if the missing sequence number is due to the PRE unable to transmit or receive metrics results to and from the SFProbe.

```
int SFFTemperature() const
```

Description: Temperature of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit signed integer in increments of 1/256 °C

```
unsigned int SFFVcc() const
```

Description: Supply voltage of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit integer.

Remarks: 16 bit unsigned integer in increments of 100 μV.

```
unsigned int SFFTxBias() const
```

Description: Laser bias current of the SFProbe.

Parameters: None

Return Value: 16 bit unsigned integer.

Remarks: Returns a 16 bit unsigned integer in increments of 2 μV.

`unsigned int SFFTxPower() const`

Description: Transmitted average optical power of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit unsigned integer.

Remarks: 16 bit unsigned integer in increments of 0.1 μW.

`int  M2SAverageNSecond() const`

Description: The average time needed for a packet to travel from PRE to SFProbe.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the PRE to SFProbe.

`int S2MAverageNSecond() const`

Description: The average time in nanoseconds needed for a packet to travel from SFProbe to PRE.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the SFProbe to PRE.

`int TimingOffset() const`

Description: M2SAverageNSecond minus the average of M2SAverageNSecond and S2MAverageNSecond. M2S - (M2S + S2M) / 2

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average round trip latency between the PRE to the SFProbe.

`bool IsTimingValid() const`

Description: Indicates whether the IsTimingLock return value is valid.

Parameters: None

Return Value: Boolean value indicating "true" if the time IsTimingLock is valid.

Remarks: This value should be used in conjunction with the IsTimingLock.

`bool IsTimingLock() const`

Description: Indicates whether the SFProbe is in time synchronization with the PRE at the time this packet is generated.

Parameters: None

Return Value: Boolean value indicating "true" if the time is synchronized between the PRE and the API.

Remarks: None

`unsigned long long EqtByteCount() const`

Description: Total number of bytes on the EQT side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the EQT side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`unsigned long long NetByteCount() const`

Description: Total number of bytes on the NET side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the NET side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`unsigned int  EqtPacketsFiltered() const`

Description: Total number of packets filtered on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered on the EQT side.

Remarks: None

`unsigned int  EqtPacketsInjected() const`

Description: Total number of packets injected by the SFProbe on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

`unsigned int  NetPacketsFiltered() const`

Description: Total number of packets filtered by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered by the SFProbe on the NET side.

Remarks: None

```
unsigned int  NetPacketsInjected() const;
```

Description: Total number of packets injected by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the NET side.

Remarks: None

```
unsigned int  EqtPacketCount() const
```

Description: Number of packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

Remarks: None

```
unsigned int  EqtIPv4Count() const;
```

Description: Number of IPv4 packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv4 packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

```
unsigned int  EqtIPv4MulticastCount() const
```

Description: Number of IPv4 multicast packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of multicast packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of “1110” (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

```
unsigned int  EqtIPv4BroadcastCount() const
```

Description: Number of IPv4 broadcast packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of broadcast packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has the IPv4 destination address of 255.255.255.255.

```
unsigned int  EqtIPv6Count() const
```

Description: Number of IPv6 packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv6 packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

```
unsigned int  EqtIPv6MulticastCount() const
```

Description: Number of IPv6 multicast packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of multicast packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`unsigned int EqtIPv6BroadcastCount() const`

Description: Number of IPv6 broadcast packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of broadcast packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`unsigned int EqtTCPCount() const`

Description: Number of TCP packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of TCP packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the TCP header.

`unsigned int  EqtUDPCount() const`

Description: Number of UDP packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of UDP packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the UDP header.

`unsigned int  EqtSCTPCount() const`

Description: Number of SCTP packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of SCTP packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`unsigned int  EqtICMPCount() const`

Description: Number of ICMP packets on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of ICMP packets injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`unsigned int  Eqt63OrLessCount() const`

Description: Number of packets on the EQT side that have less than 64 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets with less than 64 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is less than 64 bytes.

`unsigned int Eqt64To127Count() const`

Description: Number of packets on the EQT side that are between 64 and 127 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 64 and 127 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 64 and 127 bytes.

`unsigned int Eqt128To255Count() const`

Description: Number of packets on the EQT side that are between 128 and 255 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 128 and 255 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 128 and 255 bytes.

`unsigned int Eqt256To511Count() const`

Description: Number of packets on the EQT side that are between 256 and 511 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 256 and 511 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 256 and 511 bytes.

`unsigned int Eqt512To1023Count() const`

Description: Number of packets on the EQT side that are between 512 and 1023 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 512 and 1023 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`unsigned int  Eqt1024To1500Count() const`

Description: Number of packets on the EQT side that are between 1024 and 1500 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 1024 and 1500 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 1024 and 1500 bytes.

`unsigned int  Eqt1501OrMoreCount() const`

Description: Number of packets on the EQT side that are 1501 or more bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets over 1501 bytes injected by the SFProbe on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is 1501 or more bytes.

`unsigned int  EqtMisalignedCount() const`

Description: Number of packets that are misaligned on the EQT side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets that have been misaligned on the EQT side.

Remarks: None

`unsigned int  NetPacketCount() const`

Description: Number of packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets on the NET side.

Remarks: None

```
unsigned int  NetIPv4Count() const
```

Description: Number of IPv4 packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv4 packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

```
unsigned int  NetIPv4MulticastCount() const
```

Description: Number of IPv4 multicast packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv4 multicast packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of "1110" (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

```
unsigned int  NetIPv4BroadcastCount() const
```

Description: Number of IPv4 broadcast packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv4 broadcast packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet on the NET side has the IPv4 destination address of 255.255.255.255.

`unsigned int NetIPv6Count() const`

Description: Number of IPv6 packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv6 packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`unsigned int NetIPv6MulticastCount() const`

Description: Number of IPv6 multicast packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv6 multicast packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`unsigned int NetIPv6BroadcastCount() const`

Description: Number of IPv6 broadcast packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of IPv6 broadcast packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet on the NET side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`unsigned int NetTCPCount() const`

Description: Number of TCP packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of TCP packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the TCP header.

`unsigned int NetUDPCount() const`

Description: Number of UDP packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of UDP packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the UDP header.

`unsigned int NetSCTPCount() const`

Description: Number of SCTP packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of SCTP packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the SCTP header.

unsigned int NetICMPCount() const

Description: Number of ICMP packets on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of ICMP packets injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side contains the ICMP header.

unsigned int Net63OrLessCount() const

Description: Number of packets on the NET side that have less than 64 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets with less than 64 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is less than 64 bytes.

unsigned int Net64To127Count() const

Description: Number of packets on the NET side that are between than 64 and 127 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 64 and 127 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 64 and 127 bytes.

unsigned int Net128To255Count() const

Description: Number of packets on the NET side that are between than 128 and 255 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 128 and 255 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 128 and 255 bytes.

`unsigned int Net256To511Count() const`

Description: Number of packets on the NET side that are between than 256 and 511 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 256 and 511 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 256 and 511 bytes.

`unsigned int Net512To1023Count() const`

Description: Number of packets on the NET side that are between than 512 and 1023 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 512 and 1023 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`unsigned int Net1024To1500Count() const`

Description: Number of packets on the NET side that are between than 1024 and 1500 bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets between 1024 and 1500 bytes injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is between 1024 and 1500 bytes.

`unsigned int  Net1501OrMoreCount() const`

Description: Number of packets on the NET side that are 1501 or more bytes.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets that are 1501 bytes or more injected by the SFProbe on the NET side.

Remarks: This counter is incremented by the SFProbe if a packet header on the NET side is 1501 or more bytes.

`unsigned int  NetMisalignedCount() const`

Description: Number of packets that are misaligned on the NET side.

Parameters: None

Return Value: A 29-bit unsigned integer representing the total number of packets that have been misaligned on the EQT side.

Remarks: None

`unsigned int EqtFilterPacketCount(int index) const`

Description: Return the number of filtered packet for filter slot indicated by “index”.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: A 29-bit integer value of the indexed slot on the EQT side.

Remarks: None

`unsigned long long EqtFilterByteCount(int index) const`

Description: Return the number of filtered bytes for filter slot indicated by “index”.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: A 36-bit integer value of the indexed slot on the EQT side.

Remarks: This counter may not be valid if the EqtFilterByteCountInvalid of the same filter slot returns true.

`bool EqtFilterByteCountInvalid(int index) const`

Description: Return whether the EqtFilterByteCount of the same filter slot has a valid value.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: Return true if the EqtFilterByteCount of the same filter slot has a valid value.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`unsigned int NetFilterPacketCount(int index) const`

Description: Number of filtered packets on the NET side for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: A 29-bit integer value of the indexed slot on the NET side.

Remarks: None

`unsigned long long NetFilterByteCount(int index) const`

Description: Number of filtered bytes on the NET side for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: A 36-bit integer value of the indexed slot on the NET side.

Remarks: None

`bool NetFilterByteCountInvalid(int index) const`

Description: Indicates whether the filtered byte count on the NET side is valid for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: Return true if the NetFilterByteCount of the same filter slot has a valid value.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`int MetricsDataLength() const`

Description: Returns the size of the MetricsData object

Parameters: None

Return Value: Return the number of bytes needed to store a MetricsResult object.

Remarks: In some cases, an application may want to store the entire MetricsResult object away for analysis at a later time. Application can allocate a buffer of size returned by the MetricsDataLength function.
Note: MetricsResult object size is the same for the same MetricsResult object version.

`int MetricsData(void *buffer, int length)` `const`

Description: Copies the content of the MetricsResult object into a byte array.

Parameters: buffer:
Type: pointer to a byte array
pointer to a buffer of size “length”.

length:
Type: int
size in bytes of the buffer.

Return Value: Returns the number of bytes copied.

Remarks: If “length” is greater than the number returned by MetricsDataLength, then only “MetricsDataLength” bytes are copied.

If “length” is less than MetricsDataLength, then only “length” bytes are copied. In that case, if you use this buffer to obtain a MetricsResult object using CreateMetricsResult function, the values returned by the MetricsResult’s functions are invalid.

`std::string MetricsData() const`

Description: Copies the content of the MetricsResult object into the returned string.

Parameters: None

Return Value: The copied string is returned.

Remarks: An application can use CreateMetricsResult function and the value in s to recreate a MetricsResult object.

C++ Static Version

Namespace: PacketAccessNS

Class: PacketSourceTypeInfo

PacketSourceTypeInfo provides information on a packet source supported by the PacketPortal Library.

Methods

Each of the PacketSourceTypeInfo class methods is described below.

`std::string Name() const`

Description: Returns the name of the packet source.

Parameters: None

Return Value: Returns the name of the packet source.

Remarks: The name of the packet source can be passed to PacketResultAccess's SetSourceType function. This value is not case-sensitive.

`std::string Description() const`

Description: Returns the description of the packet source.

Parameters: None

Return Value: Returns the description of the packet source.

`std::string HelpText() const`

Description: Returns more information on packet source properties.

Parameters: None

Return Value: Returns more information on packet source properties.

C++ Static Version

Namespace: PacketAccessNS

Class: PacketSourceTypeInfoList

PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling the global function CreatePacketSourceTypeInfoList.

Methods

Each of the PacketSourceTypeInfoList class methods is described below.

`int Size()`

Description: Returns the number of objects in the collection.

Parameters: None

Return Value: Returns the number of objects in the collection.

`PacketSourceTypeInfo *Get(int index)`

Description: Returns a pointer to a PacketSourceTypeInfo object by its position in the collection.

Parameters: index
Type: int
The position of the object to be retrieved. Positions start at 0.

Return Value: If index is valid, then returns a pointer to a PacketSourceTypeInfo object. Otherwise, returns null.

Example: See the example in the CreatePacketSourceTypeInfoList function in Global Functions.

C++ Static Version

Namespace: PacketAccessNS

Class: PacketResultAccess

The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.

Methods

Each of the PacketResultsAccess class methods is described below.

`bool SetSourceType(std::string sourceType)`

Description: Specifies the packet source.

Parameters: sourceType
Type: std::string
Set the packet source to one of the following: TCP, UDP, Libpcap, and File. This parameter is not case sensitive.

Note: Additional detailed information for each parameter is shown in the sections that immediately follow this section.

Return Value: Returns true if the sourceType parameter specified a supported source. Otherwise returns false. Call LastError for extended information.

Remarks: If this function is called after Start, then the running instance is stopped before the sourceType is applied. All counter information is lost.

Additional Parameter Information (Describing UDP, TCP, File, and Libpcap parameters in detail):

UDP retrieves PacketPortal packets using the UDP protocol. Use the UDP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using UDP. Since UDP provides unreliable data service, there may be packet loss between the PRE and the PacketAccess API application.

Property Name	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect	Positive integer 1 - 65535	Yes	None

	immediately if the port is successfully opened.			
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("UDP");

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pAccess->Start())

{

  // handle error

}

// add another listening port during the run.

if (!pAccess->SetSourceProperty("port", 10002))

{

  // handle error

}

// remove a listening port during the run.

if (!pAccess->SetSourceProperty("removePort", 10001))

{

  // handle error

}
```

TCP retrieves PacketPortal packets using TCP protocol. Use the TCP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using TCP. Since TCP provides a reliable data service, there may be minimal packet loss between the PRE and the PacketAccess API application. However, TCP adds some overhead and may cause more delays in the PacketAccess API.

Property Name:	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default
MaxConnections:	The maximum number of pending TCP connections. This value is passed to the system call listen, and is subject to the limit set by the operating system.	Positive integer	No	64

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("TCP");

// optionally set default socket buffer size for listening TCP ports

if (!pAccess->SetSourceProperty("socketBufferSize", 1024 * 64))

{

}

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pAccess->Start())

{

  // handle error
```

```
}
```

File retrieves PacketPortal packets from a PCAP capture file. The File source can be used for post processing of captured filter results packets, or used in the emulation mode with a regular PCAP file. TimeBreakpoint is ignored when using the File source. Inter-packet gap is also ignored, as the FilterResultAccess object returns the filter results (or emulated filter results) to the application as fast as it can read and sequence the packets.

Property Name	Description	Type	Allow Multiple?	Defaults
FileName:	Use packets from this PCAP file. Must set this property before Start. If file does not exist or user does not have sufficient permission to read it, then Start returns false. Setting of this property after a Start will be ignored until the next Start.	Pointer to a NULL-terminated char array	No	None
Loop:	The number of times the file is looped.	Integer. 0 means loop forever	No	1
IdleTime:	Idle this many milliseconds every “idleInterval” number of packets.	Positive integer	No	0
IdleInterval:	Idle the number milliseconds specified by “IdleTime” every this many number of packets.	Positive integer	No	100

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->SetSourceProperty("loop", 2);

if (!pAccess->Start())

{

  // Error situations:

  //    File does not exist;

  //    The application does not have sufficient

  //        permission to open the file;
```

```
  //    The file is not a valid PCAP file.

}
```

Libpcap retrieves PacketPortal packets from an Ethernet device in promiscuous mode. When the PacketPortal system is configured to send the filter results packets to the PacketAccess API using UDP, the application can choose to use the “libpcap” mode instead of the TCP mode. One advantage of using the libpcap source instead of the UDP source is that it can limit receiving packets by a device; the libpcap source also allows the application to receive filter results packets from any UDP port.

Property Name	Description	Type	Allow Multiple?	Defaults
Device:	Monitor this device (network interface name). This property may be set before or after Start, and it will take effect immediately if the device is successfully opened.	Pointer to a NULL-terminated char array	Yes	None
RemoveDevice:	Stop monitoring this device.	Pointer to a NULL-terminated char array	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
// find libpcap devices on the host by calling pcap_findalldevs

// this example uses the first device found by the libpcap library

char error[PCAP_ERRBUF_SIZE + 1];

pcap_if_t *alldevs = NULL;

if (pcap_findalldevs(&alldevs, error) == 0 && alldevs != NULL)

{

   FilterResultAccess *pAccess = CreateFilterResultAccess();

   pAccess->SetSourceType("libpcap");

   pAccess->SetSourceProperty("device", t->name);
```

```
    if (!pAccess->Start())
    {
      std::string s = pAccess->LastError();
      printf("Error: %s\n", s.c_str());
    }
}
```

bool SetSourceProperty (std::string name, std::string value)

Description: Sets or adds a value to a source property.

Parameters: name
Type: std::string
Specifies the property name.

value
Type: std::string
Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

If the value of the property name is a numeric type, this function will convert the value string to the numeric value automatically.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();
pAccess->SetSourceType("file");
pAccess->SetSourceProperty("fileName", "test.pcap");
pAccess->SetSourceProperty("loop", "2");
if (!pAccess->Start())
{
  // handle error
```

```
  std::string error = pAccess->LastError();
}
```

bool SetSourceProperty (std::string name, int value)

Description: Sets or adds a value to a source property.

Parameters: name
Type: std::string
Specifies the property name.

value
Type: int
Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

Example:

```
FilterResultAccess * pAccess = CreateFilterResultAccess();

pAccess->SetSourceType("UDP");

if (!pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pAccess->SetSourceProperty("port", 10002))

{

  // handle error

}
```

std::string GetSourceType()

Description: Returns the currently specified source type

Parameters: None

Return Value: Returns the currently specified source type.

Remarks: Returns an empty string if the source is unspecified

void GetSourcePropertyNames(std::vector<std::string>& v)

Description: Gets all the valid property names of the packet source associated with the object.

Parameters: v

Type: std::vector<std::string>&

All the property names for the packet source associated with the object is stored to a string vector.

Return Value: None

Remarks: The object should have a valid packet source before calling GetSourcePropertyNames.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("UDP"))

{

  // handle error

}

std::vector<std::string> names;

pAccess->GetSourcePropertyNames(names);

for (size_t i = 0; i < names.size(); i++)

{

  printf("property: %s", names[i].c_str());

}
```

std::string GetSourceProperty(std::string name)

Description: Gets the first value associated with the property name.

Parameters: name
Type: std::string
Specifies the property name

Return Value: Returns the value associated with the property name.

Remarks: If there is no value associated with this property, GetSourceProperty returns an empty string. If there is more than one value associated with this property, then the first value is returned.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("TCP"))

{

  // handle error

}

std::string portValue = pAccess->GetSourceProperty("port");

if (portValue.length() == 0)

{

  // no port value set

}
```

void GetSourceProperties(std::string name, std::vector<std::string>& values)

Description: Gets all the values associated with the property name.

Parameters: name
Type: std::string
Specifies the property name.

values
Type: std::vector<std::string>&
Store all the values associated with the property to the reference to the string vector.

Return Value: None

`void Emulate(bool b)`

Description: Turns emulation mode on or off.

Parameters: b
Type: bool
When b is true, turns on emulation mode, otherwise, turns off emulation mode.

Return Value: None

`bool Emulate() const`

Description: Returns the current state of emulation

Parameters: None

Return Value: Returns true if emulation is on, otherwise returns false.

`std::string LastError() const`

Description: Returns the last error.

Parameters: None

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->Start())

{

   std::string error = pAccess->LastError();

   printf("Error: %s\n", error.c_str());

}

pAccess->ClearError();
```

`void ClearError()`

Description: Clears the last error.

Parameters: None

Return Value: None

Example: See example in LastError

`void BufferSize(long size)`

Description: Specifies the maximum number of objects stored in the internal buffer.

Parameters: size
Type: long
Specifies the maximum number of objects.

Return Value: None

Remarks: Application should adjust the buffer size based on memory available for use with the API, how fast the PacketPortal packets are arriving and if there are high latencies among PacketPortal packets routed from multiple SFProbes.

The memory usage is roughly equal to (size * 2000) + (N * 2000) where N = number of actual objects in the buffer.

`long BufferSize() const`

Description: Returns the current setting of the buffer size.

Parameters: None

Return Value: Returns the maximum number of objects stored in the internal buffer.

C++ Static Version

Namespace: PacketAccessNS

Class: FilterResultAccess

The FilterResultAccess class retrieves FilterResult objects.

Methods

Each of the FilterResultAccess class methods is described below.

bool Start()

Description: Start processing filter result packets.

Parameters: None

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a FilterResultAccess object by using CreateFilterResultAccess. After setting the appropriate source properties, the application typically calls Start. All counters are reset to zero at start. The application can then call Get to retrieve available filter results. When the application decides to stop processing filter results, it should call Stop and then DeleteFilterResultAccess to free up resources used by FilterResultAccess.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

if (!pAccess->SetSourceType("udp"))

{

// handle error

}

if (!pAccess->SetSourceProperty("port", 25000))

{

// handle error

}

if (!pAccess->Start())

{

// handle error

}
```

```
FilterResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

// handle FilterResult

// ...

DeleteFilterResult(pResult);

}

pAccess->Stop();

DeleteFilterResultAccess(pAccess);
```

`bool Stop()`

Description: Stop processing filter result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: No more filter results are available to the application after Stop is called. All counters are still valid until DeleteFilterResultAccess or another Start is called.

Example: See **Start** example provided earlier in this section.

`bool LoadSettings(std::string s)`

Description: Configure the object according to the settings string.

Parameters: s

Type: std::string
A string containing the configuration settings of a FilterResultAccess object.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

pAccess->BufferSize(50000);

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->Sequencing(false);

std::string s = pAccess->SaveSettings();

// s can be treated as an opaque string

// stored with other application settings.

// ...

FilterResultAccess *pAccess2 = CreateFilterResultAccess();

if (!pAccess2->LoadSettings(s))

{

  // handle error

}

if (!pAccess2->Start())

{

  // handle error

}
```

`std::string SaveSettings()`

Description: Save the current configuration of the object to a string.

Parameters: None

Return Value: Return a string representing the current configuration of the object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Example: See **LoadSettings** example provided earlier in this section.

`void Sequencing(bool b)`

Description: This function sets sequencing on or off. When sequencing is turned on, the FilterResultAccess object will return filter results to the application according to a set of sequencing rules. If sequencing is turned off, filter results are made available to the application immediately on a first-in, first-out basis.

Parameters: b: bool

When b is true, turns on sequencing, otherwise, turns off sequencing.

Remarks: An application using the FilterResultAccess object with sequencing turned on is subject to the following rules for sequencing:

- The timestamp for a FilterResult will be the same or later than the previous FilterResult provided to the application.
- Each FilterResult will be held for a minimum of the specified MinBufferTime before it is available to the application.
- Each FilterResult will be held for a maximum of the specified MaxBufferTime before it is available to the application.
- FilterResults of the same SFProbe are ordered by sequence numbers.
- FilterResults of different SFProbes are ordered by timestamps.
- For FilterResults of the same SFProbe, if a FilterResult with an earlier sequence has a later timestamp than another FilterResult, then the FilterResult with the later sequence will have its timestamp adjusted to be a later time than the FilterResult with an earlier sequence.
- For FilterResults from different SFProbes and have the same timestamp, a FilterResult that arrived earlier is provided to the application before a FilterResult that arrived later.

Example: See **Start** example provided earlier in this section.

`bool Sequencing() const`

Description: Returns the sequencing setting.

Parameters: None

Return Value: Returns true if sequencing is turned on; otherwise, false is returned.

`void DiscardLate(bool b)`

Description: This sets the application to discard Filter Results that are received late. A filter result is considered “late” if the application has already retrieved a filter result with a more recent timestamp.

Parameters: b:
Type: bool
When b is true, discard late filter results, otherwise, late filter results' timestamp will be adjusted to the timestamp that is most recently provided to the application, and then sequence accordingly.

Return Value: None

Remarks: DiscardLate takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`bool DiscardLate() const`

Description: Returns the DiscardLate setting.

Parameters: None

Return Value: Returns true if the object is set to discard late filter results, otherwise, false is returned.

`void MinBufferTime(long timeout)`

Description: This specifies the time period (the minimum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This allows filter results from multiple SFProbes with different latencies to be time ordered. The MinBufferTime should typically be set to the maximum expected delta in the latency among feeds.

Parameters: timeout
Type: long
Specifies in millisecond the minimum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: MinBufferTime takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`long MinBufferTime() const`

Description: Returns the minimum number of milliseconds that a filter result is kept in the

FilterResultAccess buffer.

Parameters: None

Return Value: The minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`void MaxBufferTime(long timeout)`

Description: This specifies the time period (the maximum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This is used when there are sequence number gaps in the Filter Results and the application is allowing extra time for the missing Filter Results to arrive.

Parameters: timeout
Type: long
Specifies in millisecond the maximum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: If sequencing is turned off, this parameter is ignored for sequencing purposes, but is used to determine when an unmatched truncated or fragmented filter result packet will be discarded from the buffer.

`long MaxBufferTime() const`

Description: Returns the maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

Parameters: None

Return Value: The maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`void SequenceBreakpoint(long breakpoint)`

Description: This sets the number of missing sequence numbers before FilterResultAccess treats the filter result as a new feed. A filter result sequence number specifies the order of the original captured packets. If there is a large gap in sequence numbers between two filter results from the same SFProbe, this may indicate that a feed has been stopped and restarted.

Parameters: breakpoint
Type: long
Specifies the number of missing sequence number.

Return Value: None

Remarks: An application should set sequence breakpoint to a large number (e.g. the same as the buffer size) if the FilterResultAccess object is expected to capture a feed that does not stop and restart. Conversely, if the application anticipates that the feed often stops and restarts during a running instance of the FilterResultAccess object, then it should set the sequence breakpoint to a relatively small number. If sequencing is turned off, this parameter is ignored.

`long SequenceBreakpoint() const`

Description: Returns the sequence breakpoint value.

Parameters: None

Return Value: The sequence breakpoint value

`void TimeBreakpoint(long millisecond)`

Description: This sets the time period that the application waits to receive the next Filter Result Packet in the sequence. If a new Filter Result Packet has not arrived within the number of milliseconds specified by TimeBreakpoint value, then any Filter Result Packet that arrives after that will be treated as a new feed. This allows feeds to be stopped and restarted, and be sequenced correctly within the same running instance of FilterResultAccess.

Parameters: millisecond
Type: long
Specifies the time breakpoint in milliseconds. If the value is 0, then time breakpoint is not used.

Return Value: None

Remarks: Time breakpoint should be set to shortest expected time delay between stopping a feed and starting a feed. If sequencing is turned off, this parameter is ignored.

`long TimeBreakpoint() const`

Description: Returns the time breakpoint value.

Parameters: None

Return Value: The time breakpoint value

`unsigned int NumProbes()`

Description: This is the number of unique SFProbes that the application has retrieved filter results from.

Parameters: None

Return Value: Number of unique SFProbes that the application has retrieved filter results from

Remarks: This counter does not count filter results that are in the FilterResultAccess buffer, but not yet retrieved by the application. This number is only valid when sequencing is turned on.

Example:

```
FilterResultAccess *pAccess = CreateFilterResultAccess();

// ... set source type and properties

FilterResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

  // ...

  DeleteFilterResult(pResult);

}

pAccess->Stop();

printf("Number of probes: %lu\n", pAccess->NumProbes());

printf("Number of input filter result packets: %llu\n",

        pAccess->InputFRPCount());
```

```
DeleteFilterResultAccess(pAccess);
```

unsigned long long LostCount()

Description: This number represents the number of missing filter results, by sequence number, for all SFProbes. This number is only valid when sequencing is turned on.

Parameters: None

Return Value: The number of missing filter results of all SFProbes by sequence number

Remarks: LostCount can be affected by TimeBreakpoint and SequenceBreakpoint values.

For example, filter results from the same probe ID with the following sequence numbers arrive:

Filter Result 1
<5 ms gap>
Filter Result 3
<20 ms gap>
Filter Result 15

Sequence Breakpoint	Time Breakpoint	Lost Count	Description
10	0	1	The filter result with sequence number 2 is considered lost, and the filter result with sequence number 15 is considered the start of a new sequence
20	0	12	The filter result with sequence number 2 and the filter results with sequence numbers 4 through 14 are all considered lost.
20	10	1	The filter result with sequence number 2 is considered lost because filter result 3 arrives less than 10 ms after filter result 1. Filter results with sequence 4 through 14 are not considered lost because filter results 15 arrives

more than 10 ms later, even though filter results 15 is less than sequenceBreakpoint away from filter results 3.

Since LostCount counts all the gaps in filter results, it may not reflect the loss of filter results if the loss happens before the first filter results of a particular probe, or if the loss happens after the last filter result was processed by the application.

The following examples illustrate how LostCount and DiscardCount are related.

Scenario 1: Multiple Probes with Late Filter Results

In this scenario, the results from Probe B are all "late" and FilterResultAccess is configured to discard late packets. The following is the filter result packets arrival order (where Probe A results are: A1, A2, A3, and A4 -and- Probe B results are: B1 and B2):

A1, A2, B1, B2, A3, A4

FilterResultAccess will discard B1 and B2 because they are considered late, therefore, the application receives four filter results: A1 - A4.

In this case, LostCount is 0 since there are no gaps in sequence numbers for probe A
and DiscardCount is 2.

Scenario 2: Filter Results discarded towards the end of a run

In this scenario, the following packets arrived from probe A: A1, A2, A3, A5, A6, A7, A8, A9, A10.

Assume that A8, A9 and A10 are discarded by FilterResultAccess because of buffer overflow.

In this case, after the application receives A7, the LostCount is 1 (because A4 is missing), and the DiscardCount is 3 (because A8, A9 and A10 are discarded).

`unsigned long long DiscardCount()`

Description: This is the number of filter results discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The number of filter results discarded

Remarks: This is valid whether sequencing is turned on or not.

`unsigned long long DiscardDuplicateCount()`

Description: This is the number of filter results discarded because the filter result is considered a duplicate.

Parameters: None

Return Value: The number of filter results discarded because they are duplicates

Remarks: This is a relatively rare occasion, and usually occurs when the Filter Result Packets are duplicated by the network due to incorrect network configuration.

`unsigned long long DiscardLateCount()`

Description: This is the number of filter results discarded because the filter result is considered to be late.

Parameters: None

Return Value: The number of filter results discarded because they are considered to be late.

Remarks: There are several reasons that a filter result can be considered late. For example:

- The SFProbes are not time synchronized with the PRE.
- Multiple PREs are not time synchronized with one another.
- The MinBufferTime value is not set high enough to accommodate the difference in network latencies among the Filter Result Packets.

If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

`unsigned long long DiscardOutOfSequenceCount()`

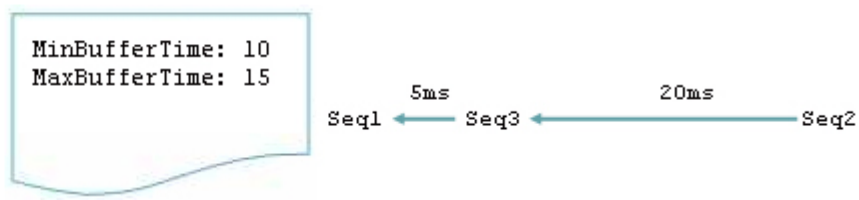
Description: This is the number of filter results discarded because the filter result is considered out of sequence. A filter result is considered out of sequence if the application is provided with a filter result of the same probe ID and a later sequence number.

Parameters: None

Return Value: The number of filter results discarded because the filter result is considered out of sequence

Remarks: This count can be affected by MinBufferTime and MaxBufferTime.

For example, filter results from the same probe ID arrive with the following sequence numbers and time gaps:



T is the time when the filter result with sequence number 1(Seq1) arrives.

T + 0 Seq 1 arrives

T + 5 Seq 3 arrives

T + 10 Seq 1 has been held for MinBufferTime, so it can be provided to application

T + 15 Seq 3 has been held for MinBufferTime, but it will wait 5 more ms to MaxBufferTime because a sequence is missing

T + 20 Seq 3 has been held MaxBufferTime, so it will be provided to application

T + 25 Seq 2 arrives

T + 35 Seq 2 is ready for the application, but it is discarded because it has an earlier sequence number than Seq 3

`unsigned long long DiscardOverflowCount()`

Description: This is the number of filter results discarded because the FilterResultAccess buffer is too full. Filter results are not inserted in the buffer once the number of filter results in the buffer reaches the BufferSize value.

Parameters: None

Return Value: The number of filter results discarded because the FilterResultAccess buffer is too full

Remarks: Changing the BufferSize value can affect the number of filter results that are discarded.

`unsigned long long InputFRPCount()`

Description: This is the number of filter result packets retrieved from the FilterResultAccess source. Only packets that appear to contain a legitimate FilterResults header is counted.

Parameters: None

Return Value: The number of filter result packets retrieved from the FilterResultAccess source

Remarks: This count is valid whether sequencing is turned on or not.

`unsigned long long DiscardFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The total number of Filter Result Packets discarded

Remarks: This count is valid whether sequencing is turned on or not.

`unsigned long long DiscardDuplicateFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates

`unsigned long long DiscardFragmentedFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Remarks: An unmatched filter result packets is discarded once it is kept in the buffer for a period set in MaxBufferTime. Changing the MaxBufferTime value can affect the number of fragmented Filter Result Packets that are discarded.

`unsigned long long DiscardLateFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be late.

Parameters: None

Return Value: The number of filter result packets discarded because the filter result packets are considered to be late

Remarks: If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

`unsigned long long DiscardOutOfSequenceFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be out of sequence.

Parameters: None

Return Value: The number of filter result packets discarded because they are considered to be out of sequence

Remarks: See DisardOutOfSequenceCount.

`unsigned long long DiscardOverflowFRPCount()`

Description: This is the number of filter result packets discarded because the FilterResultAccess buffer is too full.

Parameters: None

Return Value: The number of filter result packets discarded because the FilterResultAccess buffer is too full

Remarks: See DisardOverflowCount.

`FilterResult *Get()`

Description: This retrieves the next filter result from the FilterResultAccess buffer that is available at this moment. If no filter result is currently available, then a NULL is returned.

Parameters: None

Return Value: Returns a pointer to a FilterResult object or NULL

Remarks: When the pointer to the FilterResult object is no longer needed, call DeleteFilterResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains filter result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`FilterResult *Get(long timeout)`

Description: This retrieves the next filter result from the FilterResultAccess buffer, waiting the specified time for an available filter result. If no filter result is available at the end of the specified time, then a NULL is returned.

Parameters: timeout
Type: long
Specifies the maximum time in millisecond before this function returns. If timeout is 0, then this function behaves the same as Get().

Return Value: Returns a pointer to a FilterResult object or NULL

Remarks: When the pointer to the FilterResult object is no longer needed, call DeleteFilterResult to release resources used by the object.

`unsigned int  NumResultsInBuffer()`

Description: Retrieves the number of filter results in the buffer.

Parameters: None

Return Value: Number of filter results in the buffer.

Remarks: Allows an application to detect how full the PA-API internal buffer is.

C++ Static Version

Namespace: PacketAccessNS

Class: MetricsResultAccess

The MetricsResultsAccess class retrieves MetricsResult objects.

Methods

Each of the MetricsResultAccess class methods is described below.

`bool Start()`

Description: Start processing metrics result packets.

Parameters: None

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a MetricsResultAccess object by using CreateMetricsResultAccess. After setting the appropriate source properties, the application typically calls Start. The application can then call Get to retrieve available metrics results. When the application decides to stop processing metrics results, it should call Stop and then

DeleteMetricsResultAccess to free up resources used by MetricsResultAccess.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

if (!pAccess->SetSourceType("udp"))

{

// handle error

}

if (!pAccess->SetSourceProperty("port", 25000))

{

// handle error

}

if (!pAccess->Start())

{

// handle error

}

MetricsResult *pResult;

while ((pResult = pAccess->Get(timeout)) != NULL)

{

// handle MetricsResult

// ...

DeleteMetricsResult(pResult);

}

pAccess->Stop();

DeleteMetricsResultAccess(pAccess);
```

`bool Stop()`

Description: Stop processing metrics result packets.

Parameters: This function has no parameters.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Remarks: No more metrics results are available to the application after Stop is called.

Example: See **Start** example provided earlier in this section.

`bool LoadSettings(std::string s)`

Description: Configure the object according to the settings string.

Parameters: s

Type: std::string
A string containing the configuration settings of a MetricsResultAccess object.

Return Value: If the function succeeds, the return value is true. If the function fails, the return value is false. Call LastError to get extended error information.

Example:

```
MetricsResultAccess *pAccess = CreateMetricsResultAccess();

pAccess->BufferSize(50000);

pAccess->SetSourceType("file");

pAccess->SetSourceProperty("fileName", "test.pcap");

pAccess->Sequencing(false);

std::string s = pAccess->SaveSettings();

// s can be treated as an opaque string

// stored with other application settings.

// ...

MetricsResultAccess *pAccess2 = CreateMetricsResultAccess();

if (!pAccess2->LoadSettings(s))

{

  // handle error

}
```

```
if (!pAccess2->Start())

{

  // handle error

}
```

std::string SaveSettings()

Description: Save the current configuration of the object to a string.

Parameters: None

Return Value: Return a string representing the current configuration of the object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Example: See **LoadSettings** example provided earlier in this section.

MetricsResult *Get()

Description: This retrieves the next metrics result from the MetricsResultAccess buffer that is available at this moment. If no metrics result is currently available, then a NULL is returned.

Parameters: None

Return Value: Returns a pointer to a MetricsResult object or NULL.

Remarks: When the pointer to the MetricsResult object is no longer needed, call DeleteMetricsResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains metrics result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

MetricsResult *Get(long timeout)

Description: This retrieves the next metrics result from the MetricsResultAccess buffer, waiting the specified time for an available metrics result. If no metrics result is available at the end of the specified time, then a NULL is returned.

Parameters: timeout

Type: long

Specifies the maximum time in millisecond before this function returns. If timeout is 0, then this function behaves the same as Get().

Return Value: Returns a pointer to a MetricsResult object or NULL.

Remarks: When the pointer to the MetricsResult object is no longer needed, call DeleteMetricsResult to release resources used by the object.

PacketAccess Java API

Java API Library

This section describes the API Library for Java.

Overview

The PacketAccess API for Java is provided by packetaccess.jar. The API calls into a native code library.

- On Windows, the native code libraries supported are 32-bit or 64-bit DLL (packetAccess_java.dll).
- On Linux, the native code library is a 64-bit shared library (libpacketaccess_java.so).

Package packetaccess provides the following classes for accessing captured network packets from the PacketPortal system.

PacketAccess	Allows applications to access version information, create and delete other PacketAccess objects.
FilterResult	Represents an original captured packet and its metadata. This object is obtained through the FilterResultAccess class.
MetricsResult	Represents information contained in a metrics packet. An instance of this object is obtained through the MetricsResultAccess class, or created from previously obtained metric data.
PacketSourceTypeInfo	Provides information on a packet source supported by the PacketAccess Library.
PacketSourceTypeInfoList	Contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling PacketAccess.CreatePacketSourceTypeInfoList.
PacketResultAccess	Provides the base implementation for accessing packets generated by the PacketPortal system.

FilterResultAccess	Retrieves FilterResult objects.
MetricsResultAccess	Retrieves MetricsResult objects.
StringVector	Contains a collection of String objects.

Java

Package packetaccess

PacketAccess

Allows application to access version information, create and delete PacketPortal objects, etc.

Methods

Each of the PacketAccess class methods is described below.

```
public static int GetMajorVersion()
```

Description: This method returns the major version number.

Parameters: None

Return Value: Returns the major version number

```
public static int GetMinorVersion()
```

Description: This method returns the minor version number.

Parameters: None

Return Value: Returns the minor version number

```
public static int GetPatchVersion()
```

Description: This method returns the patch version number.

Parameters: None

Return Value: Returns the patch version number

`public static int GetBuildVersion()`

Description: This method returns the build version number.

Parameters: None

Return Value: Returns the build version number

`public static long GetBuildTime()`

Description: This method returns the build time.

Parameters: None

Return Value: Returns the build time in number of seconds since January 01, 1970, 00:00:00.

`public static String GetVersion()`

Description: This method returns a version string representing the version information.

Parameters: None

Return Value: Returns a string representing the version.

Example:

```
System.out.println("Version: " + PacketAccess.GetVersion());
```

`public static FilterResultAccess CreateFilterResultAccess()`

Description: This method creates a FilterResultAccess object.

Parameters: None

Return Value: Returns a FilterResultAccess object

Remarks: Call DeleteFilterResultAccess to release resources used by the FilterResultAccess object.

Example:

```
FilterResultAccess pa =
   PacketAccess.CreateFilterResultAccess();


if (pa == null)
{
  System.out.println("Error creating object");
  // exit or return
}
// ...
PacketAccess.DeleteFilterResultAccess(pa);
```

`public static void DeleteFilterResultAccess(FilterResultAccess p)`

Description: This method deletes a FilterResultAccess object.

Parameters: p
   Type: FilterResultAccess *
   A FilterResultAccess object

Return Value: None

Remarks: A FilterResultAccess object is created by using CreateFilterResultAccess.

Example: See the example in CreateFilterResultAccess

public static MetricsResultAccess CreateMetricsResultAccess()

Description: This method creates a MetricsResultAccess instance

Parameters: None

Return Value: Returns a MetricsResultAccess object.

Remarks: Call DeleteMetricsResultAccess to release resources used by the object.

Example:

```
MetricsResultAccess mra = PacketAccess.CreateMetricsResultAccess();

if (mra == null)

{

  System.out.println("Error creating object");

  //Handle error case

}

// ...

PacketAccess.DeleteMetricsResultAccess(mra);
```

public static void DeleteMetricsResultAccess(MetricsResultAccess toDelete)

Description: This method frees up the resources used by a MetricsResultAccess instance.

Parameters:

toDelete
Type: MetricsResultAccess
The instance to delete.

Return Value: None

Remarks: A MetricsResultAccess object is created by using the CreateMetricsResultAccess method.

Example: See example in CreateMetricsResultAccess

`FilterResult CreateFilterResult(byte[] header)`

Description: Create a FilterResult object from a previous stored header.

Parameters: header

Type: byte[]

A byte array of the header.

Return Value: A FitlerResult object

Remarks: The buffer used to create the FilterResult can be obtained by calling the Header()function of a previously retrieved FilterResult. This allows a FilterResult to be stored away and recreated for later processing. This version of the CreateFitlerResult() does not recreate the payload portion of the FilterResult.

The following information is lost for this re-created FilterResult:

* IsLate() - will always return false

* IsNewSequence() - will always return false

* Seconds() - will always be the same as ProbeSeconds()

* NSeconds() - will always be the same as ProbeNSeconds()

The reason the above information is lost for the recreated FilterResult is because the above functions is affected by the state of FitlerResultAccess (if sequencing is turned on), and not part of the data stored in the FilterResult header.

`FilterResult CreateFilterResult(byte[]  header, byte[] payload)`

Description: Create a FilterResult object from a previous stored header.

Parameters: header

Type: byte[]

A byte arry of the header.

payload

Type: byte[]

A byte array of the payload

Return Value: A FitlerResult object.

Remarks: See remarks for the other version of CreateFilterResult. This version of CreateFilterResult also restored the payload portion of the filter result. The payload buffer pointer can be obtained by calling the Payload() function of a previously retrieved FilterResult

`static void DeleteFilterResult(FilterResult p)`

Description: This method deletes a FilterResult object.

Parameters: p
Type: FilterResult
A FilterResult object.

Return Value: None

Remarks: A FilterResult object is returned by calling FilterResultAccess object's Get methods, when metrics result becomes available from the FilterResultAccess object. Call DeleteFilterResult to release resources used by the FilterResult object.

`public static MetricsResult CreateMetricsResult(byte[] data)`

Description: This method creates a MetricsResult instance from a byte array

Parameters: data
Type: byte[]
An array of metrics data. This data can be obtained by calling the MetricsData function on a MetricsResult instance. This allows an application to store and retrieve metrics data as a byte array.

Return Value: Returns a MetricsResult object.

Remarks: This function always returns a MetricsResult pointer, even if the buffer may point to invalid data. You may receive invalid data when you call functions in MetricsResult if the buffer contains invalid or insufficient data.

An application should release the resources used in this MetricsResult instance by calling DeleteMetricsResult after it is no longer needed.

Example:

```
byte [] metricsData = //load data from some data store

MetricsResult metricsResult = PacketAccess.CreateMetricsResult(metricsData);

// ...

PacketAccess.DeleteMetricsResult(metricsResult);
```

`public static void DeleteMetricsResult(MetricsResult toDelete)`

Description: This method frees up the resources used by a MetricsResult instance

Parameters: toDelete
Type: MetricsResult
A MetricsResult instance created earlier.

Return Value: None

Remarks: A MetricsResult instance can be obtained either through a MetricsResultAccess object or by creating one from a byte array using the CreateMetricsResult method.

`public static PacketSourceTypeInfoList CreatePacketSourceTypeInfoList()`

Description: This method creates a PacketSourceTypeInfoList object.

Parameters: None

Return Value: Returns a PacketSourceTypeInfoList object.

Remarks: Call DeletePacketSourceTypeInfoList to release resources used by the object.

Example:

```
PacketSourceTypeInfoList sourceTypeInfoList =
                PacketAccess.CreatePacketSourceTypeInfoList();
PacketAccess.GetPacketSourceTypeInfo(sourceTypeInfoList);
for (int i = 0; i < sourceTypeInfoList.Size(); i++)
{
    PacketSourceTypeInfo sourceTypeInfo =
                            sourceTypeInfoList.Get(i);
    System.out.println(sourceTypeInfo.Name() + " (" +
                       sourceTypeInfo.Description() + ")");
}
```

static void GetPacketSourceTypeInfo(PacketSourceTypeInfoList pList)

Description: This method fills in the PacketSourceTypeInfoList object with all the source types supported by the PacketAccess Library.

Parameters: pList
Type: GetPacketSourceTypeInfoList
The packet source type information is stored in pList.

Return Value: None

Remarks: The application can use this method to dynamically find out all the packet source types supported by the PacketAccess library. The name of the source type can be passed to the PacketResultAccess object's SetSourceType method.

Example: See example in CreatePacketSourceTypeInfoList.

static void DeletePacketSourceTypeInfoList(PacketSourceTypeInfoList pList)

Description: This method deletes the PacketSourceTypeInfoList object.

Parameters: pList
Type: PacketSourceTypeInfoList
A PacketSourceTypeInfoList.

Return Value: None

Remarks: A FilterResult object is returned by calling CreatePacketSourceTypeInfoList.

Example: See example in CreatePacketSourceTypeInfoList.

Java

Package packetaccess

FilterResult

The FilterResult class represents an original captured packet and its meta-data. This object is obtained through the FilterResultAccess class.

Methods

Each of the FilterResult class methods is described below.

`int HeaderLength() const`

Description: Returns the length of the header.

Parameters: None

Return Value: Number of bytes in the FilterResult header.

`byte[] Header()`

Description: Returns a byte array containing the FilterResult header.

Parameters: None

Return Value: None

`public int Version()`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Library version.

`public byte[] ProbeId()`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: None

Return Value: Returns the probe ID.

Remarks: A probe ID is not null-terminated. Applications should use the returned string's length method to return the length of the probe ID string.

`public long Seconds()`

Description: Returns the "Seconds" portion of the timestamp. This value may or may not be the same as the "ProbeSeconds" value depending on sequencing rules. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: If sequencing is turned on, a FilterResult's timestamp may be adjusted. See the Sequencing section in the Understanding Filter Results chapter for more information.

`public long NSeconds()`

Description: Returns the “nanoseconds” portion of the timestamp. This value may or may not be the same as the “ProbeNSeconds” value depending on sequencing rules.

Parameters: None

Return Value: Returns the “nanoseconds” portion of the timestamp.

`public long Sequence()`

Description: Returns a value that represents the sequence number of the result. The sequence number is interpreted as non-negative and wrap at 32-bit boundary.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a FilterResult is missing.

Example:

```
final long maxSequence = 2 ^ 32 - 1;

final int timeout = 1000;  // 1 second

FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

//... setup source and source properties

boolean newSequence = true;

long lastSequence = 0;


while ((res = pa.Get(timeout)) != null)

{

   long currentSequence = res.Sequence();
```

```
    if (newSequence ||
        (lastSequence + 1 == currentSequence) ||
        (lastSequence == maxSequence && currentSequence == 0))
    {
        lastSequence = currentSequence;
        newSequence = false;
    }
    else
    {
        // one or more FilterResult is lost
        System.out.println("last sequence = " + lastSequence);
        System.out.println("new sequence =   " + currentSequence);
    }
    PacketAccess.DeleteFilterResult(res);
}
pa.Stop();
PacketAccess.DeleteFilterResultAccess(pa);
```

public long FilterMatchBits()

Description:	Returns a value that represents which filters are matched
Parameters:	None
Return Value:	A value that represents which filters are matched.

public long CongestionCount()

Description: The number of packets that has matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow.

Parameters: None

Return Value: Returns a 29-bit value representing packets that matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow. This counter only applies to the side for this filtered packet.

Remarks: Only 29-bits of this value are valid. There are two congestion counters, one for equipment side and one for network side. When the SFProbe is unable to process a packet due to buffer overflow, it increments this counter for the side of this filtered packet.

Since the sequence number is not incremented in this situation, the application may receive filter results with consecutive sequence numbers, when in fact there are missing filtered packets. When the packets that the SFProbe are unable to process are on the same side as the next successfully injected filter result packets, application can check the CongestionCount for potential packet loss.

`public long InjectedCount()`

Description: Returns the number of captured packets that the SFProbe has successfully injected.

Parameters: None

Return Value: The number of captured packets that the SFProbe has successfully injected.

`public boolean IsBadFCS()`

Description: Returns whether the original captured packet has a bad FCS.

Parameters: None

Return Value: Returns true if the original captured packet has a bad FCS.

`public boolean IsHeaderOnly()`

Description: Returns whether the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

Remarks: If the original capture packet matches more than 1 filter, and not all of them has the "headers only" setting, then the captured payload may contain more than the protocol headers.

`public boolean IsInjectNet()`

Description: Returns whether the filter result was injected on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the filter result was injected on the network side of the SFProbe, otherwise returns false.

Remarks: This flag is not related to whether the original captured packet is on the network or equipment side of the SFProbe.

`public boolean IsLate()`

Description: Returns whether the filter result is considered late.

Parameters: None

Return Value: Returns true if the filter result is considered late. Otherwise returns false.

Remarks: A filter result is considered late if the application has already retrieved a filter result with a more recent timestamp. The application can optionally discard these filter results by calling FilterResultAccess object's DiscardLate(false) method.

`public boolean IsNet()`

Description: Returns whether the original captured packet is on the network side of the SFProbe.

Parameters: None

Return Value: Returns true if the original captured packet was captured on the network side of the SFProbe. Otherwise returns false, indicating the original captured packet was captured on the equipment side.

`public boolean IsNewSequence()`

Description: Returns whether the filter result indicates a new sequence.

Parameters: None

Return Value: Returns true if the filter result indicates a new sequence. Otherwise returns false.

Remarks: A filter result is considered a new sequence depending on sequencing rules. A filter result can be considered a new sequence if it is the first filter result from a particular SFProbe; a filter result that has a sequence number that is sufficiently far away from the previous filter result's sequence number from the same SFProbe; or if this filter result arrives a long time after other filter results. The SequenceBreakpoint and Timebreakpoint methods of FilterResultAccess can be used to adjust the breakpoint values.

`public boolean IsOnlyRoute()`

Description: Returns whether this machine and port is the only recipient of this FilterResult.

Parameters: None

Return Value: Returns true if this machine and port is the only recipient of this FilterResult, otherwise returns false.

Remarks: If this machine and port is the only recipient of this FilterResult, then a missing sequence in the filter result indicates that a filter result is unable to reach the application. If multiple machine and ports can be the intended recipients of this FilterResult, then a missing sequence number only indicates that a filter result may be missing.

`public boolean IsSliced()`

Description: Returns whether the filter expression requested the SFProbe to slice the payload of the original captured packet.

Parameters: None

Return Value: Returns true if the filter expression requested the SFProbe to slice the payload of the original captured packet. If slicing was not been requested, a false is returned.

Remarks: This value indicates the setting of the filter used to capture the original captured packet. The captured payload may not necessarily be truncated. If the original packet is too big, the captured payload may be truncated even if the filter does not specify truncation.

To determine if the Filter Result object is sliced, you can compare the "real packet length" and "payload length" of the Filter Result object. The payload is sliced if either of the two following conditions are true:

- the real packet length is zero
- the payload length is less than the real packet length

The "real packet length" and "payload length" are methods documented later in the FilterResult class.

`public boolean IsTimingLock()`

Description: Returns whether the filter result is captured when the SFProbe is time synchronized with the PRE.

Parameters: None

Return Value: Returns true if the filter result is captured when the SFProbe is time synchronized with the PRE. Otherwise returns false.

Remarks: If the SFProbe is not time synchronized with the PRE and original captured packets are expected to be captured by multiple SFProbes, then the timestamps may not reflect the true packet order. In that case, the application may consider either turning off sequencing, or setup time synchronization for the SFProbes involved.

`public boolean WasFragmented()`

Description: Returns whether the filter result was assembled from two filter result packets.

Parameters: None

Return Value: Returns true if the filter result was assembled from two filter result packets. Otherwise returns false.

Remarks: If the captured payload is over a specified limit (usually around the MTU of the network), then two filter result packets are needed to carry the metadata and the original captured packet as payload. This method is useful if the application wants to identify this situation.

`public int RealPacketLength()`

Description: Returns the original packet length in bytes, if known. This length does not include the 4 byte FCS of the original packet.

Parameters: None

Return Value: Returns the original packet length in number of bytes.

Remarks: The maximum number of bytes counted by the probe depends on its configuration and network encapsulation. Typically, the maximum number is around the maximum MTU size, or up to around 2000 bytes. When the actual number of bytes in the original packet is not known, the function returns 0.

The application can determine if the payload returned for the filter result is sliced by comparing the return value of PayloadLength function and

RealPacketLength function. The payload for the filter result is sliced if either of the following conditions exist:

- If RealPacketLength returns zero
- If PayloadLength is less than RealPacketLength

`public int PayloadLength()`

Description: Returns the length of the payload captured.

Parameters: None

Return Value: Returns number of bytes of the payload captured.

`public byte[] Payload()`

Description: Returns the payload.

Parameters: None

Return Value: Returns the captured packet.

Remarks: The captured payload maybe truncated by the probe depending on probe configuration. The payload does not include the 4 byte FCS.

`public long ProbeSeconds()`

Description: Returns the “seconds” portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The “seconds” portion of the real time that the original packet is captured by the SFProbe.

public long ProbeNSeconds()

Description: Returns the “nanoseconds” portion of the real time that the original packet is captured by the SFProbe.

Parameters: None

Return Value: The “nanoseconds” portion of the real time that the original packet is captured by the SFProbe.

Java

Package packetaccess

MetricsResult

The MetricsResult class represents metrics result packets generated by the SFProbe. An instance of this object is obtained through the MetricsResultAccess class or the static PacketAccess.CreateMetricsResult method.

Example

```
        public static void printResult(MetricsResult result) {

              long seconds = result.Seconds();

              long nSeconds = result.NSeconds();

              long sequence = result.Sequence();

              long netPacketCount = result.NetPacketCount();

              long eqtPacketCount = result.EqtPacketCount();



              String outputFormat = "%09d.%09d: seq [%d] net packets: %10d, 
eqt packets: %10d\n";

              System.out.format(outputFormat, 
seconds,nSeconds,sequence,netPacketCount,eqtPacketCount);



        }
```

Methods

Each of the MetricsResult class' methods is described below.

`bool IsPRETimeSync() const`

Description:	Indicates whether the PRE is time synced with the wall clock.
Parameters:	None
Return Value:	Returns true if the PRE is time sync with the wall clock.
Remarks:	The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off. When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

`unsigned int PRETimeSyncLossCount() const`

Description:	Indicates the number of times the PRE has lost time sync with the wall clock.
Parameters:	None
Return Value:	Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.
Remarks:	If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

`public int Version()`

Description:	Returns the version of the object.
Parameters:	None
Return Value:	The object version.
Remarks:	This version number identifies the metric result packet format version associated with this object. It is not related to the PacketAccess Library version.

`public byte[] ProbeId()`

Description: Returns the ID of the SFProbe that captured the original packet.

Parameters: None

Return Value: The ID of the probe

Remarks: A probe ID is not null-terminated.

`public long Seconds()`

Description: Returns the "Seconds" portion of the timestamp. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: MetricsResults are returned to the application in a first-in, first-out manner. The application may receive a MetricsResult with an earlier timestamp than the previous MetricsResult it receives. This is very common if there are multiple probes in different parts of the network sending MetricsResults to the same MetricsResultsAccess object, or if the probes are not time synchronized with the PRE.

`public long NSeconds()`

Description: Returns the "nanoseconds" portion of the timestamp.

Parameters: None

Return Value: Returns the "nanoseconds" portion of the timestamp.

`public long Sequence()`

Description: Returns an unsigned 16-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a MetricsResult is lost. A MetricsResult can be lost in transit, or due to buffer overflow.

A gap in the sequence number indicates that an intended MetricsResult for that probe was not delivered. In most cases, an application can safely ignore this situation.

There are two situations where an application may want to re-baseline its counters if there is a skipped MetricsResult:

1. If an application is interested in the filter byte counters. The filter byte counter may become invalid if a jumbo packet (more than about 2000 bytes) has been filtered. In that case, the filter byte counter invalid flag for that filter slot will be set. This flag is cleared for each Metrics Result request. If there is a lost Metrics Result, the application may not be aware that the filter byte counter invalid flag has been reset.

2. Under rare situations, if there are too many missed sequence numbers, then the counters may rollover more than once. Different counters rollover at different rates, depending on the counter's capacity and network traffic volume,. The application should decide when the number of missed sequences may cause a double rollover.

   For example, the theoretical maximum number of Ethernet frames per second on a 1G network is around 1.4 million frames per second, and the 29-bit total packet count can count up to around 536 million packets. So packet counter may rollover around every 6 minutes. If the metrics result request interval (configurable through System Manager) is every 5 minutes, then missing two consecutive Metrics Results may cause a double rollover.

`public long ResetCount()`

Description: Returns a value that represents how many times the application should treat this metrics result as a new baseline.

Parameters: None

Return Value: None

`public long  RetryCount()`

Description: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Parameters: None

Return Value: Returns a value that represents how many times the PRE had to retransmit the metrics result request to the SFProbe.

Remarks: This value may be important to applications that want to determine if the missing sequence number is due to the PRE unable to transmit or receive metrics results to and from the SFProbe.

`public int SFFTemperature()`

Description: Temperature of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit signed integer in increments of 1/256 °C

`public int SFFVcc()`

Description: Supply voltage of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit unsigned integer in increments of 100 μV.

`public int SFFTxBias()`

Description: Laser bias current of the SFProbe.

Parameters: None

Return Value: Returns a bit unsigned integer in increments of 2 μV.

`public int SFFTxPower()`

Description: Transmitted average optical power of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit unsigned integer in increments of 0.1 µW.

`public int SFFRxPower()`

Description: Received average optical power of the SFProbe.

Parameters: None

Return Value: Returns a 16 bit unsigned integer in increments of 0.1 µW.

`public int M2SAverageNSecond()`

Description: The average time needed for a packet to travel from PRE to SFProbe.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the PRE to the SFProbe.

`public int S2MAverageNSecond()`

Description: The average time needed for a packet to travel from SFProbe to PRE.

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average latency from the SFProbe to the PRE.

`public int TimingOffset()`

Description: M2SAverageNSecond minus the average of M2SAverageNSecond and S2MAverageNSecond. M2S - (M2S + S2M) / 2

Parameters: None

Return Value: Returns 32-bit integer.

Remarks: This represents the average round trip latency between the PRE to the SFProbe.

`public boolean IsTimingValid()`

Description: Indicates whether the IsTimingLock return value is valid.

Parameters: None

Return Value: true if the isTimingLock return value is valid.

`public boolean IsTimingLock()`

Description: Indicates whether the SFProbe is in time synchronization with the PRE at the time this packet is generated.

Parameters: None

Return Value: true if the SFProbe was in synchronization with the PRE when this packet was generated.

`public BigInteger EqtByteCount()`

Description: Total number of bytes on the EQT side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the EQT side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`public BigInteger NetByteCount()`

Description: Total number of bytes on the NET side.

Parameters: None

Return Value: A 48-bit unsigned integer representing the total number of bytes on the NET side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`public long EqtPacketsFiltered()`

Description: Total number of packets filtered on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered on the EQT side.

`public long EqtPacketsInjected()`

Description: Total number of packets injected by the SFProbe on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

`public long NetPacketsFiltered()`

Description: Total number of packets filtered by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered by the SFProbe on the NET side.

`public long NetPacketsInjected()`

Description: Total number of packets injected by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the NET side.

`public long EqtPacketCount()`

Description: Number of packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

`public long EqtIPv4Count()`

Description: Number of IPv4 packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

`public long EqtIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of “1110” (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

`public long EqtIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has the IPv4 destination address of 255.255.255.255.

`public long EqtIPv6Count()`

Description: Number of IPv6 packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`public long EqtIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`public long EqtIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`public long EqtTCPCount()`

Description: Number of TCP packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the TCP header.

`public long EqtUDPCount()`

Description: Number of UDP packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the UDP header.

`public long EqtSCTPCount()`

Description: Number of SCTP packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`public long EqtICMPCount()`

Description: Number of ICMP packets on the EQT side.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`public long Eqt63OrLessCount()`

Description: Number of packets on the EQT side that have less than 64 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is less than 64 bytes.

`public long Eqt64To127Count()`

Description: Number of packets on the EQT side that are between 64 and 127 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 64 and 127 bytes.

`public long Eqt128To255Count()`

Description: Number of packets on the EQT side that are between 128 and 255 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 128 and 255 bytes.

`public long Eqt256To511Count()`

Description: Number of packets on the EQT side that are between 256 and 511 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 256 and 511 bytes.

`public long Eqt512To1023Count()`

Description: Number of packets on the EQT side that are between 512 and 1023 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`public long Eqt1024To1500Count()`

Description: Number of packets on the EQT side that are between 1024 and 1500 bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 1024 and 1500 bytes.

`public long Eqt1501OrMoreCount()`

Description: Number of packets on the EQT side that are 1501 or more bytes.

Parameters: None

Return Value: An unsigned 29-bit value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is 1501 or more bytes.

`public long EqtMisalignedCount()`

Description: Number of packets that are misaligned on the EQT side.

Parameters: None

Return Value: 29-bit

`public long NetPacketCount()`

Description: Number of packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv4Count()`

Description: Number of IPv4 packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv6Count()`

Description: Number of IPv6 packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetTCPCount()`

Description: Number of TCP packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetUDPCount()`

Description: Number of UDP packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetSCTPCount()`

Description: Number of SCTP packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long NetICMPCount()`

Description: Number of ICMP packets on the NET side.

Parameters: None

Return Value: 29-bit

`public long Net63OrLessCount()`

Description: Number of packets on the NET side that have less than 64 bytes.

Parameters: None

Return Value: 29-bit

`public long Net64To127Count()`

Description: Number of packets on the NET side that are between than 64 and 127 bytes.

Parameters: None

Return Value: 29-bit

`public long Net128To255Count()`

Description: Number of packets on the NET side that are between than 128 and 255 bytes.

Parameters: None

Return Value: 29-bit

`public long Net256To511Count()`

Description: Number of packets on the NET side that are between than 256 and 511 bytes.

Parameters: None

Return Value: 29-bit

`public long Net512To1023Count()`

Description: Number of packets on the NET side that are between than 512 and 1023 bytes.

Parameters: None

Return Value: 29-bit

`public long Net1024To1500Count()`

Description: Number of packets on the NET side that are between than 1024 and 1500 bytes.

Parameters: None

Return Value: 29-bit

`public long Net1501OrMoreCount()`

Description: Number of packets on the NET side that are 1501 or more bytes.

Parameters: None

Return Value: 29-bit

`public long NetMisalignedCount()`

Description: Number of packets that are misaligned on the NET side.

Parameters: None

Return Value: 29-bit

`public long EqtFilterPacketCount(int index)`

Description: Return the number of filtered packet for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 29-bit

`public BigInteger EqtFilterByteCount(int index)`

Description: Return the number of filtered bytes for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 36-bit

Remarks: This counter may not be valid if the EqtFilterByteCountInvalid of the same filter slot returns true.

`public boolean EqtFilterByteCountInvalid(int index)`

Description: Return whether the EqtFilterByteCount of the same filter slot has a valid value.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: Return true if the EqtFilterByteCount of the same filter slot has a valid value.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`public long NetFilterPacketCount(int index)`

Description: Number of filtered packets on the NET side for a filter slot.

Parameters: None

Return Value: 29-bit

```
public BigInteger NetFilterByteCount(int index)
```

Description: Number of filtered bytes on the NET side for a filter slot.

Parameters: None

Return Value: 29-bit

```
public boolean NetFilterByteCountInvalid(int index)
```

Description: Indicates whether the filtered byte count on the NET side is valid for a filter slot.

Parameters: index
Type: int
A number between 0 and 15.

Return Value: Return true if the NetFilterByteCount of the same filter slot has a valid value.

```
public int MetricsDataLength()
```

Description: Return the size of the MetricsData object

Parameters: None

Return Value: Return the number of bytes needed to store a MetricsResult object

Remarks: In some cases, an application may want to store the entire MetricsResult object away for analysis at a later time. Application can allocate a buffer of size returned by the MetricsDataLength function.
Note: MetricsResult object size is the same for the same MetricsResult object version.

```
public byte[]  MetricsData()
```

Description: Copies the content of the MetricsResult object into a byte array.

Parameters: None

Return Value: A byte array containing the results data.

Remarks: You can store this data and use the PacketAccess.CreateMetricsResult to recreate a MetricsResult object at a later time.

```
public boolean IsPRETimeSync()
```

Indicates whether the PRE is time synced with the wall clock.

Returns true if the PRE is time sync with the wall clock.

The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off.

When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

```
public long PRETimeSyncLossCount()
```

Indicates the number of times the PRE has lost time sync with the wall clock.

Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

Java

Package packetaccess

PacketSourceTypeInfo

PacketSourceTypeInfo provides information on a packet source supported by the PacketPortal Library.

Methods

Each of the PacketSourceTypeInfo class methods is described briefly in the table below. Each method is described more completely in the list of methods below the table.

```
public String Name()
```

Description: Returns the name of the packet source.

Parameters: None

Return Value: Returns the name of the packet source.

Remarks: The name of the packet source can be passed to PacketResultAccess's SetSourceType method. This value is not case-sensitive.

`public String Description()`

Description: Returns the description of the packet source.

Parameters: None

Return Value: Returns the description of the packet source.

`public String HelpText()`

Description: Returns more information on packet source properties.

Parameters: None

Return Value: Returns more information on packet source properties.

Java

Package packetaccess

PacketSourceTypeInfoList

PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling PacketAccess.CreatePacketSourceTypeInfoList method.

Methods

Each of the PacketSourceTypeInfoList class methods is described below.

`public int Size()`

Description: Returns the number of objects in the collection.

Parameters: None

Return Value: Returns the number of objects in the collection.

`public PacketSourceTypeInfo Get(int index)`

Description: Returns a PacketSourceTypeInfo object by its position in the collection.

Parameters: index
Type: int
The position of the object to be retrieved. Positions start at 0.

Return Value: If index is valid, then returns a PacketSourceTypeInfo object. Otherwise, returns null.

Example: See the example in the CreatePacketSourceTypeInfoList method in PacketAccess class.

Java

Package packetaccess

PacketResultsAccess

The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.

Methods

Each of the PacketResultsAccess class methods is described briefly in the table below. Each method is described more completely in the list of methods below the table.

`public boolean SetSourceType(String sourceType)`

Description: Specifies the packet source.

Parameters: sourceType

Type: String

Set the packet source to one of the following: TCP, UDP, Libpcap, and File. This parameter is not case sensitive.

Note: Additional detailed information for each parameter is shown in the sections that immediately follow this section.

Return Value: Returns true if the sourceType parameter specified a supported source. Otherwise returns false. Call LastError for extended information.

Remarks: If this method is called after Start, then the running instance is stopped before the sourceType is applied. All counter information is lost.

Additional Parameter Information (Describing UDP, TCP, File, and Libpcap parameters in detail):

UDP retrieves PacketPortal packets using the UDP protocol. Use the UDP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using UDP. Since UDP provides unreliable data service, there may be packet loss between the PRE and the PacketAccess API application.

Property Name	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("UDP");

if (!pa.SetSourceProperty("port", 10001))

{

  // handle error
```

```
}

if (!pa.Start())

{

  // handle error

}

// add another listening port during the run.

if (!pa.SetSourceProperty("port", 10002))

{

  // handle error

}

// remove a listening port during the run.

if (!pa.SetSourceProperty("removePort", 10001))

{

  // handle error

}
```

TCP retrieves PacketPortal packets using TCP protocol. Use the TCP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using TCP. Since TCP provides a reliable data service, there may be minimal packet loss between the PRE and the PacketAccess API application. However, TCP adds some overhead and may cause more delays in the PacketAccess API.

Property Name:	**Description**	**Type**	**Allow Multiple?**	**Defaults**
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for	Positive integer 1 to MAX_INT	No	Use system

	all open sockets.			default
MaxConnections:	The maximum number of pending TCP connections. This value is passed to the system call listen, and is subject to the limit set by the operating system.	Positive integer	No	64

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("TCP");

if (!pa.SetSourceProperty("socketBufferSize", 1024 * 64))

{

  // handle error

}

// add a listening port during the run.

if (!pa.SetSourceProperty("port", 10001))

{

  // handle error

}

if (!pa.Start())

{

  // handle error

}
```

File retrieves PacketPortal packets from a PCAP capture file. The File source can be used for post processing of captured filter results packets, or used in the emulation mode with a regular PCAP file. TimeBreakpoint is ignored when using the File source. Inter-packet gap is also ignored, as the FilterResultAccess object returns the filter results (or emulated filter results) to the application as fast as it can read and sequence the packets.

Property Name	Description	Type	Allow Multiple?	Defaults

FileName:	Use packets from this PCAP file. Must set this property before Start. If file does not exist or user does not have sufficient permission to read it, then Start returns false. Setting of this property after a Start will be ignored until the next Start.	Pointer to a null-terminated char array	No	None
Loop:	The number of times the file is looped.	Integer. 0 means loop forever	No	1
IdleTime:	Idle this many milliseconds every "idleInterval" number of packets.	Positive integer	No	0
IdleInterval:	Idle the number milliseconds specified by "IdleTime" every this many number of packets.	Positive integer	No	100

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("file");

if (!pa.SetSourceProperty("fileName", "test.pcap"))

{

  // handle error

}

if (!pa.Start())

{

  // handle error

}
```

Libpcap retrieves PacketPortal packets from an Ethernet device in promiscuous mode. When the PacketPortal system is configured to send the filter results packets to the PacketAccess API using UDP, the application can choose to use the "libpcap" mode instead of the TCP mode. One advantage of using the libpcap source instead of the UDP source is that it can limit receiving packets by a device; the libpcap source also allows the application to receive filter results packets from any UDP port.

Property Name	Description	Type	Allow Multiple?	Defaults

Device:	Monitor this device (network interface name). This property may be set before or after Start, and it will take effect immediately if the device is successfully opened.	Pointer to a null-terminated char array	Yes	None
RemoveDevice:	Stop monitoring this device.	Pointer to a null-terminated char array	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("libpcap");

if (!pa.SetSourceProperty("device", "eth1"))

{

  // handle error

}

if (!pa.Start())

{

  // handle error

}
```

public boolean SetSourceProperty(String name, String value)

Description: Sets or adds a value to a source property.

Parameters: name
Type: String
Specifies the property name.

value
Type: String

Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

If the value of the property name is a numeric type, this method will convert the value string to the numeric value automatically.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("file");

pa.SetSourceProperty("fileName", "test.pcap");

pa.SetSourceProperty("loop", "2");

if (!pa.Start())

{

  // handle error

  String error = pa.LastError();

}
```

`public boolean SetSourceProperty(String name, int value)`

Description: Sets or adds a value to a source property.

Parameters: name
Type: String
Specifies the property name.

value
Type: int
Specifies the property value.

Return Value: Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

Remarks: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.SetSourceType("file");

pa.SetSourceProperty("fileName", "test.pcap");

pa.SetSourceProperty("loop", 2);

if (!pa.Start())

{

  // handle error

  String error = pa.LastError();

}
```

`public String GetSourceType()`

Description: Returns the currently specified source type

Parameters: None

Return Value: Returns the currently specified source type.

Remarks: Returns an empty string if the source is unspecified

`public void GetSourcePropertyNames(StringVector names)`

Description: Gets all the valid property names of the packet source associated with the object.

Parameters: v

Type: StringVector

All the property names for the packet source associated with the object is stored to a string vector.

Return Value: None

Remarks: The object should have a valid packet source before calling GetSourcePropertyNames.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

if (!pa.SetSourceType("UDP"))

{

  // handle error

}

StringVector names = new StringVector();

pa.GetSourcePropertyNames(names);

for (int i = 0; i < names.size(); i++)

   System.out.println("property: " + names.get(i));
```

public String GetSourceProperty(String name)

Description: Gets the first value associated with the property name.

Parameters: name
Type: String
Specifies the property name

Return Value: Returns the value associated with the property name.

Remarks: If there is no value associated with this property, GetSourceProperty returns an empty string. If there is more than one value associated with this property, then the first value is returned.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

if (pa.SetSourceType("TCP"))
```

```
{
  // handle error
}
String portValue = pa.GetSourceProperty("port");
if (portValue.length() == 0)
{
  // no port value set
}
```

public void GetSourceProperties(String name, StringVector values)

Description: Gets all the values associated with the property name.

Parameters: name
Type: String
Specifies the property name.

values
Type: StringVector
Store all the values associated with the property to the reference to the string vector.

Return Value: None

public void Emulate(boolean b)

Description: Turns emulation mode on or off.

Parameters: b
Type: boolean
When b is true, turns on emulation mode, otherwise, turns off emulation mode.

Return Value: None

`public boolean Emulate()`

Description: Returns the current state of emulation

Parameters: None

Return Value: Returns true if emulation is on, otherwise returns false.

`public String LastError()`

Description: Returns the last error.

Parameters: None

Return Value: Returns the last error.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();
if (!pa.Start())
{
   System.out.println("Error starting filter result access: " +
                      pa.LastError());
   pa.ClearError();
}
```

`public void ClearError()`

Description: Clears the last error.

Parameters: None

Return Value: None

Example: See example in LastError

`public void BufferSize(int size)`

Description: Specifies the maximum number of objects stored in the internal buffer.

Parameters: size
Type: int
Specifies the maximum number of objects.

Return Value: None

Remarks: Application should adjust the buffer size based on memory available for use with the API, how fast the PacketPortal packets are arriving and if there are high latencies among PacketPortal packets routed from multiple SFProbes.

The memory usage is roughly equal to (size * 2000) + (N * 2000) where N = number of actual objects in the buffer.

`public int BufferSize()`

Description: Returns the current setting of the buffer size.

Parameters: None

Return Value: Returns the maximum number of objects stored in the internal buffer.

Java

Package packetaccess

FilterResultAccess

The FilterResultAccess class retrieves FilterResult objects.

Methods

Each of the FilterResultAccess class methods is described below.

public boolean Start()

Description: Start processing filter result packets.

Parameters: None

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a FilterResultAccess object by using PacketAccess.CreateFilterResultAccess. After setting the appropriate source properties, the application typically calls Start. All counters are reset to zero at start. The application can then call Get to retrieve available filter results. When the application decides to stop processing filter results, it should call Stop and then PacketAccess.DeleteFilterResultAccess to free up resources used by FilterResultAccess.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

if (!pa.SetSourceType("udp"))

{

  // handle error

}

if (!pa.SetSourceProperty("port", 25000))

{

  // handle error

}

if (!pa.Start())

{

  // handle error

}

final int timeout = 1000;  // 1 second

FilterResult result;

while ((result = pa.Get(timeout)) != null)
```

```
{
  // handle FilterResult
  // ...
  PacketAccess.DeleteFilterResult(result);
}
pa.Stop();
PacketAccess.DeleteFilterResultAccess(pa);
```

`public boolean Stop()`

Description: Stop processing filter result packets.

Parameters: This method has no parameters.

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Remarks: No more filter results are available to the application after Stop is called. All counters are still valid until PacketAccess.DeleteFilterResultAccess or another Start is called.

Example: See **Start** example provided earlier in this section.

`boolean LoadSettings(String s)`

Description: Configure the object according to the settings string.

Parameters: s
Type: String
A string containing the configuration settings of a FilterResultAccess object.

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

pa.BufferSize(50000);

pa.SetSourceType("file"));

pa.SetSourceProperty("fileName", "test.pcap");

pa.Sequencing(false);

String s = pa.SaveSettings();

// s can be treated as an opaque string stored with other application settings.

...

FilterResultAccess pa2 = PacketAccess.CreateFilterResultAccess();

if (!pa2.LoadSettings(s))

{

  // handle error

}

if (!pa2.Start())

{

  // handle error

}
```

`public String SaveSettings()`

Description: Save the current configuration of the object to a string.

Parameters: None

Return Value: Return a string representing the current configuration of the object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Example: See **LoadSettings** example provided earlier in this section.

`public void Sequencing(boolean b)`

Description: This method sets sequencing on or off. When sequencing is turned on, the FilterResultAccess object will return filter results to the application according to a set of sequencing rules. If sequencing is turned off, filter results are made available to the application immediately on a first-in, first-out basis.

Parameters: b: boolean

When b is true, turns on sequencing, otherwise, turns off sequencing.

Remarks: An application using the FilterResultAccess object with sequencing turned on is subject to the following rules for sequencing:

- The timestamp for a FilterResult will be the same or later than the previous FilterResult provided to the application.
- Each FilterResult will be held for a minimum of the specified MinBufferTime before it is available to the application.
- Each FilterResult will be held for a maximum of the specified MaxBufferTime before it is available to the application.
- FilterResults of the same SFProbe are ordered by sequence numbers.
- FilterResults of different SFProbes are ordered by timestamps.
- For FilterResults of the same SFProbe, if a FilterResult with an earlier sequence has a later timestamp than another FilterResult, then the FilterResult with the later sequence will have its timestamp adjusted to be a later time than the FilterResult with an earlier sequence.
- For FilterResults from different SFProbes and have the same timestamp, a FilterResult that arrived earlier is provided to the application before a FilterResult that arrived later.

Example: See **Start** example provided earlier in this section.

`public boolean Sequencing()`

Description: Returns the sequencing setting.

Parameters: None

Return Value: Returns true if sequencing is turned on; otherwise, false is returned.

`public void DiscardLate(boolean b)`

Description: This sets the application to discard Filter Results that are received late. A filter result is considered "late" if the application has already retrieved a filter result with a more recent timestamp.

Parameters: b:

Type: boolean
When b is true, discard late filter results, otherwise, late filter results' timestamp will be adjusted to the timestamp that is most recently provided to the application, and then sequence accordingly.

Return Value: None

Remarks: DiscardLate takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`public boolean DiscardLate()`

Description: Returns the DiscardLate setting.

Parameters: None

Return Value: Returns true if the object is set to discard late filter results, otherwise, false is returned.

`public void MinBufferTime(int millisecond)`

Description: This specifies the time period (the minimum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This allows filter results from multiple SFProbes with different latencies to be time ordered. The MinBufferTime should typically be set to the maximum expected delta in the latency among feeds.

Parameters: timeout

Type: int

Specifies in millisecond the minimum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: MinBufferTime takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`public int MinBufferTime()`

Description: Returns the minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

Parameters: None

Return Value: The minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`public void MaxBufferTime(int millisecond)`

Description: This specifies the time period (the maximum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This is used when there are sequence number gaps in the Filter Results and the application is allowing extra time for the missing Filter Results to arrive.

Parameters: timeout

Type: int

Specifies in millisecond the maximum amount of time that a filter result is kept in the FilterResultAccess buffer.

Return Value: None

Remarks: If sequencing is turned off, this parameter is ignored for sequencing purposes, but is used to determine when an unmatched truncated or fragmented filter result packet will be discarded from the buffer.

`public int MaxBufferTime()`

Description: Returns the maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

Parameters: None

Return Value: The maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`public void SequenceBreakpoint(int breakpoint)`

Description: This sets the number of missing sequence numbers before FilterResultAccess treats the filter result as a new feed. A filter result sequence number specifies the order of the original captured packets. If there is a large gap in sequence numbers between two filter results from the same SFProbe, this may indicate that a feed has been stopped and restarted.

Parameters: breakpoint
Type: int
Specifies the number of missing sequence number.

Return Value: None

Remarks: An application should set sequence breakpoint to a large number (e.g. the same as the buffer size) if the FilterResultAccess object is expected to capture a feed that does not stop and restart. Conversely, if the application anticipates that the feed often stops and restarts during a running instance of the FilterResultAccess object, then it should set the sequence breakpoint to a relatively small number. If sequencing is turned off, this parameter is ignored.

`public int SequenceBreakpoint()`

Description: Returns the sequence breakpoint value.

Parameters: None

Return Value: The sequence breakpoint value

`public void TimeBreakpoint(int millisecond)`

Description: This sets the time period that the application waits to receive the next Filter Result Packet in the sequence. If a new Filter Result Packet has not arrived within the number of milliseconds specified by TimeBreakpoint value, then any Filter Result Packet that arrives after that will be treated as a new feed. This allows feeds to be stopped and restarted, and be sequenced correctly within the same running instance of FilterResultAccess.

Parameters: millisecond
Type: int
Specifies the time breakpoint in milliseconds. If the value is 0, then time breakpoint is not used.

Return Value: None

Remarks: Time breakpoint should be set to shortest expected time delay between stopping a feed and starting a feed. If sequencing is turned off, this parameter is ignored.

`public int TimeBreakpoint()`

Description: Returns the time breakpoint value.

Parameters: None

Return Value: The time breakpoint value

`public long NumProbes()`

Description: This is the number of unique SFProbes that the application has retrieved filter results from.

Parameters: None

Return Value: Number of unique SFProbes that the application has retrieved filter results from

Remarks: This counter does not count filter results that are in the FilterResultAccess buffer, but not yet retrieved by the application. This number is only valid when sequencing is turned on.

Example:

```
FilterResultAccess pa = PacketAccess.CreateFilterResultAccess();

// ... set source type and properties

FilterResult result;

while ((result = pa.Get(1000)) != null)

{

  // ...

  PacketAccess.DeleteFilterResult(result);

}

pa.Stop();

System.out.println("Total probes: " + pa.NumProbes());

PacketAccess.DeleteFilterResultAccess(pa);
```

`public java.math.BigInteger LostCount()`

Description: This number represents the number of missing filter results, by sequence number, for all SFProbes. This number is only valid when sequencing is turned on.

Parameters: None

Return Value: The number of missing filter results of all SFProbes by sequence number

Remarks: LostCount can be affected by TimeBreakpoint and SequenceBreakpoint values.

For example, filter results from the same probe ID with the following sequence numbers arrive:

Filter Result 1
<5 ms gap>
Filter Result 3
<20 ms gap>
Filter Result 15

Sequence Breakpoint	Time Breakpoint	Lost Count	Description
10	0	1	The filter result with sequence number 2 is considered lost, and the filter result with sequence number 15 is considered the start of a new sequence
20	0	12	The filter result with sequence number 2 and the filter results with sequence numbers 4 through 14 are all considered lost.
20	10	1	The filter result with sequence number 2 is considered lost because filter result 3 arrives less than 10 ms after filter result 1. Filter results with sequence 4 through 14 are not considered lost because filter results 15 arrives more than 10 ms later, even though filter results 15 is less than sequenceBreakpoint away from filter results 3.

Since LostCount counts all the gaps in filter results, it may not reflect the loss of filter results if the loss happens before the first filter results of a particular probe, or if the loss happens after the last filter result was processed by the application.

The following examples illustrate how LostCount and DiscardCount are related.

Scenario 1: Multiple Probes with Late Filter Results

In this scenario, the results from Probe B are all "late" and FilterResultAccess is configured to discard late packets. The following is the filter result packets arrival order (where Probe A results are: A1, A2, A3, and A4 -and- Probe B results are: B1 and B2):

A1, A2, B1, B2, A3, A4

FilterResultAccess will discard B1 and B2 because they are considered late, therefore, the application receives four filter results: A1 - A4.

In this case, LostCount is 0 since there are no gaps in sequence numbers for probe A
and DiscardCount is 2.

Scenario 2: Filter Results discarded towards the end of a run

In this scenario, the following packets arrived from probe A: A1, A2, A3, A5, A6, A7, A8, A9, A10.

Assume that A8, A9 and A10 are discarded by FilterResultAccess because of buffer overflow.

In this case, after the application receives A7, the LostCount is 1 (because A4 is missing), and the DiscardCount is 3 (because A8, A9 and A10 are discarded).

`public java.math.BigInteger DiscardCount()`

Description: This is the number of filter results discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The number of filter results discarded

Remarks: This is valid whether sequencing is turned on or not.

`public java.math.BigInteger DiscardDuplicateCount()`

Description: This is the number of filter results discarded because the filter result is considered a duplicate.

Parameters: None

Return Value: The number of filter results discarded because they are duplicates

Remarks: This is a relatively rare occasion, and usually occurs when the Filter Result Packets are duplicated by the network due to incorrect network configuration.

`public java.math.BigInteger DiscardLateCount()`

Description: This is the number of filter results discarded because the filter result is considered to be late.

Parameters: None

Return Value: The number of filter results discarded because they are considered to be late.

Remarks: There are several reasons that a filter result can be considered late. For example:

- The SFProbes are not time synchronized with the PRE.
- Multiple PREs are not time synchronized with one another.
- The MinBufferTime value is not set high enough to accommodate the difference in network latencies among the Filter Result Packets.

If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late method.

`public java.math.BigInteger DiscardOutOfSequenceCount()`

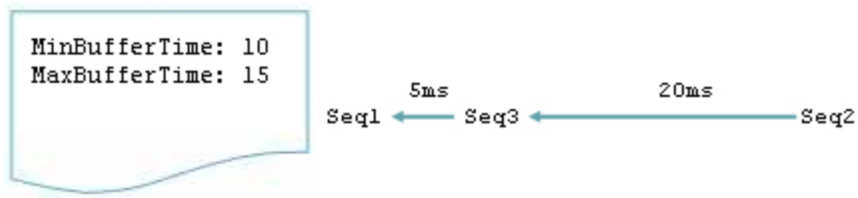
Description: This is the number of filter results discarded because the filter result is considered out of sequence. A filter result is considered out of sequence if the application is provided with a filter result of the same probe ID and a later sequence number.

Parameters: None

Return Value: The number of filter results discarded because the filter result is considered out of sequence

Remarks: This count can be affected by MinBufferTime and MaxBufferTime.

For example, filter results from the same probe ID arrive with the following sequence numbers and time gaps:



T is the time when the filter result with sequence number 1(Seq1) arrives.

T + 0 Seq 1 arrives

T + 5 Seq 3 arrives

T + 10 Seq 1 has been held for MinBufferTime, so it can be provided to application

T + 15 Seq 3 has been held for MinBufferTime, but it will wait 5 more ms to MaxBufferTime because a sequence is missing

T + 20 Seq 3 has been held MaxBufferTime, so it will be provided to application

T + 25 Seq 2 arrives

T + 35 Seq 2 is ready for the application, but it is discarded because it has an earlier sequence number than Seq 3

```
public java.math.BigInteger DiscardOverflowCount()
```

Description: This is the number of filter results discarded because the FilterResultAccess buffer is too full. Filter results are not inserted in the buffer once the number of filter results in the buffer reaches the BufferSize value.

Parameters: None

Return Value: The number of filter results discarded because the FilterResultAccess buffer is too full

Remarks: Changing the BufferSize value can affect the number of filter results that are discarded.

`public java.math.BigInteger InputFRPCount()`

Description: This is the number of filter result packets retrieved from the FilterResultAccess source. Only packets that appear to contain a legitimate FilterResults header are counted.

Parameters: None

Return Value: The number of filter result packets retrieved from the FilterResultAccess source

Remarks: This count is valid whether sequencing is turned on or not.

`public java.math.BigInteger DiscardFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object for any reason.

Parameters: None

Return Value: The total number of Filter Result Packets discarded

Remarks: This count is valid whether sequencing is turned on or not.

`public java.math.BigInteger DiscardDuplicateFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates

`public java.math.BigInteger DiscardFragmentedFRPCount()`

Description: This is the number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Parameters: None

Return Value: The number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Remarks: An unmatched filter result packets is discarded once it is kept in the buffer for a period set in MaxBufferTime. Changing the MaxBufferTime value can affect the number of fragmented Filter Result Packets that are discarded.

`public java.math.BigInteger DiscardLateFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be late.

Parameters: None

Return Value: The number of filter result packets discarded because the filter result packets are considered to be late

Remarks: If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late method.

`public java.math.BigInteger DiscardOutOfSequenceFRPCount()`

Description: This is the number of filter result packets discarded because the filter result packets are considered to be out of sequence.

Parameters: None

Return Value: The number of filter result packets discarded because they are considered to be out of sequence

Remarks: See DisardOutOfSequenceCount.

`public java.math.BigInteger DiscardOverflowFRPCount()`

Description: This is the number of filter result packets discarded because the FilterResultAccess buffer is too full.

Parameters: None

Return Value: The number of filter result packets discarded because the FilterResultAccess buffer is too full

Remarks: See DisardOverflowCount.

`public FilterResult Get()`

Description: This retrieves the next filter result from the FilterResultAccess buffer that is available at this moment. If no filter result is currently available, then a null is returned.

Parameters: None

Return Value: Returns a FilterResult object or null

Remarks: When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains filter result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`public FilterResult Get(int timeout)`

Description: This retrieves the next filter result from the FilterResultAccess buffer, waiting the specified time for an available filter result. If no filter result is available at the end of the specified time, then a null is returned.

Parameters: timeout
Type: int
Specifies the maximum time in millisecond before this method returns. If timeout is 0, then this method behaves the same as Get().

Return Value: Returns a FilterResult object or null.

Remarks: When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

`public int NumResultsInBuffer()`

Description: Retrieves the number of filter results in the buffer.

Parameters: None

Return Value: Number of filter results in the buffer.

Remarks: Allows an application to detect how full the PA-API internal buffer is.

Java

Package packetaccess

MetricsResultAccess

The MetricsResultAccess class retrieves MetricsResult objects.

Methods

Each of the MetricsResultAccess class’ methods is described below.

`public boolean Start()`

Description: Start processing metrics result packets.

Parameters: None

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Remarks: An application gets a MetricsResultAccess object by using PacketAccess.CreateMetricsResultAccess. After setting the appropriate source properties, the application typically calls Start. The application can then call Get to retrieve available filter results. When the application decides to stop processing results, it should call Stop and then PacketAccess.DeleteMetricsResultAccess to free up resources used by MetricsResultAccess.

Example:

```
MetricsResultAccess mra = PacketAccess.CreateMetricsResultAccess();

if (!mra.SetSourceType("udp"))

{

  // handle error

}

if (!mra.SetSourceProperty("port", 25000))

{

  // handle error

}

if (!mra.Start())

{

  // handle error

}

int timeout = 1000;  // 1 second

MetricsResult result;

while ((result = mra.Get(timeout)) != null)

{

  // handle MetricsResult

  // ...
```

```
  PacketAccess.DeleteMetricsResult(result);

}

mra.Stop();

PacketAccess.DeleteMetricsResultAccess(mra);
```

public boolean Stop()

Description: Stop processing metrics result packets.

Parameters: This method has no parameters.

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Remarks: No more metrics results are available to the application after Stop is called.

Example: See **Start** example provided earlier in this section.

boolean LoadSettings(String s)

Description: Configure the object according to the settings string.

Parameters: s

Type: String
A string containing the configuration settings of a MetricsResultAccess object.

Return Value: If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Example:

```
MetricsResultAccess mra = PacketAccess.CreateMetricsResultAccess();

mra.BufferSize(50000);

mra.SetSourceType("file"));

mra.SetSourceProperty("fileName", "test.pcap");

String s = mra.SaveSettings();
```

```
// s can be treated as an opaque string stored with other application settings.

// ...

MetricsResultAccess mra2 = PacketAccess.CreateMetricsResultAccess();

if (!mra2.LoadSettings(s))

{

  // handle error

}

if (!mra2.Start())

{

  // handle error

}
```

public String SaveSettings()

Description: Save the current configuration of the object to a string.

Parameters: None

Return Value: Return a string representing the current configuration of the object.

Remarks: Application should treat the returned string as an opaque value and should not alter it.

Example: See **LoadSettings** example provided earlier in this section.

public MetricsResult Get()

Description: This retrieves the next MetricsResult from the MetricsResultAccess buffer that is available at this moment. If no filter result is currently available, then null is returned.

Parameters: None

Return Value: Returns a MetricsResult object or null

Remarks: When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains metrics result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`public MetricsResult Get(int timeout)`

Description: This retrieves the next metrics result from the MetricsResultAccess buffer, waiting the specified time for an available metrics result. If no metrics result is available at the end of the specified time, then null is returned.

Parameters: timeout

Type: int

Specifies the maximum time in millisecond before this method returns. If timeout is 0, then this method behaves the same as Get().

Return Value: Returns a MetricsResult object or null.

Remarks: When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

Java

Package packetaccess

StringVector

StringVector represents an ordered collection of strings. The primary purpose of this class is to pass a collection of string objects between the application and the PacketPortal library.

Methods

Each of the StringVector class methods is described below.

`public StringVector()`

Description: Initialize the object to an empty collection.

Parameters: None

Return Value: None

`public void clear()`

Description: All the strings in the collection are removed.

Parameters: None

Return Value: None

Remarks: The size of the collection is zero after clear is called.

`public void add(String x)`

Description: Add a string to the collection.

Parameters: x:
Type: String
The string to be added to the collection.

Return Value: None

`public String get(int i)`

Description: Returns the string at position i

Parameters: i
Type: int
The position of the string in the collection.

Return Value: The string at position i.

Remarks: Index positions start at zero. If index is out of range, then s is set to an empty string.

`public long size()`

Description:	Returns the number of strings in the collection.
Parameters:	None
Return Value:	The number of string objects in the collection

PacketAccess Perl API

Perl API Module

This section describes the Perl PacketAccess API Module.

MODULE

PacketAccess.pm - The PacketAccess API module for Perl. The API uses a Perl extension module which relies on the compiled libraries:

- PacketAccess.dll – Windows extension modules supported are 32-bit or 64-bit DLL. Perl version 5.10 is required for 32-bit windows and 5.14 is required for 64-bit windows.

9. PacketAccess.so – Linux extension module is a 64-bit shared library. Perl version 5.10 is required for Linux.

NOTE: Perl programs using the PacketAccess Perl modules require a reference to the perl PacketAccess libraries. Two recommended approaches are:

> "use lib /opt/PacketPortal/sdk/default/bin/perl5/; " # add this to the import section of the perl program.
> "export PERLLIB5 before running the gadget # for example
>
> export PERL5LIB=/opt/PacketPortal/sdk/default/bin/perl5/

SYNOPSIS

```
use PacketAccess;

use lib /opt/PacketPortal/sdk/default/bin/perl5/;
```

CLASSES

Module provides the following classes for accessing captured network packets from the PacketPortal system.

PacketAccess	Static object that allows applications to access version information, create

	and delete other PacketAccess objects. `my $pFilter = PacketAccess::CreateFilterResultAccess();`
FilterResult	Represents an original captured packet and its metadata. This object is obtained through the FilterResultAccess class.
MetricsResult	Represents information contained in a metrics packet. A pointer to this object is obtained through the MetricsResultAccess class, or created from previously obtained metric data.
PacketSourceTypeInfo	Provides information on a packet source supported by the PacketAccess Library.
PacketSourceTypeInfoList	Contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling PacketAccess.CreatePacketSourceTypeInfoList.
PacketResultAccess	Provides the base implementation for accessing packets generated by the PacketPortal system.
FilterResultAccess	Retrieves FilterResult objects.
MetricsResultAccess	Retrieves MetricsResult objects.
StringVector	Contains a collection of String objects. These string objects are provided as a convenience class to the underlying C++ interface.

Perl

Global Functions

Global functions allow you to access version information, create and delete PacketPortal objects, and more.

METHODS

Each of the Global Functions class methods is described below.

GetMajorVersion

Returns the major version number.

```
my $MajorVersion =PacketAccess::GetMajorVersion();
```

```
print ("Major Version is: $MajorVersion \n");
```

GetMinorVersion

Returns the minor version number.

```
my $MinorVersion = PacketAccess::GetMinorVersion();

print ("Minor Version is: $MinorVersion \n");
```

GetPatchVersion

Returns the patch version number.

```
my $PatchVersion = PacketAccess::GetPatchVersion();

print ("Patch Version is: $PatchVersion \n");
```

GetBuildVersion

Returns the build version number.

```
my $BuildVersion = PacketAccess::GetBuildVersion();

print ("BuildVersion Version is: $BuildVersion \n");
```

GetBuildTime

Returns the time since epoch that the PacketAccess Library was built. Epoch time is the number of seconds since January 01, 1970, 00:00:00.

```
my $BuildTime = PacketAccess::GetBuildTime();
```

```
print (“BuildTime is: $BuildTime \n”);
```

GetVersion

Returns a version string representing the version information, in format: MM.mm.pppp (YYYY-MM-DD Build bbbb)

```
my $Version = PacketAccess::GetVersion();

print (“Version is: $Version \n”);
```

CreateFilterResultAccess

Creates a FilterResultAccess object.

```
my $pFilter = PacketAccess::CreateFilterResultAccess();

if ($pFilter == NULL)

{

   print ("Error creating packet access.  Maybe out of memory\n");

   return -1;

}

### ...

PacketAccess.DeleteFilterResultAccess($pFilter);
```

DeleteFilterResultAccess

Deletes a FilterResultAccess object.

```
PacketAccess.DeleteFilterResultAccess(pFilter);
```

DeleteMetricsResultAccess

DeleteFilterResult

Deletes a FilterResult object. A FilterResult object is returned by calling FilterResultAccess object's Get methods, when filter result becomes available from the FilterResultAccess object. Call DeleteFilterResult to release resources used by the FilterResult object.

```
while (($res = $pFilter->Get(TIMEOUT)) != null)

{

    PrintResult($res);

    PacketAccess::DeleteFilterResult($res);

}

PacketAccess::DeleteFilterResult($res);
```

DeleteMetricsResult

Deletes a MetricsResult object. A MetricsResult object is returned by calling MetricResultsAccess object's Get methods, when filter result becomes available from the MetricsResultAccess object. Call DeleteMetricsResult to release resources used by the MetricsResult object.

```
while (($res = $pMetrics->Get(TIMEOUT)) != null)

{

    PrintResult($res);

    PacketAccess::DeleteMetricsResult($res);

}

PacketAccess:: DeleteMetricsResult ($res);
```

CreatePacketSourceTypeInfoList

Creates a PacketSourceTypeInfoList object. Call DeletePacketSourceTypeInfoList to release resources used by the object.

```
sub PrintInfo {

        $sourceTypeInfoList =
PacketAccess::CreatePacketSourceTypeInfoList();

        PacketAccess::GetPacketSourceTypeInfo($sourceTypeInfoList);

        $InfoListLength = length(sourceTypeInfoList);

        for (my $i=0; $i < $InfoListLength; $i++)

        {

            $sourceTypeInfo = $sourceTypeInfoList->Get($i);

            print($sourceTypeInfo->Name() . " (" .

                        $sourceTypeInfo->Description() . ")");

        }

}
```

GetPacketSourceTypeInfo(pList)

Fills in the PacketSourceTypeInfoList pList object with all the source types supported by the PacketAccess Library. The application can use this function to dynamically find out all the packet source types supported by the PacketAccess library. The name of the source type can be passed to the PacketResultAccess object's SetSourceType function. See example in CreatePacketSourceTypeInfoList.

DeletePacketSourceTypeInfoList(pList)

Deletes the PacketSourceTypeInfoList pList object. See example in CreatePacketSourceTypeInfoList.

Perl

PacketAccess

NAME

PacketAccess implements the ability for applications to access version information, create and delete PacketPortal objects, and more.

DESCRIPTION

This class is the starting point for a programmer. Global information for the SDK and other objects can be created, retrieved or deleted:

- FilterResult
- MetricsResult
- PacketSourceTypeInfo
- PacketSourceTypeInfoList
- FilterResultAccess
- MetricsResultAccess

SYNOPSIS

```
use PacketAccess;

my $pFilter = PacketAccess::CreateFilterResultAccess;
```

Perl

FilterResult

NAME

FilterResult is the class that provides the metadata information for the original captured packet.

DESCRIPTION

The FilterResult class represents an original captured packet and its metadata. This object is obtained through the FilterResultAccess class. The methods provided for this class allow you to retrieve the metadata elements and utilize them.

SYNOPSIS

```
use strict;

use PacketAccess;

...

sub _pa_test {

   my $pa;

   $pa = PacketAccess::CreateFilterResultAccess();

   my $max_sequence = 2 ^ 32 - 1;

   my $timeout = 1000; # 1 second

   ###... setup source and source properties

   my $new_sequence = 1;

   my $last_sequence = 0;

   while ((my $res = $pa->Get($timeout)) != 0)

   {

      my $current_sequence = $res->Sequence();

      if ($new_sequence ||

      ($last_sequence + 1 == $current_sequence) ||

      ($last_sequence == $max_sequence && $current_sequence == 0))

      {

      $last_sequence = $current_sequence;

      $new_sequence = 0;

      }

      else

      {

      ## one or more FilterResult is lost

      print("last sequence = " + $last_sequence);

      print("new sequence = " + $current_sequence);

   }

      PacketAccess->DeleteFilterResult($res);
```

```
    }

    $pa->Stop();

    PacketAccess::DeleteFilterResultAccess($pa);

}
```

METHODS

Version()

Returns the version of this object. This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Module version.

ProbeId()

Returns the ID of the SFProbe that captures the original packet. A probe ID is not null-terminated. Applications should use the returned string's length method to return the length of the probe ID string.

Seconds()

Returns the "Seconds" portion of the timestamp. This value may or may not be the same as the "ProbeSeconds" value depending on sequencing rules. This value is the number of seconds since January 01, 1970 00:00:00.

If sequencing is turned on, a FilterResult's timestamp may be adjusted. See the Sequencing section in the Understanding Filter Results chapter for more information.

NSeconds()

Returns the "nanoseconds" portion of the timestamp. This value may or may not be the same as the "ProbeNSeconds" value depending on sequencing rules.

Sequence()

Returns a value that represents the sequence number of the result. The sequence number is interpreted as non-negative and wrap at 32-bit boundary. For a given SFProbe, an application can use the sequence number to determine if a FilterResult is missing.

FilterMatchBits()

Returns a value that represents which filters are matched.

CongestionCount()

The number of packets that has matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow.

Returns a 29-bit value representing packets that matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow. This counter only applies to the side for this filtered packet.

Only 29-bits of this value are valid. There are two congestion counters, one for equipment side and one for network side. When the SFProbe is unable to inject a filter result packet due to buffer overflow, it increments this counter for the side of the filtered packet.

Since the sequence number is not incremented in this situation, the application may receive filter results with consecutive sequence numbers, when in fact there are missing filtered packets. When the packets that the SFProbe is unable to process are on the same side as the next successfully injected filter result packets, application can check the CongestionCount for potential packet loss.

InjectedCount()

Returns the number of captured packets that the SFProbe has successfully injected.

IsBadFCS()

Returns whether the original captured packet has a bad FCS. Returns true if the original captured packet has a bad FCS.

IsHeaderOnly()

Returns whether the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet. Returns true if the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet.

If the original capture packet matches more than 1 filter, and not all of them has the "headers only" setting, then the captured payload may contain more than the protocol headers.

IsInjectNet()

Returns whether the filter result was injected on the network side of the SFProbe. Returns true if the filter result was injected on the network side of the SFProbe, otherwise returns false.

This flag is not related to whether the original captured packet is on the network or equipment side of the SFProbe.

IsLate()

Returns whether the filter result is considered "late". See definition below. Returns true if the filter result is considered late. Otherwise returns false.

A filter result is considered late if the application has already retrieved a filter result with a more recent timestamp. The application can optionally discard these filter results by calling FilterResultAccess object's DiscardLate(false) method.

IsNet()

Returns whether the original captured packet was captured on the network side of the SFProbe or on the equipment side. Returns true if the original captured packet was captured on the network side of the SFProbe. Otherwise returns false, indicating the original captured packet was captured on the equipment side.

IsNewSequence()

Returns whether the filter result indicates a new sequence. Returns true if the filter result indicates a new sequence. Otherwise returns false.

A filter result is considered a new sequence depending on sequencing rules. A filter result can be considered a new sequence if it is the first filter result from a particular SFProbe; a filter result that has a sequence number that is sufficiently far away from the previous filter result's sequence number from the same SFProbe; or if this filter result arrives a long time after other filter results. The SequenceBreakpoint and Timebreakpoint methods of FilterResultAccess can be used to adjust the breakpoint values.

IsOnlyRoute()

Returns whether this machine and port is the only recipient of this FilterResult. Returns true if this machine and port is the only recipient of this FilterResult, otherwise returns false.

If this machine and port is the only recipient of this FilterResult, then a missing sequence in the filter result indicates that a filter result is unable to reach the application. If multiple machine and ports can be the intended recipients of this FilterResult, then a missing sequence number only indicates that a filter result may be missing.

IsSliced()

Returns whether the payload contains only a portion of the original captured packet. Returns true if the filter expression requested the SFProbe to slice the payload of the original captured packet. If slicing was not been requested, a false is returned.

This value indicates the setting of the filter used to capture the original captured packet. The captured payload may not necessarily be truncated. If the original packet is too big, the captured payload may be truncated even if the filter does not specify truncation.

To determine if the Filter Result object is sliced, you can compare the "real packet length" and "payload length" of the Filter Result object. The payload is sliced if either of the two following conditions are true:

- the real packet length is zero
- the payload length is less than the real packet length

The "real packet length" and "payload length" are methods documented later in the FilterResult class.

IsTimingLock()

Returns whether the filter result was captured when the SFProbe is time synchronized with the PRE. Returns true if the filter result is captured when the SFProbe is time synchronized with the PRE. Otherwise returns false.

If the SFProbe is not time synchronized with the PRE and original captured packets are expected to be captured by multiple SFProbes, then the timestamps may not reflect the true packet order. In that case, the application may consider either turning off sequencing, or setup time synchronization for the SFProbes involved.

WasFragmented()

Returns whether the filter result was assembled from two filter result packets. Returns true if the filter result was assembled from two filter result packets. Otherwise returns false.

If the captured payload is over a specified limit (usually around the MTU of the network), then two filter result packets are needed to carry the metadata and the original captured packet as payload. This method is useful if the application wants to identify this situation.

RealPacketLength()

Returns the original packet length, if known. This length does not include the 4 byte FCS of the original packet. Returns the original packet length in number of bytes.

The maximum number of bytes counted by the probe depends on its configuration and network encapsulation. Typically, the maximum number is around the maximum MTU size, or up to around 2000 bytes. When the actual number of bytes in the original packet is not known, the function returns 0.
The application can determine if the payload returned for the filter result is sliced by comparing the return value of PayloadLength function and RealPacketLength function. The payload for the filter result is sliced if either of the following conditions exist:

- If RealPacketLength returns zero
- If PayloadLength is less than RealPacketLength

PayloadLength ()

Returns the length of the payload captured. Returns number of bytes of the payload captured.

Payload()

Returns the payload captured.

The captured payload maybe truncated by the probe depending on probe configuration. The payload does not include the 4 byte FCS.

ProbeSeconds()

Returns the "seconds" portion of the real time that the original packet is captured by the SFProbe.

ProbeNSeconds()

Returns the "nanoseconds" portion of the real time that the original packet is captured by the SFProbe.

Perl

Class: MetricsResult

The MetricsResult class represents metrics result packets generated by the SFProbe. A pointer to this object is obtained through the MetricsResultAccess class.

All of the metrics are available from a metricsResults object. The example below specifies the process for retrieving the metrics. The full example is available in the SDK 'gadgets' directory for perl5.

```
## Use Packet Access API to read metrics packets from a PCAP file

sub FromFile {

    my($filename, $bEmulate) = @_;


    my $pMetrics = PacketAccess::CreateMetricsResultAccess;

    if (! defined($pMetrics)) {

       print ("Error creating metrics access.  Maybe out of memory\n");

       return -1;

    }


    if (!$pMetrics->SetSourceType("file")) {

       my $error = $pMetrics->LastError();

       print "Error creating metrics access source: $error";

       PacketAccess::DeleteMetricsResultAccess($pMetrics);

       exit(-1);

    }
```

```
    $pMetrics->SetSourceProperty("filename", $filename);
    $pMetrics->Emulate($bEmulate);

    if (!$pMetrics->Start()) {
       my $error = $pMetrics->LastError();
       print "Error starting metrics access: $error";
       PacketAccess::DeleteMetricsResultAccess($pMetrics);
       exit(-1);
    }

    my $res;

    while (defined($res = $pMetrics->Get(TIMEOUT))) {
        PrintResult($res);
        PacketAccess::DeleteMetricsResult($res);
    }
    PacketAccess::DeleteMetricsResult($res);
    return 1;
}



sub PrintResult {
    my ($res) = @_;
    my $probeId = $res->ProbeId();
    my $seconds = $res->Seconds();
    my $nSeconds = $res->NSeconds();

    my $l1 = length($probeId);
```

```
    my $resultString1 = BinaryToText($probeId, $l1);


    print "Probe: " . $resultString1 . ": ";

        if ($humanTime) {

               my ($sec, $min, $hour, $mday, $mon, $year) = localtime($seconds);

               printf "%04d-%02d-%02d %02d:%02d:%02d", $year+1900, $mon+1, $mday,
$hour, $min, $sec;

        } else {

               print $seconds;

        }

        printf(".%09d\n", $nSeconds);

    print "  Version                : " .  $res->Version()."\n";

    print "  Sequence               : " .  $res->Sequence()."\n";

    print "  ResetCount             : " .  $res->ResetCount()."\n";

    print "  RetryCount             : " .  $res->RetryCount()."\n";


    print "  IsTimingValid          : " .  $res->IsTimingValid()."\n";

    print "  IsTimingLock           : " .  $res->IsTimingLock()."\n";

    print "  TimingOffset           : " .  $res->TimingOffset()."\n";

    print "  M2SAverageNSecond      : " .  $res->M2SAverageNSecond()."\n";

    print "  S2MAverageNSecond      : " .  $res->S2MAverageNSecond()."\n";


    print "  EqtPacketsFiltered     : " .  $res->EqtPacketsFiltered()."\n";

    print "  EqtPacketsInjected     : " .  $res->EqtPacketsInjected()."\n";

    print "  NetPacketsFiltered     : " .  $res->NetPacketsFiltered()."\n";

    print "  NetPacketsInjected     : " .  $res->NetPacketsInjected()."\n";


    print "  SFFVcc                 : " .  sprintf("%4.2f", $res->SFFVcc() * 0.0001)
." Volts  [".$res->SFFVcc()."]\n";

    print "  SFFTemperature         : " .  sprintf("%4.1f", $res->SFFTemperature() /
256) ." Centigrade  [".$res->SFFTemperature()."]\n";
```

```
    print "  SFFTxBias                : " .   sprintf("%4.2f", $res->SFFTxBias() * 
0.002) ." milli-Amps  [".$res->SFFTxBias()."]\n";

    print "  SFFTxPower               : " .   sprintf("%5.2f", sffPowerToDbm($res-
>SFFTxPower()) ) ." dBm  [".$res->SFFTxPower()."]\n";

    print "  SFFRxPower               : " .   sprintf("%5.2f", sffPowerToDbm($res-
>SFFRxPower()) ) ." dBm  [".$res->SFFRxPower()."]\n";



    print "  EqtByteCount             : " .   $res->EqtByteCount()."\n";

    print "  NetByteCount             : " .   $res->NetByteCount()."\n";


    print "  EqtPacketCount           : " .   $res->EqtPacketCount()."\n";

    print "  EqtIPv4Count             : " .   $res->EqtIPv4Count()."\n";

    print "  EqtIPv4MulticastCount    : " .   $res->EqtIPv4MulticastCount()."\n";

    print "  EqtIPv4BroadcastCount    : " .   $res->EqtIPv4BroadcastCount()."\n";

    print "  EqtIPv6Count             : " .   $res->EqtIPv6Count()."\n";

    print "  EqtIPv6MulticastCount    : " .   $res->EqtIPv6MulticastCount()."\n";

    print "  EqtIPv6BroadcastCount    : " .   $res->EqtIPv6BroadcastCount()."\n";

    print "  EqtTCPCount              : " .   $res->EqtTCPCount()."\n";

    print "  EqtUDPCount              : " .   $res->EqtUDPCount()."\n";

    print "  EqtSCTPCount             : " .   $res->EqtSCTPCount()."\n";

    print "  EqtICMPCount             : " .   $res->EqtICMPCount()."\n";


    print "  Eqt63OrLessCount         : " .   $res->Eqt63OrLessCount()."\n";

    print "  Eqt64To127Count          : " .   $res->Eqt64To127Count()."\n";

    print "  Eqt128To255Count         : " .   $res->Eqt128To255Count()."\n";

    print "  Eqt256To511Count         : " .   $res->Eqt256To511Count()."\n";

    print "  Eqt512To1023Count        : " .   $res->Eqt512To1023Count()."\n";

    print "  Eqt1024To1500Count       : " .   $res->Eqt1024To1500Count()."\n";

    print "  Eqt1501OrMoreCount       : " .   $res->Eqt1501OrMoreCount()."\n";

    print "  EqtMisalignedCount       : " .   $res->EqtMisalignedCount()."\n";
```

```
print "  NetPacketCount          : " .   $res->NetPacketCount()."\n";

print "  NetIPv4Count            : " .   $res->NetIPv4Count()."\n";

print "  NetIPv4MulticastCount   : " .   $res->NetIPv4MulticastCount()."\n";

print "  NetIPv4BroadcastCount   : " .   $res->NetIPv4BroadcastCount()."\n";

print "  NetIPv6Count            : " .   $res->NetIPv6Count()."\n";

print "  NetIPv6MulticastCount   : " .   $res->NetIPv6MulticastCount()."\n";

print "  NetIPv6BroadcastCount   : " .   $res->NetIPv6BroadcastCount()."\n";

print "  NetTCPCount             : " .   $res->NetTCPCount()."\n";

print "  NetUDPCount             : " .   $res->NetUDPCount()."\n";

print "  NetSCTPCount            : " .   $res->NetSCTPCount()."\n";

print "  NetICMPCount            : " .   $res->NetICMPCount()."\n";


print "  Net63OrLessCount        : " .   $res->Net63OrLessCount()."\n";

print "  Net64To127Count         : " .   $res->Net64To127Count()."\n";

print "  Net128To255Count        : " .   $res->Net128To255Count()."\n";

print "  Net256To511Count        : " .   $res->Net256To511Count()."\n";

print "  Net512To1023Count       : " .   $res->Net512To1023Count()."\n";

print "  Net1024To1500Count      : " .   $res->Net1024To1500Count()."\n";

print "  Net1501OrMoreCount      : " .   $res->Net1501OrMoreCount()."\n";

print "  NetMisalignedCount      : " .   $res->NetMisalignedCount()."\n";


print "  EqtFilterPacketCount     : " .  $res->EqtFilterPacketCount(0)."\n";

print "  EqtFilterByteCount       : " .  $res->EqtFilterByteCount(0)."\n";


unless($res->EqtFilterByteCountInvalid(0)) {

    $boolResult = "false";

} else {

    $boolResult = "true";

}
```

```
    print "  EqtFilterByteCountInvalid: " .  $boolResult."\n";
    print "  NetFilterPacketCount     : " .  $res->NetFilterPacketCount(0)."\n";
    print "  NetFilterByteCount       : " .  $res->NetFilterByteCount(0)."\n";
    unless ($res->NetFilterByteCountInvalid(0)) {
        $boolResult = "false";
    } else {
        $boolResult = "true";
    }
    print "  NetFilterByteCountInvalid: " .  $boolResult."\n";

        print "\n";
        select STDOUT; $|=1; # Flush print buffer
}


sub BinaryToText {
#   xx:xx:xx:xx:xx:xx
    my($stringId, $l) = @_;
    my $h = "";

    for (my $i=0; $i<$l; $i++) {
        my $vec =  vec($stringId, $i, 8) & 0xff + 0x100;
        my $v = sprintf "%02x", $vec;
        if ($i == 0) {
                     $h = $v;
              } else {
                     $h = $h . ":" . $v;
              }
```

```
    }

    return $h

}


sub sffPowerToDbm {

    my $raw_power = shift;

        my $micro_watts = $raw_power * 0.1;

        my $dbm = (10 * log10($micro_watts/1e6)) + 30;

        return $dbm;

}


sub log10 {

    my $n = shift;

    return log($n) / log(10);

}
```

Methods

Each of the MetricsResult class methods is described below.

IsPRETimeSync

Description:	Indicates whether the PRE is time synced with the wall clock.
Return Value:	Returns true if the PRE is time sync with the wall clock.
Remarks:	The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off. When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

PRETimeSyncLossCount

Description:	Indicates the number of times the PRE has lost time sync with the wall clock.

Return Value: Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

Remarks: If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

`Version()`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the metrics result packet format version associated with this object. It is not related to the PacketAccess Library version.

`ProbeId()`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: None

Return Value: StringVector value of the probe Id

Remarks: A probe ID is not null-terminated. Applications should use PAString's length function to return the length of the probe ID string. See the example above.

`Seconds()`

Description: Returns the "Seconds" portion of the timestamp. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: MetricsResults are returned to the application in a first-in, first-out manner. The application may receive a MetricsResult with an earlier timestamp than the previous MetricsResult it receives. This is very common if there are multiple probes in different parts of the network sending MetricsResults to the same MetricsResultsAccess object, or if the probes are not time synchronized with the PRE.

`NSeconds()`

Description: Returns the "nanoseconds" portion of the timestamp.

Parameters: None

Return Value: Returns the "nanoseconds" portion of the timestamp.

`Sequence()`

Description: Returns an unsigned 16-bit unsigned value that represents the sequence number of the result.

Parameters: None

Return Value: Returns a value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a MetricsResult is lost. A MetricsResult can be lost in transit, or due to buffer overflow.

A gap in the sequence number indicates that an intended MetricsResult for that probe was not delivered. In most cases, an application can safely ignore this situation.

There are two situations an application may want to re-baseline its counters if there is a skipped MetricsResult:

1. If an application is interested in the metrics byte counters. The metrics byte counter may become invalid if a jumbo packet (more than about 2000 bytes) has been filtered. In that case, the filter byte counter invalid flag for that filter slot will be set. This flag is cleared for each Metrics Result request. If there is a lost Metrics Result, the application may not be aware that the filter byte counter invalid flag has been reset.

2. Under rare situations, if there are too many missed sequence numbers, then the counters may rollover more than once. Different counters rollover at different rates, depending on the counter's capacity and network traffic volume,. The application should decide when the number of missed sequences may cause a double rollover.

   For example, the theoretical maximum number of Ethernet frames per second on a 1G network is around 1.4 million frames per second, and the 29-bit total packet count can count up to around 536 million packets. So packet counter may rollover around every 6 minutes. If the metrics result request interval (configurable through System Manager) is every 5 minute, then missing two consecutive Metrics Results may cause a double rollover.

Example:

```
sub _pa_test {

   my $pa;

   $pa = PacketAccess::CreateMetricsResultAccess();

   my $max_sequence = 2 ^ 16 - 1;

   my $timeout = 1000; # 1 second

   ###... setup source and source properties

   my $new_sequence = 1;

   my $last_sequence = 0;

   while ((my $res = $pa->Get($timeout)) != 0)

   {

      my $current_sequence = $res->Sequence();

      if ($new_sequence ||

      ($last_sequence + 1 == $current_sequence) ||

      ($last_sequence == $max_sequence && $current_sequence ==
0))

      {

      $last_sequence = $current_sequence;

      $new_sequence = 0;

      }

      else

      {

      ## one or more MetricsResult is lost

      print("last sequence = " + $last_sequence);

      print("new sequence = " + $current_sequence);

   }

      PacketAccess->DeleteMetricsResult($res);
```

```
    }

    $pa->Stop();

    PacketAccess::DeleteMetricsResultAccess($pa);

}
```

ResetCount()

Description: Returns a value that represents how many times the application should treat this metrics result as a new baseline. Returns a 16 bit integer.

RetryCount()

Description: Returns a value that represents how many times the PRE has to retransmit the metrics result request to the SFProbe. Returns a 16 bit integer. This value may be important to applications that want to determine if the missing sequence number is due to the PRE unable to transmit or receive metrics results to and from the SFProbe.

SFFTemperature()

Description: Temperature of the SFProbe. Returns a 16 bit signed integer in increments of 1/256 °C

SFFVcc()

Description: Supply voltage of the SFProbe. Returns a 16 bit unsigned integer in increments of 100 μV.

SFFTxBias()

Description: Transmitted laser bias current of the SFProbe. Returns a 16 bit unsigned integer in increments of 2 μV.

SFFTxPower()

Description: Transmitted average optical power of the SFProbe. Returns a 16 bit unsigned integer in increments of 0.1 µW.

`SFFRxPower()`

Description: Received average optical power of the SFProbe. Returns a 16 bit unsigned integer in increments of 0.1 µW.

`M2SAverageNSecond()`

Description: The average time needed for a packet to travel from PRE to SFProbe. Represents the average latency from the PRE to the SFProbe.

Parameters: None

Return Value: The 32-bit average number of nanoseconds needed for a packet to travel from PRE to the SFProbe.

`S2MAverageNSecond()`

Description: The average time needed for a packet to travel from SFProbe to PRE. Represents the average latency from the SFProbe to PRE.

Parameters: None

Return Value: The 32-bit average number of nanoseconds needed for a packet to travel from the SFProbe to the PRE.

`TimingOffset()`

Description: M2SAverageNSecond minus the average of M2SAverageNSecond and S2MAverageNSecond. M2S - (M2S + S2M) / 2.

Parameters: None

Return Value: Returns a 32-bit integer which represents the average round trip latency between the PRE and the SFProbe.

`IsTimingValid()`

Description: Indicates whether the IsTimingLock return value is valid. Returns 1 for true and null for false.

`IsTimingLock()`

Description: Indicates whether the SFProbe is in time synchronization with the PRE at the time this packet is generated. Returns 1 for true and null for false.

`EqtByteCount()`

Description: Total number of bytes on the EQT side. Returns a 48-bit unsigned integer representing the total number of bytes on the EQT side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`NetByteCount()`

Description: Total number of bytes on the NET side. Returns a 48-bit unsigned integer representing the total number of bytes on the NET side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

`EqtPacketsFiltered()`

Description: Total number of packets filtered on the EQT side.

Parameters: None

Return Value: A 32-bit integer representing the total number of packets filtered on the EQT side.

`EqtPacketsInjected()`

Description: Total number of packets injected by the SFProbe on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

`NetPacketsFiltered()`

Description: Total number of packets filtered by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit integer representing the total number of packets filtered by the SFProbe on the NET side.

`NetPacketsInjected()`

Description: Total number of packets injected by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit integer representing the total number of packets injected by the SFProbe on the NET side.

`EqtPacketCount()`

Description: Number of packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

`EqtIPv4Count()`

Description: Number of IPv4 packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

`EqtIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of "1110" (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

`EqtIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has the IPv4 destination address of 255.255.255.255.

`EqtIPv6Count()`

Description: Number of IPv6 packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`EqtIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`EqtIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`EqtTCPCount()`

Description: Number of TCP packets on the EQT side.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the TCP header.

`EqtUDPCount()`

Description:	Number of UDP packets on the EQT side.
Parameters:	None
Return Value:	29-bit integer value
Remarks:	This counter is incremented by the SFProbe if a packet header on the EQT side contains the UDP header.

`EqtSCTPCount()`

Description:	Number of SCTP packets on the EQT side.
Parameters:	None
Return Value:	29-bit integer value
Remarks:	This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`EqtICMPCount()`

Description:	Number of ICMP packets on the EQT side.
Parameters:	None
Return Value:	29-bit integer value
Remarks:	This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`Eqt63OrLessCount()`

Description:	Number of packets on the EQT side that have less than 64 bytes.
Parameters:	None
Return Value:	29-bit integer value
Remarks:	This counter is incremented by the SFProbe if a packet header on the EQT side is less than 64 bytes.

`Eqt64To127Count()`

Description: Number of packets on the EQT side that are between 64 and 127 bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 64 and 127 bytes.

`Eqt128To255Count()`

Description: Number of packets on the EQT side that are between 128 and 255 bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 128 and 255 bytes.

`Eqt256To511Count()`

Description: Number of packets on the EQT side that are between 256 and 511 bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 256 and 511 bytes.

`Eqt512To1023Count()`

Description: Number of packets on the EQT side that are between 512 and 1023 bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`Eqt1024To1500Count()`

Description: Number of packets on the EQT side that are between 1024 and 1500 bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 1024 and 1500 bytes.

`Eqt1501OrMoreCount()`

Description: Number of packets on the EQT side that are 1501 or more bytes.

Parameters: None

Return Value: 29-bit integer value

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is 1501 or more bytes.

`EqtMisalignedCount()`

Description: Number of packets that are misaligned on the EQT side. Returns 29-bit integer value.

`NetPacketCount()`

Description: Number of packets on the NET side. Returns 29-bit integer value.

`NetIPv4Count()`

Description: Number of IPv4 packets on the NET side. Returns 29-bit integer value.

`NetIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the NET side. Returns 29-bit integer value.

`NetIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the NET side. Returns 29-bit integer value.

`NetIPv6Count()`

Description: Number of IPv6 packets on the NET side. Returns 29-bit integer value.

`NetIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the NET side. Returns 29-bit integer value.

`NetIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the NET side. Returns 29-bit integer value.

`NetTCPCount()`

Description: Number of TCP packets on the NET side. Returns 29-bit integer value.

`NetUDPCount()`

Description: Number of UDP packets on the NET side. Returns 29-bit integer value.

`NetSCTPCount()`

Description: Number of SCTP packets on the NET side. Returns 29-bit integer value.

`NetICMPCount()`

Description: Number of ICMP packets on the NET side. Returns 29-bit integer value.

`Net63OrLessCount()`

Description: Number of packets on the NET side that have less than 64 bytes. Returns 29-bit integer value.

Net64To127Count()

Description: Number of packets on the NET side that are between than 64 and 127 bytes. Returns 29-bit integer value.

Net128To255Count()

Description: Number of packets on the NET side that are between than 128 and 255 bytes. Returns 29-bit integer value.

Net256To511Count()

Description: Number of packets on the NET side that are between than 256 and 511 bytes. Returns 29-bit integer value.

Net512To1023Count()

Description: Number of packets on the NET side that are between than 512 and 1023 bytes. Returns 29-bit integer value.

Net1024To1500Count()

Description: Number of packets on the NET side that are between than 1024 and 1500 bytes. Returns 29-bit integer value.

Net1501OrMoreCount()

Description: Number of packets on the NET side that are 1501 or more bytes. Returns 29-bit integer value.

NetMisalignedCount()

Description: Number of packets that are misaligned on the NET side.

Return Value: 29-bit integer value.

EqtFilterPacketCount(int index)

Description: Returns the number of filtered packet for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 29-bit integer value.

`EqtFilterByteCount(index)`

Description: Returns the number of filtered bytes for filter slot indicated by "index".

Parameters: index
Type: int
A number between 0 and 15.

Return Value: 36-bit integer value

Remarks: This counter may not be valid if the EqtFilterByteCountInvalid of the same filter slot returns true.

`EqtFilterByteCountInvalid(index)`

Description: Returns whether the EqtFilterByteCount of the same filter slot has a valid value.

Parameters: index
Type: int
A number between 0 and 7.

Return Value: Return 1 if the EqtFilterByteCount of the same filter slot has a valid value, null otherwise.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`NetFilterPacketCount(index)`

Description: Number of filtered packets on the NET side for a filter slot.

Parameters: index
Type: int
A number between 0 and 7.

Return Value: 29-bit integer value

Remarks: None

`NetFilterByteCount(index)`

Description: Number of filtered bytes on the NET side for a filter slot.

Parameters: index
Type: int
A number between 0 and 7.

Return Value: 36-bit integer value

Remarks: None

`NetFilterByteCountInvalid(index)`

Description: Indicates whether the filtered byte count on the NET side is valid for a filter slot.

Parameters: index
Type: int
A number between 0 and 7.

Return Value: Returns 1 if the NetFilterByteCount of the same filter slot has a valid value, null otherwise.

`bool HighTempAlarm() const`

Description: return value indicating the SFProbe's temperature is above the upper limit

Parameters:

Return Value:

Remarks:

`bool LowTempAlarm() const`

Description: return value indicating the SFProbe's temperature is below the lower limit

Parameters:

Return Value:

Remarks:

`bool HighVccAlarm() const`

Description: return value indicating the SFProbe's VCC is above the upper limit

Parameters:

Return Value:

Remarks:

`bool LowVccAlarm() const`

Description: return value indicating the SFProbe's VCC is below the lower limit

Parameters:

Return Value:

Remarks:

`bool TxBiasHighAlarm() const`

Description:

Parameters:

Return Value:

Remarks:

`bool TxBiasLowAlarm() const`

Description:

Parameters:

Return Value:

Remarks:

```
bool  TxPowerHighAlarm() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  TxPowerLowAlarm() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  RxPowerHighAlarm() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  RxPowerLowAlarm() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  HighTempWarning() const
```

Description:

Parameters:

Return Value:

Remarks:

`bool LowTempWarning() const`

Description:

Parameters:

Return Value:

Remarks:

`bool HighVccWarning() const`

Description:

Parameters:

Return Value:

Remarks:

`bool LowVccWarning() const`

Description:

Parameters:

Return Value:

Remarks:

`bool TxBiasHighWarning() const`

Description:

Parameters:

Return Value:

Remarks:

```
bool  TxBiasLowWarning() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  TxPowerHighWarning() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  TxPowerLowWarning() const
```

Description:

Parameters:

Return Value:

Remarks:

```
bool  RxPowerHighWarning() const
```

Description:

Parameters:

Return Value:

Remarks:

`bool  RxPowerLowWarning() const`

Description:

Parameters:

Return Value:

Remarks:

`MetricsDataLength()`

Description: Return the size of the MetricsData object

Parameters: None

Return Value: Return the number of bytes needed to store a MetricsResult object

Remarks: In some cases, an application may want to store the entire MetricsResult object away for analysis at a later time. Application can allocate a buffer of size returned by the MetricsDataLength function.
Note: MetricsResult object size is the same for the same MetricsResult object version.

`MetricsData(length)`

Description: Copies the content of the MetricsResult object into a byte array.

Parameters: buffer:
Type: pointer to a byte array
pointer to a buffer of size “length”.

length:
Type: int
size in bytes of the buffer.

Return Value: Returns the number of bytes copied.

Remarks: If “length” is greater than the number returned by MetricsDataLength, then only “MetricsDataLength” bytes are copied.

If “length” is less than MetricsDataLength, then only “length” bytes are copied. In that case, if you use this buffer to obtain a MetricsResult object using CreateMetricsResult function, the values returned by the

MetricsResult's functions are invalid.

`MetricsData()`

Description:	Copies the content of the MetricsResult object into the StringVector object.
Parameters:	None
Return Value:	StringVector
Remarks:	An application can use CreateMetricsResult function and the value in s to recreate a MetricsResult object.

`IsPRETimeSync()`

Indicates whether the PRE is time synced with the wall clock.

Returns true if the PRE is time sync with the wall clock.

The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off.

When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

`PRETimeSyncLossCount()`

Indicates the number of times the PRE has lost time sync with the wall clock.

Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

Perl

PacketSourceTypeInfo

NAME

PacketSourceTypeInfo is a class enabling the retrieval of the source types available within the PacketAccess API.

DESCRIPTION

SourceType is one of the parameters used when creating a PacketAccess::PacketResultsAccess object. It is helpful to know what SourceTypes are available to the application writer. This class is provided to support the retrieval of the SourceTypes supported on a particular system from the PacketAccess. API. The PacketSourceTypeInfo object can only be retrieved from a PacketSourceTypeInfoList object. The following example shows details in how this is done.

SYNOPSIS

```
use strict;

use PacketAccess;

…

sub PrintInfo {

        my $src_info_type_list = PacketAccess::CreatePacketSourceTypeInfoList();

        PacketAccess::GetPacketSourceTypeInfo($src_info_type_list);

        my $info_list_len = $src_info_type_list->size();

        print ("Length is $info_list_len\n");

        my $source_type_info = $src_info_type_list->Get(0);

        for (my $i=1; $i < $info_list_len; $i++)

        {

            print("\ntype is: ");

            print($source_type_info->Name() . " (" .

                    $source_type_info->Description() . ")");

            $source_type_info = $src_info_type_list->Get($i);

        }

}
```

METHODS

Name()

Returns the name of the packet source. The name of the packet source can be passed to PacketResultAccess's SetSourceType method. This value is not case-sensitive.

```
Examples of names are strings: "tcp", "udp", file, libpcap.

print($source_type_info->Name() . " (" .

              $source_type_info->Description() . ")");
```

Description()

Returns the description of the packet source.

Examples of descriptions are:

(read packets from an UDP port)

(read packets from an TCP port)

(read packets from standard PCAP file)

(read packets from an Ethernet device)

(internally generated packets)

```
print($source_type_info->Name() . " (" .

              $source_type_info->Description() . ")");
```

HelpText()

Returns more information on packet source properties.

```
 $source_type_info->Help();
```

Perl

PacketSourceTypeInfoList

NAME

PacketSourceTypeInfoList is an object containing a list of PacketSourceTypeInfo objects.

DESCRIPTION

This object is the container for all PacketSourceTypeInfo objects available to the running application. It is the class used to retrieve the PacketSourceTypeInfo objects from the environment.

SYNOPSIS

```
use strict;

use PacketAccess;

…

sub PrintInfo {

        my $src_info_type_list = PacketAccess::CreatePacketSourceTypeInfoList();

        PacketAccess::GetPacketSourceTypeInfo($src_info_type_list);

        my $info_list_len = $src_info_type_list->size();

        print ("Length is $info_list_len\n");

        my $source_type_info = $src_info_type_list->Get(0);

        for (my $i=1; $i < $info_list_len; $i++)

        {

            print("\ntype is: ");

            print($source_type_info->Name() . " (" .   $source_type_info-
>Description() . ")");

            $source_type_info = $src_info_type_list->Get($i);

        }

}
```

METHODS

Size()

Returns the number of objects in the collection.

```
my $info_list_length = $src_info_type_list->Size();
```

Get(int index)

Returns a PacketSourceTypeInfo object by its position in the collection. If index is valid, then returns a PacketSourceTypeInfo object. Otherwise, returns null.

Parameter: index – Integer index of the desired element to retrieve in the list.

```
my $_info = $src_info_type_list->Get(1);
```

Perl

PacketResultAccess

NAME

PacketResultAccess is a class provides the base implementation for accessing packets generated by the PacketPortal system. FilterResultAccess is derived from this class.

DESCRIPTION

PacketResultAccess is the base class for FilterResultAccess. The FilterResultAccess class retrieves FilterResult objects. An application using this class can manipulate the way that FilterResult objects are retrieved. For example, control of whether packets are sequenced or not, or discarded when packets are late. It can control the attributes of the internal buffer that collects packets.

SYNOPSIS

```
use constant TIMEOUT => 1000;

use strict;

use PacketAccess;

# create and return a FilterResultAccess object which is derived from the class
```

```
# PacketResultAccess. You cannot create a PacketResultAccess since it is virtual.

my $port = 16222;

my $bEmulate = 1;

my $pFilter = PacketAccess::CreateFilterResultAccess();

if ($pFilter == 0) {

   # Handle error

}

if (!$pFilter->SetSourceType("UDP")) {

   # Handle error

}

$pFilter->SetSourceProperty("port", $port);

$pFilter->Emulate($bEmulate);

if (!$pFilter->Start()) {

   # Handle error

}

my $res;

while (($res = $pFilter->Get(TIMEOUT)) != 0) {

   PrintResult($res);

   PacketAccess::DeleteFilterResult($res);

}

PacketAccess::DeleteFilterResult($res);

$pFilter->Stop();

PacketAccess::DeleteFilterResultAccess($pFilter);
```

METHODS

SetSourceType(sourceType)

Specifies the packet source. Returns true if the sourceType parameter specified a supported source. Otherwise returns false. Call LastError for extended information.

If this function is called after Start, then the running instance is stopped before the sourceType is applied. All counter information is lost.

Parameter: sourceType – This string sets the packet source to one of the following: TCP, UDP, Libpcap, and File. This parameter is not case sensitive.

Additional Parameter Information (Describing UDP, TCP, File, and Libpcap parameters in detail):

UDP retrieves PacketPortal packets using the UDP protocol. Use the UDP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using UDP. Since UDP provides unreliable data service, there may be packet loss between the PRE and the PacketAccess API application.

Property Name	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
$pAccess = PacketAccess::CreateFilterResultAccess();

$pAccess->SetSourceType("UDP");

if (!$pAccess->SetSourceProperty("port", 10001))

{

  // handle error

}

if (!$pAccess->Start())

{

  // handle error

}
```

```
// add another listening port during the run.

if (!$pAccess->SetSourceProperty("port", 10002))

{

  // handle error

}

// remove a listening port during the run.

if (!$pAccess->SetSourceProperty("removePort", 10001))

{

  // handle error

}
```

TCP retrieves PacketPortal packets using TCP protocol. Use the TCP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using TCP. Since TCP provides a reliable data service, there may be minimal packet loss between the PRE and the PacketAccess API application. However, TCP adds some overhead and may cause more delays in the PacketAccess API.

Property Name:	**Description**	**Type**	**Allow Multiple?**	**Defaults**
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default
MaxConnections:	The maximum number of pending TCP connections. This value is passed to the system call listen, and is subject to the limit set by the operating system.	Positive integer	No	64

Example:

```
my $pAccess = PacketAccess::CreateFilterResultAccess();

$pAccess->SetSourceType("TCP");

# optionally set default socket buffer size for listening TCP ports

if (!$pAccess->SetSourceProperty("socketBufferSize", 1024 * 64))

{

}

if (!$pAccess->SetSourceProperty("port", 10001))

{

  # handle error

}

if (!$pAccess->Start())

{

  # handle error

}
```

File retrieves PacketPortal packets from a PCAP capture file. The File source can be used for post processing of captured filter results packets, or used in the emulation mode with a regular PCAP file. TimeBreakpoint is ignored when using the File source. Inter-packet gap is also ignored, as the FilterResultAccess object returns the filter results (or emulated filter results) to the application as fast as it can read and sequence the packets.

Property Name	Description	Type	Allow Multiple?	Defaults
FileName:	Use packets from this PCAP file. Must set this property before Start. If file does not exist or user does not have sufficient permission to read it, then Start returns false. Setting of this property after a Start will be ignored until the next Start.	Pointer to a null-terminated char array	No	None
Loop:	The number of times the file is looped.	Integer. 0 means loop forever	No	1
IdleTime:	Idle this many milliseconds every “idleInterval” number	Positive	No	0

	of packets.	integer		
IdleInterval:	Idle the number milliseconds specified by "IdleTime" every this many number of packets.	Positive integer	No	100

Example:

```
my $pAccess = PacketAcess::CreateFilterResultAccess();

$pAccess->SetSourceType("file");

$pAccess->SetSourceProperty("fileName", "test.pcap");

$pAccess->SetSourceProperty("loop", 2);

if (!$pAccess->Start())

{

  ## Error situations:

  ##    File does not exist;

  ##    The application does not have sufficient permission to open the file;

  ##    The file is not a valid PCAP file.

}
```

Libpcap retrieves PacketPortal packets from an Ethernet device in promiscuous mode. When the PacketPortal system is configured to send the filter results packets to the PacketAccess API using UDP, the application can choose to use the "libpcap" mode instead of the TCP mode. One advantage of using the libpcap source instead of the UDP source is that it can limit receiving packets by a device; the libpcap source also allows the application to receive filter results packets from any UDP port.

Property Name	**Description**	**Type**	**Allow Multiple?**	**Defaults**
Device:	Monitor this device (network interface name). This property may be set before or after Start, and it will take effect immediately if the device is successfully opened.	Pointer to a null-terminated char array	Yes	None
RemoveDevice:	Stop monitoring this device.	Pointer to a null-terminated	Yes	None

		char array		
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
use strict;

use PacketAccess;

use Net::Pcap;

## find libpcap devices on the host by calling pcap_findalldevs

## this example uses the first device found by the libpcap library

sub _pcap {

  my $pAccess;

  my ($error, %description);

  foreach(Net::Pcap::findalldevs(\$error, \%description)) {

    my $dev = $_;

    if ($error){

      print $error;

      break;

    }

    print "$dev\n   $description{$dev}\n\n";

    $pAccess = PacketAccess::CreateFilterResultAccess();

    $pAccess->SetSourceType("libpcap");

    $pAccess->SetSourceProperty("device", $dev->name);

    if (!pAccess->Start())

    {

      $s = pAccess->LastError();

      print("Error: $s\n");
```

```
        }

        break;

      }

    }
```

SetSourceProperty(name, value)

Sets a source property by a string value. Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

If the value of the property name is a numeric type, this method will convert the value string to the numeric value automatically.

Parameter: name – This string value specifies the property name.

Parameter: value – This string value specifies the property value.

```
$fra = PacketAccess::CreateFilterResultAccess();

$fra->SetSourceType("file");

$fra->SetSourceProperty("fileName", "test.pcap");

$fra->SetSourceProperty("loop", "2");

if (!$fra->Start())

{

// handle error

$error = $fra->LastError();

}
```

SetSourceProperty(String name, int value)

Sets a source property by an integer value. Returns false when it can be immediately detected that the property value cannot be set successfully; otherwise returns true.

The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

Parameter: name – This string value specifies the property name.

Parameter: value – This integer value specifies the property value.

```
$fra = PacketAccess::CreateFilterResultAccess();

$fra->SetSourceType("file");

$fra->SetSourceProperty("fileName", "test.pcap");

$fra->SetSourceProperty("loop", "2");

if (!$fra->Start())

{

// handle error

$error = $fra->LastError();

}
```

GetSourceType()

Returns the source properties type in current use. Returns an empty string if the source is unspecified.

```
my $src_type = $fra->GetSourceType();
```

GetSourcePropertyNames(names)

Returns all the property names belonging to the source associated with this object.

Parameter: names – All the property names for the packet source associated with the object are stored to a string vector.

```
use PacketAccess;

use strict;

my $names = PacketAccess::StringVector->new();

my $values = PacketAccess::StringVector->new();

my $fra;

$fra = PacketAccess::CreateFilterResultAccess();

$fra->SetSourceType("UDP");

$fra->GetSourcePropertyNames($names);

my ($i, $j, $name, $value);

for ($i = 0; $i < $names->size(); $i++) {

   $name = $names->get($i);

   print("property: " . $name . "\n");

   $fra->GetSourceProperties($name, $values);

   for ($j = 0; $j < $values->size(); $j++){

      $value = $values->get($j);

      print("property name: " .  $name . "   property value: " . $value
. "\n");

   }

}
```

GetSourceProperty(name)

Returns the first value associated with the specified source property. If there is no value associated with this property, GetSourceProperty returns an empty string. If there is more than one value associated with this property, then the first value is returned.

Parameter: name – Specifies the property name

```
my $fpa = PacketAccess::CreateFilterResultAccess();
```

```
if ($fpa->SetSourceType("TCP"))

{

  ## handle error

}

my $port_value = $fpa->GetSourceProperty("port");

if (length($value) == 0)

{

  ## no port value set

}
```

GetSourceProperties(String name, StringVector values)

Returns all the values associated with the specified source properties.

Parameter: name – Specifies the property name

Parameter: values – All the property values for the packet source associated with the property are stored to a string vector.

```
use PacketAccess;

use strict;

my $names = PacketAccess::StringVector->new();

my $values = PacketAccess::StringVector->new();

my $fra;

$fra = PacketAccess::CreateFilterResultAccess();

$fra->SetSourceType("UDP");

$fra->GetSourcePropertyNames($names);

my ($i, $j, $name, $value);

for ($i = 0; $i < $names->size(); $i++) {

   $name = $names->get($i);
```

```
    print("property: " . $name . "\n");

    $fra->GetSourceProperties($name, $values);

    for ($j = 0; $j < $values->size(); $j++){

        $value = $values->get($j);

        print("property name: " .  $name . "   property value: " . $value
. "\n");

    }

}
```

Emulate(boolean b)

Sets the emulation mode.

Parameter: b – a true/false to turn on/off emulation mode.

Emulate()

Returns the state of the emulation mode. Returns a true value (1) if emulation is on, otherwise returns false.

LastError()

Reports the last error.

```
$fpa = PacketAccess::CreateFilterResultAccess();

if (!$fpa->Start())

{

    print("Error starting filter result access: " .
```

```
                 $fpa->LastError());
   $fpa->ClearError();
}
```

ClearError()

Clears the last error.

```
$fpa = PacketAccess::CreateFilterResultAccess();
if (!$fpa->Start())
{
   print("Error starting filter result access: " .
                 $fpa->LastError());
   $fpa->ClearError();
}
```

BufferSize(size)

Specifies the maximum number of objects stored in the internal buffer.

Application should adjust the buffer size based on memory available for use with the API, how fast the PacketPortal packets are arriving and if there are high latencies among PacketPortal packets routed from multiple SFProbes.

The memory usage is roughly equal to (size * 2000) + (N * 2000) where N = number of actual objects in the buffer.

Parameter: size – Integer value that specifies the maximum number of objects.

BufferSize()

Returns the maximum number of objects stored in the buffer.

Perl

FilterResultAccess

NAME

FilterResultAccess is a class for retrieving FilterResult objects.

DESCRIPTION

The FilterResultAccess class retrieves FilterResult objects. An application using this class can manipulate the way that FilterResult objects are retrieved. For example, control of whether packets are sequenced or not, or discarded when packets are late. It can control the attributes of the internal buffer that collects packets.

SYNOPSIS

```
use PacketAccess;

   my $pFilter = PacketAccess::CreateFilterResultAccess;

    my $pFilter = PacketAccess::CreateFilterResultAccess();

    if ($pFilter == 0)    {

 # Handle error

    }

    if (!$pFilter->SetSourceType("UDP")) {

# Handle error

    }


    $pFilter->SetSourceProperty("port", $port);

    $pFilter->Emulate($bEmulate);

    if (!$pFilter->Start())    {

# Handle error

    }
```

```
my $res;

while (($res = $pFilter->Get(TIMEOUT)) != 0) {

    PrintResult($res);

    PacketAccess::DeleteFilterResult($res);

}

PacketAccess::DeleteFilterResult($res);

pFilter->Stop();

PacketAccess::DeleteFilterResultAccess($pFilter);
```

METHODS

Start()

Starts processing filter results packets. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

An application gets a FilterResultAccess object by using PacketAccess:CreateFilterResultAccess. After setting the appropriate source properties, the application typically calls Start. All counters are reset to zero at start. The application can then call Get to retrieve available filter results. When the application decides to stop processing filter results, it should call Stop and then PacketAccess:DeleteFilterResultAccess to free up resources used by FilterResultAccess.

```
my $pFilter = PacketAccess::CreateFilterResultAccess();

if ($pFilter == 0) {

   # Handle error

}

if (!$pFilter->SetSourceType("UDP")) {

   # Handle error

}

$pFilter->SetSourceProperty("port", $port);

$pFilter->Emulate($bEmulate);
```

```
if (!$pFilter->Start()) {
   # Handle error
}
```

Stop()

Stops processing filter results packets. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

No more filter results are available to the application after Stop is called. All counters are still valid until PacketAccess.DeleteFilterResultAccess or another Start is called.

```
pFilter->Stop();
```

LoadSettings(s)

Configures the FilterResultAccess object according to the settings string passed in. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Parameter: s - A string containing the configuration settings of a FilterResultAccess object.

```
my $fra = PacketAccess::CreateFilterResultAccess();

$fra->BufferSize(50000);

$fra->SetSourceType("file"));

$fra->SetSourceProperty("fileName", "test.pcap");

$fra->Sequencing(0);

my $s = fra->SaveSettings();


...
```

```
my $fra2 = PacketAccess::CreateFilterResultAccess();

if (!$fra2->LoadSettings($s))

{

  // handle error

}

if (!$fra2->Start())

{

  // handle error

}
```

SaveSettings()

Saves the current configuration of the FilterResultAccess object to a string. The application should treat the returned string as an opaque value and should not alter it.

```
my $s = $fra->SaveSettings();
```

Sequencing (boolean)

Sets the FilterResult Sequencing to on or off. When sequencing is turned on, the FilterResultAccess object will return filter results to the application according to a set of sequencing rules. If sequencing is turned off, filter results are made available to the application immediately on a first-in, first-out basis.

An application using the FilterResultAccess object with sequencing turned on is subject to the following rules for sequencing:

- The timestamp for a FilterResult will be the same or later than the previous FilterResult provided to the application.
- Each FilterResult will be held for a minimum of the specified MinBufferTime before it is available to the application.
- Each FilterResult will be held for a maximum of the specified MaxBufferTime before it is available to the application.
- FilterResults of the same SFProbe are ordered by sequence numbers.

- FilterResults of different SFProbes are ordered by timestamps.
- For FilterResults of the same SFProbe, if a FilterResult with an earlier sequence has a later timestamp than another FilterResult, then the FilterResult with the later sequence will have its timestamp adjusted to be a later time than the FilterResult with an earlier sequence.
- For FilterResults from different SFProbes and have the same timestamp, a FilterResult that arrived earlier is provided to the application before a FilterResult that arrived later.

Parameter: b - When b is true, turns on sequencing, otherwise, turns off sequencing.

```
$fra->Sequencing(1);
```

Sequencing()

Returns whether sequencing is on or off. Returns true if sequencing is turned on; otherwise, false is returned.

```
$is_sequenced = $fra->Sequencing();
```

DiscardLate (boolean)

Allows late filter results to be discarded when sequencing is turned on. This sets the application to discard Filter Results that are received late. A filter result is considered “late” if the application has already retrieved a filter result with a more recent timestamp.

DiscardLate takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

Parameter: b - type Boolean, when b is true, discard the late filter results, otherwise, late filter results’ timestamp will be adjusted to the timestamp that is most recently provided to the application, and then sequence accordingly.

```
$fra->DiscardLate(1);
```

DiscardLate

Returns a value to show whether DiscardLate is on or off.

```
$is_late = $fra->DiscardLate();
```

MinBufferTime (int)

Specifies the minimum number of milliseconds that a filter result stays in the FilterResultAccess buffer when sequencing is turned on. This specifies the time period (the minimum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This allows filter results from multiple SFProbes with different latencies to be time ordered. The MinBufferTime should typically be set to the maximum expected delta in the latency among feeds.

MinBufferTime takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

Parameter: timeout – Integer that specifies in millisecond the minimum amount of time that a filter result is kept in the FilterResultAccess buffer.

```
$fra->MinBufferTime(25);
```

MinBufferTime()

Returns the minimum number of milliseconds that a filter result stays in the FilterResultAccess buffer.

```
$min_buffer_time =  $fra->MinBufferTime();
```

MaxBufferTime (int millisecond)

Specifies the maximum number of milliseconds that a filter result stays in the FilterResultAccess buffer before it is made available to the application when sequencing is turned on. This specifies the time period (the maximum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This is used when there are sequence number gaps in the Filter Results and the application is allowing extra time for the missing Filter Results to arrive.

If sequencing is turned off, this parameter is ignored for sequencing purposes, but is used to determine when an unmatched truncated or fragmented filter result packet will be discarded from the buffer.

Parameter: timeout – Integer that specifies in millisecond the maximum amount of time that a filter result is kept in the FilterResultAccess buffer.

```
$fra->MaxBufferTime(30);
```

MaxBufferTime()

Returns the maximum number of milliseconds that a filter result stays in the FilterResultAccess buffer.

```
$max_buffer_time =  $fra->MaxBufferTime();
```

SequenceBreakpoint(int breakpoint)

Specifies the number of allowable missing sequence numbers before FilterResultAccess treats the filter result as a new feed when sequencing is turned on. . A filter result sequence number specifies the order of the original captured packets. If there is a large gap in sequence numbers between two filter results from the same SFProbe, this may indicate that a feed has been stopped and restarted.

An application should set sequence breakpoint to a large number (e.g. the same as the buffer size) if the FilterResultAccess object is expected to capture a feed that does not stop and restart. Conversely, if the application anticipates that the feed often stops and restarts during a running instance of the FilterResultAccess object, then it should set the sequence breakpoint to a relatively small number. If sequencing is turned off, this parameter is ignored.

Parameter: breakpoint – Integer that specifies the number of missing sequence numbers before treating the filter result object as a new feed object.

```
$frp->SequenceBreakpoint(10);
```

SequenceBreakpoint()

Returns the SequenceBreakpoint value.

```
$sbp_val = $frp->SequenceBreakpoint();
```

TimeBreakpoint (int millisecond)

Specifies the number of milliseconds without a Filter Result Packet being received before the FilterResultAccess treats any subsequent filter result as a new feed when sequencing is turned on. If a new Filter Result Packet has not arrived within the number of milliseconds specified by TimeBreakpoint value, then any Filter Result Packet that arrives after that will be treated as a new feed. This allows feeds to be stopped and restarted, and be sequenced correctly within the same running instance of FilterResultAccess.

Time breakpoint should be set to shortest expected time delay between stopping a feed and starting a feed. If sequencing is turned off, this parameter is ignored.

Parameter: millisecond – Integer that specifies the time breakpoint in milliseconds. If the value is 0, then time breakpoint is not used.

```
$frp->TimeBreakpoint(100);
```

TimeBreakpoint()

Returns the TimeBreakpoint value.

```
$tbp_val = $frp->SequenceBreakpoint();
```

NumProbes()

Shows the number of unique SFProbes that the application has retrieved filter results from. This counter does not count filter results that are in the FilterResultAccess buffer, but not yet retrieved by the application. This number is only valid when sequencing is turned on.

```
$num_probes =  $frp->NumProbes();
```

LostCount()

Represents the number of missing filter results by sequence numbers for all SFProbes when sequencing is turned on. This number is only valid when sequencing is turned on.

LostCount can be affected by TimeBreakpoint and SequenceBreakpoint values.

Example: Filter results from the same probe ID with the following sequence numbers arrive:

Filter Result 1
<5 ms gap>
Filter Result 3
<20 ms gap>
Filter Result 15

Sequence Breakpoint	Time Breakpoint	Lost Count	Description
10	0	1	The filter result with sequence number 2 is considered lost, and the filter result with sequence number 15 is considered the start of a new sequence
20	0	12	The filter result with sequence number 2 and the filter results with sequence numbers 4 through 14 are all considered lost.
20	10	1	The filter result with sequence number 2 is considered lost because filter result 3 arrives less than 10 ms after filter result 1. Filter results with sequence 4 through 14 are not considered lost because filter results 15 arrives more than 10 ms later, even though filter results 15 is less than sequenceBreakpoint away from filter results 3.

Since LostCount counts all the gaps in filter results, it may not reflect the loss of filter results if the loss happens before the first filter results of a particular probe, or if the loss happens after the last filter result was processed by the application.

The following examples illustrate how LostCount and DiscardCount are related.

Scenario 1: Multiple Probes with Late Filter Results

In this scenario, the results from Probe B are all "late" and FilterResultAccess is configured to discard late packets. The following is the filter result packets arrival order (where Probe A results are: A1, A2, A3, and A4 -and- Probe B results are: B1 and B2):

A1, A2, B1, B2, A3, A4

FilterResultAccess will discard B1 and B2 because they are considered late, therefore, the application receives four filter results: A1 - A4.

In this case, LostCount is 0 since there are no gaps in sequence numbers for probe A and DiscardCount is 2.

Scenario 2: Filter Results discarded towards the end of a run

In this scenario, the following packets arrived from probe A: A1, A2, A3, A5, A6, A7, A8, A9, A10.

Assume that A8, A9 and A10 are discarded by FilterResultAccess because of buffer overflow.

In this case, after the application receives A7, the LostCount is 1 (because A4 is missing), and the DiscardCount is 3 (because A8, A9 and A10 are discarded).

```
$frp->Squencing(1);



# if the lost count for some period of time is greater than 10, reduce the
time breakpoint

if($frp-LostCount() > 10) {

   $frp->TimeBreakpoint($frp->TimeBreakpoint() - 1);

}
```

DiscardCount()

Returns the total number of filter results discarded by the FilterResultAccess.

```
$discard_cnt = $frp->DiscardCount();
```

DiscardDuplicateCount()

Returns the number of filter results discarded because the filter result is considered a duplicate. This is a relatively rare occasion, and usually occurs when the Filter Result Packets are duplicated by the network due to incorrect network configuration.

```
$discarded_dup_cnt = $frp->DiscardDuplicateCount();
```

DiscardLateCount()

Returns the integer value of filter results discarded because the filter result is considered to be late.

There are several reasons that a filter result can be considered late. For example:

- The SFProbes are not time synchronized with the PRE.
- Multiple PREs are not time synchronized with one another.
- The MinBufferTime value is not set high enough to accommodate the difference in network latencies among the Filter Result Packets.

If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late method.

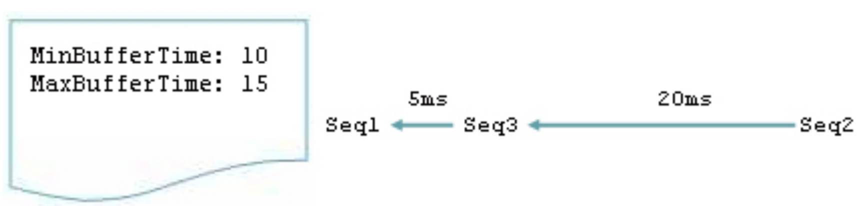
```
$discard_late_val = $frp->DiscardLateCount();
```

DiscardOutOfSequenceCount()

Shows the number of filter results discarded because the filter result is considered out of sequence.

This count can be affected by MinBufferTime and MaxBufferTime.

For example, filter results from the same probe ID arrive with the following sequence numbers and time gaps:



T is the time when the filter result with sequence number 1(Seq1) arrives.

T + 0 Seq 1 arrives

T + 5 Seq 3 arrives

T + 10 Seq 1 has been held for MinBufferTime, so it can be provided to application

T + 15 Seq 3 has been held for MinBufferTime, but it will wait 5 more ms to MaxBufferTime because a sequence is missing

T + 20 Seq 3 has been held MaxBufferTime, so it will be provided to application

T + 25 Seq 2 arrives

T + 35 Seq 2 is ready for the application, but it is discarded because it has an earlier sequence number than Seq 3

```
$discard_late_val = $frp->DiscardLateCount();
```

DiscardOverflowCount()

Returns the number of filter results discarded because the FilterResultAccess buffer is full.

Changing the BufferSize value can affect the number of filter results that are discarded.

```
$discard_ovrflow_cnt = $frp->DiscardOverflowCount();
```

InputFRPCount()

Returns the number of filter result packets retrieved from the FilterResultAccess source. Only packets that appear to contain a legitimate filter results header are counted.

This count is valid whether sequencing is turned on or not.

```
$input_frp_cnt = $frp->InputFRPCount();
```

DiscardFRPCount()

Returns the total number of Filter Result Packets discarded by the FilterResultAccess.

```
$discarded_frp_cnt = $frp->DiscardFRPCount();
```

DiscardDuplicateFRPCount()

Returns the number of Filter Result Packets discarded by the FilterResultAccess because they are duplicates.

```
$discarded_dup_frp_cnt = $frp->DiscardDuplicateFRPCount();
```

DiscardFragmentedFRPCount()

Returns the number of Filter Result Packets discarded by the FilterResultAccess because the matching filter result packet did not arrive in time to be reassembled.

An unmatched filter result packets is discarded once it is kept in the buffer for a period set in MaxBufferTime. Changing the MaxBufferTime value can affect the number of fragmented Filter Result Packets that are discarded.

```
$discarded_frag_frp_cnt = $frp->DiscardFragmentedFRPCount();
```

DiscardLateFRPCount()

Returns the number of filter result packets discarded because the filter result packet is considered to be late.

```
$discarded_late_frp_cnt = $frp->DiscardLateFRPCount();
```

DiscardOutOfSequenceFRPCount()

Returns the number of filter result packets discarded because the filter result packet is considered to be out of sequence.

See also: **DiscardOutOfSequenceCount** earlier in this section.

```
$discarded_oos_frp_cnt = $frp->DiscardOutOfSequenceFRPCount();
```

DiscardOverflowFRPCount()

Returns the number of filter result packets discarded because the FilterResultAccess buffer is full.

See also: **DiscardOverflowCount** earlier in this section.

```
$discarded_of_frp_cnt = $frp->DiscardOverflowFRPCount();
```

Get()

Retrieves the next filter result from the FilterResultAccess buffer that is available at this moment. If no filter result is currently available, then a null is returned.

When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains filter result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

```
while (($res = $pFilter->Get()) != 0)

{

    PrintResult($res);

    PacketAccess::DeleteFilterResult($res);

}
```

Get(int timeout)

Retrieves the next filter result from the FilterResultAccess buffer, waiting the specified time for an available filter result.

When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

Parameter: timeout – Maximum integer number of milliseconds for the timeout. If timeout is 0, then this method behaves the same as Get().

```
while (($res = $pFilter->Get(TIMEOUT)) != 0)

{

    PrintResult($res);

    PacketAccess::DeleteFilterResult($res);

}
```

NumResultsInBuffer()

Retrieves the number of filter results in the buffer.

Perl

MetricsResultAccess

NAME

MetricsResultAccess is a class for retrieving MetricsResult objects.

DESCRIPTION

The MetricsResultAccess class retrieves MetricsResult objects. An application using this class can manipulate the way that MetricsResult objects are retrieved. For example, control of whether packets are sequenced or not, or discarded when packets are late. It can control the attributes of the internal buffer that collects packets.

SYNOPSIS

```
use PacketAccess;

   my $metricsA = PacketAccess::CreateMetricsResultAccess;

    my $metricsA = PacketAccess::CreateMetricsResultAccess();

    if ($metricsA == 0)    {

 # Handle error

    }

    if (!$metricsA->SetSourceType("UDP")) {

# Handle error
```

```
    }


    $metricsA->SetSourceProperty("port", $port);

    $metricsA->Emulate($bEmulate);

    if (!$metricsA->Start())    {

# Handle error

    }


    my $res;

    while (($res = $metricsA->Get(TIMEOUT)) != 0) {

        PrintResult($res);

        PacketAccess::DeleteMetricsResult($res);

    }

    PacketAccess::DeleteMetricsResult($res);

    $metricsA->Stop();

    PacketAccess::DeleteMetricsResultAccess($metricsA);
```

METHODS

Start()

Starts processing metrics results packets. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

An application gets a MetricsResultAccess object by using PacketAccess:CreateMetricsResultAccess. After setting the appropriate source properties, the application typically calls Start. The application can then call Get to retrieve available metrics results. When the application decides to stop processing metrics results, it should call Stop and then PacketAccess:DeleteMetricsResultAccess to free up resources used by MetricsResultAccess.

```
my $metricsA = PacketAccess::CreateMetricsResultAccess();

if ($metricsA == 0) {

   # Handle error
```

```
}

if (!$metricsA->SetSourceType("UDP")) {

   # Handle error

}

$metricsA->SetSourceProperty("port", $port);

$metricsA->Emulate($bEmulate);

if (!$metricsA->Start()) {

   # Handle error

}
```

Stop()

Stops processing metrics results packets. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

No more metrics results are available to the application after Stop is called.

```
$metricsA->Stop();
```

LoadSettings(s)

Configures the MetricsResultAccess object according to the settings string. If the method succeeds, the return value is true. If the method fails, the return value is false. Call LastError to get extended error information.

Parameter: s - A string containing the configuration settings of a MetricsResultAccess object.

```
my $fra = PacketAccess::CreateMetricsResultAccess();

$fra->BufferSize(50000);

$fra->SetSourceType("file"));

$fra->SetSourceProperty("fileName", "test.pcap");

$fra->Sequencing(0);
```

```
my $s = fra->SaveSettings();


...

my $fra2 = PacketAccess::CreateMetricsResultAccess();

if (!$fra2->LoadSettings($s))

{

  // handle error

}

if (!$fra2->Start())

{

  // handle error

}
```

SaveSettings()

Saves the current configuration of the MetricsResultAccess object to a string. The application should treat the returned string as an opaque value and should not alter it.

my $s = $fra->SaveSettings();

Get(timeout)

Retrieves the next metrics result from the MetricsResultAccess buffer, waiting the specified time for an available metrics result. If no time is specified then the get returns immediately.

When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

Parameter: timeout – Maximum integer number of milliseconds for the timeout. If timeout is 0, then this method behaves the same as Get().

```
while (($res = $metricsA->Get(TIMEOUT)) != 0)

{

    PrintResult($res);
```

```
    PacketAccess::DeleteMetricsResult($res);

}
```

Get()

Retrieves the next metrics result from the MetricsResultAccess buffer that is available at this moment. If no metrics result is currently available, then a null is returned.

When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains metrics result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

```
while (($res = $metricsA->Get()) != 0)

{

    PrintResult($res);

    PacketAccess::DeleteMetricsResult($res);

}
```

Perl

StringVector

NAME

StringVector is a class used to contain ordered strings.

DESCRIPTION

StringVector represents an ordered collection of strings. The primary purpose of this class is to pass a collection of string objects between the application and the PacketPortal library. Examples of this can be found in the PacketAccessResult methods for GetSourcePropertyNames(stringvector) and GetSourceProperties(stringvector).

SYNOPSIS

```
use PacketAccess;

use strict;
```

```
my $names = PacketAccess::StringVector->new();

my $values = PacketAccess::StringVector->new();

my $fra;

$fra = PacketAccess::CreateFilterResultAccess();

$fra->SetSourceType("UDP");

$fra->GetSourcePropertyNames($names);

my ($i, $j, $name, $value);

for ($i = 0; $i < $names->size(); $i++) {

   $name = $names->get($i);

   print("property: " . $name . "\n");

   $fra->GetSourceProperties($name, $values);

   for ($j = 0; $j < $values->size(); $j++){

      $value = $values->get($j);

      print("property name: " .  $name . "   property value: " . $value . "\n");

   }

}
```

METHODS

new()

Initialize the content to an empty collection.

```
my $str = PacketAccess::StringVector->new();
```

clear()

All the objects in the collection are removed. The size of the collection is zero after clear is called.

```
my $str = PacketAccess::StringVector->new();
```

```
$str->clear();
```

add(x)

Add a string to the collection.

Parameter: x **-**The string to be added to the collection.

```
my $str = PacketAccess::StringVector->new();


$str->clear();

$str->add("comment:this is a comment");
```

get (i)

Returns the string at position i. Index positions start at zero. If index is out of range, then "i" is set to an empty string.

Parameter: i - The position of the string in the collection.

```
$str->get(1);
```

size()

Returns the number of strings in the collection.

```
my $size = $str->size();
```

PacketAccess Python API

Python API Library

This section describes the API Library for Python.

Overview

The PacketAccess API for Python is provided by the PacketAccess.py module. The API uses a Python extension module.

- On Windows, the extension modules supported are 32-bit or 64-bit DLL (_PacketAccess.pyd).
- On Linux, the extension module is a 64-bit shared library (_PacketAccess.so).

The PacketAccess module provides functions to access version information, create and delete other PacketPortal objects. The module also defines classes for accessing captured network packets from the PacketPortal system.

FilterResult	The FilterResult class represents an original captured packet and its metadata. This object is obtained through the FilterResultAccess class.
MetricsResult	The MetricsResult class represents information contained in a metrics packet. A pointer to this object is obtained through the MetricsResultAccess class, or created from previously obtained metric data.
PacketSourceTypeInfo	PacketSourceTypeInfo provides information on a packet source supported by the PacketAccess Library.
PacketSourceTypeInfoList	PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling PacketAccess.CreatePacketSourceTypeInfoList.
PacketResultAccess	The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.
FilterResultAccess	The FilterResultAccess class retrieves FilterResult objects.
MetricsResultAccess	The MetricsResultsAccess class retrieves MetricsResult objects.
StringVector	A collection of String objects.

Python

PacketAccess

The PacketAccess module defines the following functions:

`PacketAccess.`**`GetMajorVersion()`**

Return the major version number.

`PacketAccess.`**`GetMinorVersion()`**

Return the minor version number.

`PacketAccess.`**`GetPatchVersion()`**

Return the patch version number.

`PacketAccess.`**`GetBuildVersion()`**

Return the build version number.

`PacketAccess.`**`GetBuildTime()`**

Return the build time in number of seconds since January 01, 1970, 00:00:00.

`PacketAccess.`**`GetVersion()`**

Return a version string representing the version information.

Example:

```
>>> import PacketAccess

>>> print PacketAccess.GetVersion()
```

```
01.01.0000 (2011-11-10 Build 2650)
```

`PacketAccess.`**`CreateFilterResultAccess()`**

Return a FilterResultAccess object.

Note: Call DeleteFilterResultAccess to release resources used by the FilterResultAccess object.

Example:

```
import PacketAccess

pa = PacketAccess.CreateFilterResultAccess()

if (pa == None):

   print "Error"

   sys.exit(-1)

# ...

PacketAccess.DeleteFilterResultAccess(pa)
```

`PacketAccess.`**`DeleteFilterResultAccess`**`(`*`obj`*`)`

Delete the specified FilterResultAccess object.

Note: A FilterResultAccess object is created by using CreateFilterResultAccess.

`PacketAccess.CreateFilterResult(header)`

Remarks:

Description: Create a FilterResult object from a previous stored header.

Parameters:

None

Return Value: A FitlerResult object

Remarks: The header used to create the FilterResult can be obtained by calling the Header()function of a previously retrieved FilterResult. This allows a FilterResult to be stored away and recreated for later processing. This version of the CreateFitlerResult() does not recreate the payload portion of the FilterResult.

The following information is lost for this re-created FilterResult:

* IsLate() - will always return false

* IsNewSequence() - will always return false

* Seconds() - will always be the same as ProbeSeconds()

* NSeconds() - will always be the same as ProbeNSeconds()

The reason the above information is lost for the recreated FilterResult is because the above functions is affected by the state of FitlerResultAccess (if sequencing is turned on), and not part of the data stored in the FilterResult header.

`PacketAccess.CreateFilterResult(header, payload)`

Description: Create a FilterResult object from a previous stored header.

Parameters: header

A pointer to the header.

payload

A pointer to the payload.

Return Value: A FitlerResult object.

Remarks: See remarks for the other version of CreateFilterResult. This version of CreateFilterResult also restored the payload portion of the filter result. The payload buffer pointer can be obtained by calling the Payload() function of a previously retrieved FilterResult

`PacketAccess.`**`DeleteFilterResult`**`(`*`obj`*`)`

Delete the specified FilterResult object.

Note: FilterResult object is returned by calling FilterResultAccess object's Get methods, when filter result becomes available from the FilterResultAccess object. Call DeleteFilterResult to release resources used by the FilterResult object.

`PacketAccess.`**`CreatePacketSourceTypeInfoList()`**

Return a PacketSourceTypeInfoList object.

Note: Call DeletePacketSourceTypeInfoList to release resources used by the object.

Example:

```
import PacketAccess

list = PacketAccess.CreatePacketSourceTypeInfoList()

PacketAccess.GetPacketSourceTypeInfo(list)

for i in range(0, list.Size()):

   info = list.Get(i)

   print "%s: %s" % (info.Name(), info.Description())
```

`PacketAccess.`**`GetPacketSourceTypeInfo`**`(`*`obj`*`)`

Fill in the PacketSourceTypeInfoList object with all the source types supported by the PacketAccess Library.

Note: Application can use this method to dynamically find out all the packet source types supported by the PacketAccess library. The name of the source type can be passed to the PacketResultAccess object's SetSourceType method. See example in CreatePacketSourceTypeInfoList

`PacketAccess.`**`DeletePacketSourceTypeInfoList`**`(`*`obj`*`)`

Release resources used by the PacketSourceTypeInfoList object

Python

FilterResult

The FilterResult class represents an original captured packet and its metadata. This object is obtained through the FilterResultAccess class.

Functions

`int HeaderLength() const`

Description: Returns the length of the header.

Parameters: None

Return Value: Number of bytes in the FilterResult header.

`std::string Header() const`

Description: Returns a string containing the FilterResult header.

Parameters: None

Return Value: None

`Version()`

Returns the version of the object.

Note: This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Library version.

`ProbeId()`

Returns the ID of the SFProbe that captures the original packet.

Note: A probe ID is not null-terminated. Applications should use the returned string's length method to return the length of the probe ID string.

`Seconds()`

Return the "Seconds" portion of the timestamp. This value may or may not be the same as the "ProbeSeconds" value depending on sequencing rules. This value is the number of seconds since January 01, 1970 00:00:00.

Note: If sequencing is turned on, a FilterResult's timestamp may be adjusted. See the Sequencing section in the Understanding Filter Results chapter for more information.

`NSeconds()`

Return the "nanoseconds" portion of the timestamp. This value may or may not be the same as the "ProbeNSeconds" value depending on sequencing rules.

`Sequence()`

Returns a value that represents the sequence number of the result. The sequence number is interpreted as non-negative and wrap at 32-bit boundary.

Note: For a given SFProbe, an application can use the sequence number to determine if a FilterResult is missing.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

      # set source type and properties
```

```
        # ...
    if (not pa.Start()):
      print "Error %s" % (pa.LastError())
      PacketAccess.DeleteFilterResultAccess(pa)
      sys.exit(-1)

    maxSeq = 2 ^ 32 - 1
    newSeq = True
    timeout = 1000  # 1 second timeout
    # assume all the filter results come from the same probe
    while (True):
      res = pa.Get(timeout)
      if (res == None):
        break
      currSeq = res.Sequence()
      if (newSeq or (lastSeq + 1 == currSeq) or \
        (lastSeq == maxSeq and currSeq == 0)):
        # first filter result, or subsequent sequence
        lastSeq = currSeq
        newSeq = False
      else:
        # one of more FilterResult is lost
        print "last sequence is %s" % (lastSeq)
        print "current sequence is %s" % (currSeq)

      PacketAccess.DeleteFilterResult(res)
```

```
pa.Stop()

PacketAccess.DeleteFilterResultAccess(pa)
```

FilterMatchBits()

Return a value that represents which filters are matched

CongestionCount()

The number of packets that has matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow.

Returns a 29-bit value representing packets that matched one of the filters, but the SFProbe has been unable to inject due to internal buffer overflow. This counter only applies to the side for this filtered packet.

Only 29-bits of this value are valid. There are two congestion counters, one for equipment side and one for network side. When the SFProbe is unable to inject a filter result packet due to buffer overflow, it increments this counter for the side of the filtered packet.

Since the sequence number is not incremented in this situation, the application may receive filter results with consecutive sequence numbers, when in fact there are missing filtered packets. When the packets that the SFProbe is unable to process are on the same side as the next successfully injected filter result packets, application can check the CongestionCount for potential packet loss.

InjectedCount()

Returns the number of captured packets that the SFProbe has successfully injected.

IsBadFCS()

Return True if the original captured packet has a bad FCS.

`IsHeaderOnly()`

Returns whether the filter expression requested the SFProbe to capture only the protocol headers of the original capture packet. Returns true if the filter expression requested the SFProbe to capture only the protocol headers of the original captured packet. If the original capture packet matches more than one filter, and not all of them have the “headers only” setting, then the captured payload may contain more than the protocol headers.

`IsInjectNet()`

Return True if the filter result was injected on the network side of the SFProbe.

Note: This flag is not related to whether the original captured packet is on the network or equipment side of the SFProbe.

`IsLate()`

Return True if the filter result is considered late. Otherwise returns false.

Note: A filter result is considered late if the application has already retrieved a filter result with a more recent timestamp. The application can optionally discard these filter results by calling FilterResultAccess object’s DiscardLate(False) function.

`IsNet()`

Return True if the original captured packet was captured on the network side of the SFProbe. Otherwise returns False, indicating the original captured packet was captured on the equipment side.

`IsNewSequence()`

Returns True if the filter result indicates a new sequence. Otherwise returns False.

Note: A filter result is considered a new sequence depending on sequencing rules. A filter result can be considered a new sequence if it is the first filter result from a particular SFProbe; a filter result that has a sequence number that is sufficiently far away from the previous filter result's sequence number from the same SFProbe; or if this filter result arrives a long time after other filter results. The SequenceBreakpoint and TimeBreakpoint functions of FilterResultAccess can be used to adjust the breakpoint values.

`IsOnlyRoute()`

Return True if this machine and port is the only recipient of this FilterResult, otherwise returns False.

Note: If this machine and port is the only recipient of this FilterResult, then a missing sequence in the filter result indicates that a filter result is unable to reach the application. If multiple machine and ports can be the intended recipients of this FilterResult, then a missing sequence number only indicates that a filter result may be missing.

`IsSliced()`

Returns true if the filter expression requested the SFProbe to slice the payload of the original captured packet. If slicing was not been requested, a false is returned.

Note: This value indicates the setting of the filter used to capture the original captured packet. The captured payload may not necessarily be truncated. If the original packet is too big, the captured payload may be truncated even if the filter does not specify truncation.

To determine if the Filter Result object is sliced, you can compare the "real packet length" and "payload length" of the Filter Result object. The payload is sliced if either of the two following conditions are true:

- the real packet length is zero
- the payload length is less than the real packet length

The "real packet length" and "payload length" are methods documented later in the FilterResult class.

`IsTimingLock()`

Return True if the filter result is captured when the SFProbe is time synchronized with the PRE. Otherwise returns False.

Note: If the SFProbe is not time synchronized with the PRE and original captured packets are expected to be captured by multiple SFProbes, then the timestamps may not reflect the true packet order. In that case, the application may consider either turning off sequencing, or setup time synchronization for the SFProbes involved.

`WasFragmented()`

Return True if the filter result was assembled from two filter result packets. Otherwise returns False.

Note: If the captured payload is over a specified limit (usually around the MTU of the network), then two filter result packets are needed to carry the metadata and the original captured packet as payload. This function is useful if the application wants to identify this situation.

`RealPacketLength()`

Return the original packet length in bytes, if known. This length does not include the 4 byte FCS of the original packet.

Note: The maximum number of bytes counted by the probe depends on its configuration and network encapsulation. Typically, the maximum number is around the maximum MTU size, or up to around 2000 bytes. When the actual number of bytes in the original packet is not known, the function returns 0.

The application can determine if the payload returned for the filter result is sliced by comparing the return value of PayloadLength function and RealPacketLength function. The payload for the filter result is sliced if either of the following conditions exist:

- If RealPacketLength returns zero
- If PayloadLength is less than RealPacketLength

`PayloadLength()`

Return the number of bytes of the payload captured.

`Payload()`

Returns the captured packet.

Note: The captured payload maybe truncated by the probe depending on probe configuration.
The payload does not include the 4 byte FCS.

`ProbeSeconds()`

Returns the “seconds” portion of the real time that the original packet is captured by the SFProbe.

`ProbeNSeconds()`

Returns the “nanoseconds” portion of the real time that the original packet is captured by the SFProbe.

.

Python

MetricsResult

The MetricsResult class represents metrics result packets generated by the SFProbe. A pointer to this object is obtained through the MetricsResultAccess class, or the global function CreateMetricsResult.

All of the metrics are available from a metricsResult object. The example below specifies the process for retrieving the metrics. The full example is available in the SDK ‘gadgets’ directory for python.

Example

```
   #read from file.pcap and print out metrics information on any metrics results
packets.

   def FromFile(self, fileName):
```

```
        pa = PacketAccess.CreateMetricsResultAccess()
        if (not pa.SetSourceType("file")):
            print "Error create metrics access source: %s" % (pa.LastError())
            PacketAccess.DeleteMetricsResultAccess(pa)
            sys.exit(-1)

        pa.SetSourceProperty("filename", fileName)
        pa.Emulate(1)
        if (not pa.Start()):
            print "Error starting metrics access: %s" % (pa.LastError())
            PacketAccess.DeleteMetricsResultAccess(pa)
            sys.exit(-1)

        res = pa.Get(metricsResult.timeout)
        while(res != None):
            self.PrintResult(res)
            PacketAccess.DeleteMetricsResult(res)
            res = pa.Get(metricsResult.timeout)

        pa.Stop
        PacketAccess.DeleteMetricsResultAccess(pa)

    def PrintResult(self, res):
        probeId = res.ProbeId()
        seconds = res.Seconds()
        print "\n\n"
        t = datetime.datetime.fromtimestamp(seconds)
        print "  %s" % t
```

```
print "  ProbeID: %s " % (self.BinaryToText(probeId))
print "  Version                  : %s" % res.Version()
print "  Sequence                 : %s" % res.Sequence()
print "  ResetCount               : %s" % res.ResetCount()
print "  RetryCount               : %s" % res.RetryCount()
print "  SFFTemperature           : %s" % res.SFFTemperature()
print "  SFFVcc                   : %s" % res.SFFVcc()
print "  SFFTxBias                : %s" % res.SFFTxBias()
print "  SFFTxPower               : %s" % res.SFFTxPower()
print "  SFFRxPower               : %s" % res.SFFRxPower()
print "  TimingOffset             : %s" % res.TimingOffset()
print "  M2SAverageNSecond        : %s" % res.M2SAverageNSecond()
print "  S2MAverageNSecond        : %s" % res.S2MAverageNSecond()
print "  IsTimingValid            : %s" % res.IsTimingValid()
print "  IsTimingLock             : %s" % res.IsTimingLock()
print "  EqtByteCount             : %s" % res.EqtByteCount()
print "  NetByteCount             : %s" % res.NetByteCount()
print "  EqtPacketsFiltered       : %s" % res.EqtPacketsFiltered()
print "  EqtPacketsInjected       : %s" % res.EqtPacketsInjected()
print "  NetPacketsFiltered       : %s" % res.NetPacketsFiltered()
print "  NetPacketsInjected       : %s" % res.NetPacketsInjected()
print "  EqtPacketCount           : %s" % res.EqtPacketCount()
print "  EqtIPv4Count             : %s" % res.EqtIPv4Count()
print "  EqtIPv4MulticastCount    : %s" % res.EqtIPv4MulticastCount()
print "  EqtIPv4BroadcastCount    : %s" % res.EqtIPv4BroadcastCount()
print "  EqtIPv6Count             : %s" % res.EqtIPv6Count()
print "  EqtIPv6MulticastCount    : %s" % res.EqtIPv6MulticastCount()
print "  EqtIPv6BroadcastCount    : %s" % res.EqtIPv6BroadcastCount()
print "  EqtTCPCount              : %s" % res.EqtTCPCount()
```

```
print "  EqtUDPCount            : %s" % res.EqtUDPCount()
print "  EqtSCTPCount           : %s" % res.EqtSCTPCount()
print "  EqtICMPCount           : %s" % res.EqtICMPCount()
print "  Eqt63OrLessCount       : %s" % res.Eqt63OrLessCount()
print "  Eqt64To127Count        : %s" % res.Eqt64To127Count()
print "  Eqt128To255Count       : %s" % res.Eqt128To255Count()
print "  Eqt256To511Count       : %s" % res.Eqt256To511Count()
print "  Eqt512To1023Count      : %s" % res.Eqt512To1023Count()
print "  Eqt1024To1500Count     : %s" % res.Eqt1024To1500Count()
print "  Eqt1501OrMoreCount     : %s" % res.Eqt1501OrMoreCount()
print "  EqtMisalignedCount     : %s" % res.EqtMisalignedCount()
print "  NetPacketCount         : %s" % res.NetPacketCount()
print "  NetIPv4Count           : %s" % res.NetIPv4Count()
print "  NetIPv4MulticastCount  : %s" % res.NetIPv4MulticastCount()
print "  NetIPv4BroadcastCount  : %s" % res.NetIPv4BroadcastCount()
print "  NetIPv6Count           : %s" % res.NetIPv6Count()
print "  NetIPv6MulticastCount  : %s" % res.NetIPv6MulticastCount()
print "  NetIPv6BroadcastCount  : %s" % res.NetIPv6BroadcastCount()
print "  NetTCPCount            : %s" % res.NetTCPCount()
print "  NetUDPCount            : %s" % res.NetUDPCount()
print "  NetSCTPCount           : %s" % res.NetSCTPCount()
print "  NetICMPCount           : %s" % res.NetICMPCount()
print "  Net63OrLessCount       : %s" % res.Net63OrLessCount()
print "  Net64To127Count        : %s" % res.Net64To127Count()
print "  Net128To255Count       : %s" % res.Net128To255Count()
print "  Net256To511Count       : %s" % res.Net256To511Count()
print "  Net512To1023Count      : %s" % res.Net512To1023Count()
print "  Net1024To1500Count     : %s" % res.Net1024To1500Count()
print "  Net1501OrMoreCount     : %s" % res.Net1501OrMoreCount()
```

```
print "  NetMisalignedCount      : %s" % res.NetMisalignedCount()

print "  EqtFilterPacketCount    : %s" % res.EqtFilterPacketCount(0)

print "  EqtFilterByteCount      : %s" % res.EqtFilterByteCount(0)

print "  EqtFilterByteCountInvalid: %s" % res.EqtFilterByteCountInvalid(0)

print "  NetFilterPacketCount    : %s" % res.NetFilterPacketCount(0)

print "  NetFilterByteCount      : %s" % res.NetFilterByteCount(0)

print "  NetFilterByteCountInvalid: %s" % res.NetFilterByteCountInvalid(0)
```

Methods

Each of the MetricsResult class methods is described below.

`IsPRETimeSync()`

Indicates whether the PRE is time synced with the wall clock.

Returns true if the PRE is time sync with the wall clock.

The PRE can be configured to time sync with the wall clock using XXX (will look up the HW card name). This feature may be turned on or off.

When the feature is turned off, or when the most recent check indicates that the PRE is not time synced with the wall clock, then this function returns false.

`PRETimeSyncLossCount()`

Indicates the number of times the PRE has lost time sync with the wall clock.

Returns an unsigned integer indicating the number of times the PRE has lost time sync with the wall clock since the PRE has been running.

If this function returns 0, and the IsPRETimeSync() function returns false, then the PRE is not configured to time sync with the wall clock.

`Version()`

Description: Returns the version of the object.

Parameters: None

Return Value: The object version.

Remarks: This version number identifies the filter result packet format version associated with this object. It is not related to the PacketAccess Library version.

`ProbeId()`

Description: Returns the ID of the SFProbe that captures the original packet.

Parameters: None

Return Value: StringVector value of the probe Id

Remarks: A probe ID is not null-terminated. Applications should use PAString's length function to return the length of the probe ID string. See the example above.

`Seconds()`

Description: Returns the "Seconds" portion of the timestamp. This value is the number of seconds since January 01, 1970 00:00:00.

Parameters: None

Return Value: Returns the seconds portion of the timestamp.

Remarks: MetricsResults are returned to the application in a first-in, first-out manner. The application may receive a MetricsResult with an earlier timestamp than the previous MetricsResult it receives. This is very common if there are multiple probes in different parts of the network sending MetricsResults to the same MetricsResultAccess object, or if the probes are not time synchronized with the PRE.

`NSeconds()`

Description: Returns the "nanoseconds" portion of the timestamp.

Parameters: None

Return Value: Returns the "nanoseconds" portion of the timestamp.

`Sequence()`

Description: Returns an unsigned 16-bit value that represents the sequence number of the result.

Parameters: None

Return Value: Returns unsigned 16-bit value that represents the sequence number of the result.

Remarks: For a given SFProbe, an application can use the sequence number to determine if a MetricsResult is lost. A MetricsResult can be lost in transit, or due to buffer overflow.

A gap in the sequence number indicates that an intended MetricsResult for that probe was not delivered. In most cases, an application can safely ignore this situation.

There are two situations an application may want to re-baseline its counters if there is a skipped MetricsResult:

1. If an application is interested in the filter byte counters. The filter byte counter may become invalid if a jumbo packet (more than about 2000 bytes) has been filtered. In that case, the filter byte counter invalid flag for that filter slot will be set. This flag is cleared for each Metrics Result request. If there is a lost Metrics Result, the application may not be aware that the filter byte counter invalid flag has been reset.

2. Under rare situations, if there are too many missed sequence numbers, then the counters may rollover more than once. Different counters rollover at different rates, depending on the counter's capacity and network traffic volume,. The application should decide when the number of missed sequences may cause a double rollover.

   For example, the theoretical maximum number of Ethernet frames per second on a 1G network is around 1.4 million frames per second, and the 29-bit total packet count can count up to around 536 million packets. So packet counter may rollover around every 6 minutes. If the metrics result request interval (configurable through System Manager) is every 5 minute, then missing two consecutive Metrics Results may cause a double rollover.

Example:

```
pa = PacketAccess.CreateMetricsResultAccess()

    # set source type and properties

    # ...
```

```
if (not pa.Start()):
   print "Error %s" % (pa.LastError())
   PacketAccess.DeleteMetricsResultAccess(pa)
   sys.exit(-1)

maxSeq = 2 ^ 16 - 1
newSeq = True
timeout = 1000  # 1 second timeout
# assume all the metrics results come from the same probe
while (True):
   res = pa.Get(timeout)
   if (res == None):
      break
   currSeq = res.Sequence()
   if (newSeq or (lastSeq + 1 == currSeq) or \
      (lastSeq == maxSeq and currSeq == 0)):
      # first metrics result, or subsequent sequence
      lastSeq = currSeq
      newSeq = False
   else:
      # one of more MetricsResult is lost
      print "last sequence is %s" % (lastSeq)
      print "current sequence is %s" % (currSeq)

   PacketAccess.DeleteMetricsResult(res)

pa.Stop()
PacketAccess.DeleteMetricsResultAccess(pa)
```

`ResetCount()`

Description: Returns a value that represents how many times the application should treat this metrics result as a new baseline. Returns a 16 bit integer.

`RetryCount()`

Description: Returns a value that represents how many times the PRE has to retransmit the metrics result request to the SFProbe. Returns a 16 bit integer. This value may be important to applications that want to determine if the missing sequence number is due to the PRE unable to transmit or receive metrics results to and from the SFProbe.

`SFFTemperature()`

Description: Temperature of the SFProbe. Returns a 16 bit signed integer in increments of 1/256 °C.

`SFFVcc()`

Description: Supply voltage of the SFProbe. Returns a 16 bit unsigned integer in increments of 100 μV.

`SFFTxBias()`

Description: Laser bias current of the SFProbe. Returns a 16 bit unsigned integer in increments of 2 μV.

`SFFTxPower()`

Description: Transmitted average optical power of the SFProbe. Returns a 16 bit unsigned integer in increments of 0.1 μW.

`SFFRxPower()`

Description: Received average optical power of the SFProbe. Returns a 16 bit unsigned integer in increments of 0.1 μW.

`M2SAverageNSecond()`

Description: The average time needed for a packet to travel from PRE to SFProbe. Represents the average latency from the PRE to the SFProbe. Returns a 32-bit integer.

`S2MAverageNSecond()`

Description: The average time needed for a packet to travel from SFProbe to PRE. Represents the average latency from the SFProbe to the PRE. Returns a 32-bit integer.

`TimingOffset()`

Description: M2SAverageNSecond minus the average of M2SAverageNSecond and S2MAverageNSecond. M2S - (M2S + S2M) / 2. Returns a 32-bit integer that represents the average round trip latency between the PRE and the SFProbe.

`IsTimingValid()`

Description: Indicates whether the IsTimingLock return value is valid. Returns 1 for true and 0 for false.

`IsTimingLock()`

Description: Indicates whether the SFProbe is in time synchronization with the PRE at the time this packet is generated. Returns 1 for true and 0 for false.

`EqtByteCount()`

Description: Total number of bytes on the EQT side. Returns a 48-bit unsigned integer representing the total number of bytes on the EQT side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

NetByteCount()

Description: Total number of bytes on the NET side. Returns a 48-bit unsigned integer representing the total number of bytes on the NET side.

Remarks: This counter counts all the bytes from the first byte to the last byte of the Ethernet frame, including the FCS.

When a packet has an odd number of bytes in the Ethernet frame, this counter may undercount by one. This under-counting is not cumulative. Therefore the actual number of bytes of all Ethernet frames may be at most one more than this counter.

EqtPacketsFiltered()

Description: Total number of packets filtered on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered on the EQT side.

Remarks:

EqtPacketsInjected()

Description: Total number of packets injected by the SFProbe on the EQT side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the EQT side.

NetPacketsFiltered()

Description: Total number of packets filtered by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets filtered by the SFProbe on the NET side.

Remarks:

`NetPacketsInjected()`

Description: Total number of packets injected by the SFProbe on the NET side.

Parameters: None

Return Value: A 32-bit unsigned integer representing the total number of packets injected by the SFProbe on the NET side.

Remarks:

`EqtPacketCount()`

Description: Number of packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks:

`EqtIPv4Count()`

Description: Number of IPv4 packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if an IPv4 header is detected in a packet header. If there are two IPv4 headers in the packet header, this counter is still only incremented once.

`EqtIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has a most significant nibble of the first byte has the bit pattern of “1110” (0xE) in its IPv4 destination address.

For example, the following IP destination addresses will cause this counter to be incremented: 224.0.0.1, 233.252.1.32.

`EqtIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has the IPv4 destination address of 255.255.255.255.

`EqtIPv6Count()`

Description: Number of IPv6 packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if an IPv6 header is detected in a packet header. If there are two IPv6 headers in the packet header, this counter is still only incremented once.

`EqtIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.

- The last 2 bytes are not equal to 0x0001.
- The third to twelfth bytes are not all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: `FF3X::4000:0`

`EqtIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the EQT side.

Parameters: None

Return Value: 29-bit

Remarks: This counter is incremented by the SFProbe if a packet on the EQT side has an IPv6 destination address that meets all of the following criteria:

- The first byte of the address is 0xFF.
- The last 2 bytes are equal to 0x0001.
- The third to twelfth bytes are all zeros.

For example, the following IPv6 destination addresses will cause this counter to be incremented: FF02:0:0:0:0:0:0:1

`EqtTCPCount() const`

Description: Number of TCP packets on the EQT side.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of TCP packets on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the TCP header.

`EqtUDPCount() const`

Description: Number of UDP packets on the EQT side.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of UDP packets on the

EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the UDP header.

`EqtSCTPCount()`

Description: Number of SCTP packets on the EQT side.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of SCTP packets on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the SCTP header.

`EqtICMPCount()`

Description: Number of ICMP packets on the EQT side.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of ICMP packets on the EQT side.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side contains the ICMP header.

`Eqt63OrLessCount()`

Description: Number of packets on the EQT side that have less than 64 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that have less than 64 bytes.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is less than 64 bytes.

`Eqt64To127Count()`

Description: Number of packets on the EQT side that are between 64 and 127 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are between 64 and 127 bytes.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 64 and 127 bytes.

`Eqt128To255Count()`

Description: Number of packets on the EQT side that are between 128 and 255 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are between 128 and 255 bytes.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 128 and 255 bytes.

`Eqt256To511Count()`

Description: Number of packets on the EQT side that are between 256 and 511 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are between 256 and 511 bytes

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 256 and 511 bytes.

`Eqt512To1023Count()`

Description: Number of packets on the EQT side that are between 512 and 1023 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are between 512 and 1023 bytes

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 512 and 1023 bytes.

`Eqt1024To1500Count()`

Description: Number of packets on the EQT side that are between 1024 and 1500 bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are between 1024 and 1500 bytes

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is between 1024 and 1500 bytes.

`Eqt1501OrMoreCount()`

Description: Number of packets on the EQT side that are 1501 or more bytes.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of packets on the EQT side that are 1501 bytes or more.

Remarks: This counter is incremented by the SFProbe if a packet header on the EQT side is 1501 or more bytes.

`EqtMisalignedCount()`

Description: Number of packets that are misaligned on the EQT side.

Parameters: None

Return Value: Returns a 29-bit value representing the total number of misaligned packets on the EQT side.

Remarks: None

`NetPacketCount()`

Description: Number of packets on the NET side. Returns 29-bit value.

`NetIPv4Count()`

Description: Number of IPv4 packets on the NET side. Returns 29-bit value.

`NetIPv4MulticastCount()`

Description: Number of IPv4 multicast packets on the NET side. Returns 29-bit value.

`NetIPv4BroadcastCount()`

Description: Number of IPv4 broadcast packets on the NET side. Returns 29-bit value.

`NetIPv6Count()`

Description: Number of IPv6 packets on the NET side. Returns 29-bit value.

`NetIPv6MulticastCount()`

Description: Number of IPv6 multicast packets on the NET side. Returns 29-bit value.

`NetIPv6BroadcastCount()`

Description: Number of IPv6 broadcast packets on the NET side. Returns 29-bit value.

`NetTCPCount()`

Description: Number of TCP packets on the NET side. Returns 29-bit value.

`NetUDPCount()`

Description: Number of UDP packets on the NET side. Returns 29-bit value.

`NetSCTPCount()`

Description: Number of SCTP packets on the NET side. Returns 29-bit value.

`NetICMPCount()`

Description: Number of ICMP packets on the NET side. Returns 29-bit value.

`Net63OrLessCount()`

Description: Number of packets on the NET side that have less than 64 bytes. Returns 29-bit value.

`Net64To127Count()`

Description: Number of packets on the NET side that are between than 64 and 127 bytes. Returns 29-bit value.

`Net128To255Count()`

Description: Number of packets on the NET side that are between than 128 and 255 bytes. Returns 29-bit value.

`Net256To511Count()`

Description: Number of packets on the NET side that are between than 256 and 511 bytes. Returns 29-bit value.

`Net512To1023Count()`

Description: Number of packets on the NET side that are between than 512 and 1023 bytes. Returns 29-bit value.

`Net1024To1500Count()`

Description: Number of packets on the NET side that are between than 1024 and 1500 bytes. Returns 29-bit value.

`Net1501OrMoreCount()`

Description: Number of packets on the NET side that are 1501 or more bytes. Returns 29-bit value.

`NetMisalignedCount()`

Description: Number of packets that are misaligned on the EQT side.

Return Value: 29-bit

`EqtFilterPacketCount(index)`

Description: Returns the number of filtered packet for filter slot indicated by “index”.

Parameters: index

A number between 0 and 15.

Return Value: 29-bit

`EqtFilterByteCount(index)`

Description: Returns the number of filtered bytes for filter slot indicated by “index”.

Parameters: index

A number between 0 and 15.

Return Value: 36-bit

Remarks: This counter may not be valid if the EqtFilterByteCountInvalid of the same filter slot returns true.

`EqtFilterByteCountInvalid(index)`

Description: Returns whether the EqtFilterByteCount of the same filter slot has a valid value.

Parameters: index

A number between 0 and 7.

Return Value: Return 1 if the EqtFilterByteCount of the same filter slot has a valid value, 0 otherwise.

Remarks: If a filtered packet is a jumbo packet (more than around 2000 bytes), then the filter byte counter of that filter slot will undercount the number of bytes. This flag is reset for every MetricsResult generated by the SFProbe.

`NetFilterPacketCount(index)`

Description: Number of filtered packets on the NET side for a filter slot.

Parameters: index

A number between 0 and 7.

Return Value: 29-bit

Remarks: None

`NetFilterByteCount(index)`

Description: Number of filtered bytes on the NET side for a filter slot.

Parameters: index

A number between 0 and 7.

Return Value: 36-bit

Remarks: None

`NetFilterByteCountInvalid(index)`

Description: Indicates whether the filtered byte count on the NET side is valid for a filter slot.

Parameters: index

A number between 0 and 7.

Return Value: Returns 1 if the NetFilterByteCount of the same filter slot has a valid value, 0 otherwise.

`MetricsDataLength()`

Description: Return the size of the MetricsData object

Parameters: None

Return Value: Return the number of bytes needed to store a MetricsResult object

Remarks: In some cases, an application may want to store the entire MetricsResult object away for analysis at a later time. Application can allocate a buffer of size returned by the MetricsDataLength function.
Note: MetricsResult object size is the same for the same MetricsResult object version.

`MetricsData()`

Description: Copies the content of the MetricsResult object into a byte array.

Parameters: None

Return Value: Returns an array of bytes representing the MRP packet.

Remarks:

Python

PacketSourceTypeInfo

PacketSourceTypeInfo provides information on a packet source supported by the PacketPortal Library.

Functions

`PacketSourceTypeInfo.`**`Name()`**

Return the name of the packet source.

Note: The name of the packet source can be passed to PacketResultAccess's SetSourceType method. This value is not case-sensitive.

`PacketSourceTypeInfo.`**`Description()`**

Return the description of the packet source.

`PacketSourceTypeInfo.`**`HelpText()`**

Return more information on packet source properties.

Python

PacketSourceTypeInfoList

PacketSourceTypeInfoList contains a collection of PacketSourceTypeInfo. The primary purpose of this object is to pass packet source information between the application and the PacketAccess Library. The application can obtain an instance of this object by calling PacketAccess.CreatePacketSourceTypeInfoList method.

Functions

`PacketSourceTypeInfoList.`**`Size()`**

Return the number of objects in the collection.

`PacketSourceTypeInfoList.`**`Get(`***`index`***`)`**

Return a PacketSourceTypeInfo object by its position in the collection.

Note: If index is valid, then return a PacketSourceTypeInfo object. Otherwise, return None.

Python

PacketResultAccess

The PacketResultsAccess class provides the base implementation for accessing packets generated by the PacketPortal system.

Functions

`PacketResultAccess.`**`SetSourceType(`***`sourceType`***`)`**

Set the packet source to one of the following: TCP, UDP, Libpcap, and File. This parameter is not case sensitive. Return True if the sourceType parameter specified a supported source. Otherwise returns False. Call LastError for extended information.

If this method is called after Start, then the running instance is stopped before the sourceType is applied. All counter information is lost.

Note: Additional detailed information for each parameter is shown in the sections that immediately follow this section.

Additional Parameter Information (Describing UDP, TCP, File, and Libpcap parameters in detail):

UDP retrieves PacketPortal packets using the UDP protocol. Use the UDP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using UDP.   Since UDP provides unreliable data service, there may be packet loss between the PRE and the PacketAccess API application.

Property Name	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.SetSourceType("udp")):

   print "Error create filter access source: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)

   sys.exit(-1)


if (not pa.SetSourceProperty("port", port)):

   print "Error setting port: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)

   sys.exit(-1)
```

TCP retrieves PacketPortal packets using TCP protocol. Use the TCP source when the PacketPortal system is setup to send filter results packets from the PRE to the PacketAccess API application using TCP. Since TCP provides a reliable data service, there may be minimal packet loss between the PRE and the PacketAccess API application. However, TCP adds some overhead and may cause more delays in the PacketAccess API.

Property Name:	Description	Type	Allow Multiple?	Defaults
Port:	Monitor UDP/TCP port. This property may be set before or after Start, and it will take effect immediately if the port is successfully opened.	Positive integer 1 - 65535	Yes	None
RemovePort:	Stop monitoring a port.	Positive integer 1 - 65535	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default
MaxConnections:	The maximum number of pending TCP connections. This value is passed to the system call listen, and is subject to the limit set by the operating system.	Positive integer	No	64

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.SetSourceType("tcp")):

   print "Error create filter access source: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)

   sys.exit(-1)


if (not pa.SetSourceProperty("SsocketBufferSize", 1024 * 1024)):

   print "Error setting port: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)

   sys.exit(-1)


if (not pa.SetSourceProperty("port", port)):

   print "Error setting port: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)
```

```
    sys.exit(-1)
```

File retrieves PacketPortal packets from a PCAP capture file. The File source can be used for post processing of captured filter results packets, or used in the emulation mode with a regular PCAP file. TimeBreakpoint is ignored when using the File source. Inter-packet gap is also ignored, as the FilterResultAccess object returns the filter results (or emulated filter results) to the application as fast as it can read and sequence the packets.

Property Name	Description	Type	Allow Multiple?	Defaults
FileName:	Use packets from this PCAP file. Must set this property before Start. If file does not exist or user does not have sufficient permission to read it, then Start returns false. Setting of this property after a Start will be ignored until the next Start.	Pointer to a null-terminated char array	No	None
Loop:	The number of times the file is looped.	Integer. 0 means loop forever	No	1
IdleTime:	Idle this many milliseconds every "idleInterval" number of packets.	Positive integer	No	0
IdleInterval:	Idle the number milliseconds specified by "IdleTime" every this many number of packets.	Positive integer	No	100

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.SetSourceType("file")):

    print "Error create filter access source: %s" % (pa.LastError())

    PacketAccess.DeleteFilterResultAccess(pa)

    sys.exit(-1)


if (not pa.SetSourceProperty("fileName", "test.pcap")):

    print "Error setting port: %s" % (pa.LastError())

    PacketAccess.DeleteFilterResultAccess(pa)
```

```
    sys.exit(-1)
```

Libpcap retrieves PacketPortal packets from an Ethernet device in promiscuous mode. When the PacketPortal system is configured to send the filter results packets to the PacketAccess API using UDP, the application can choose to use the "libpcap" mode instead of the TCP mode. One advantage of using the libpcap source instead of the UDP source is that it can limit receiving packets by a device; the libpcap source also allows the application to receive filter results packets from any UDP port.

Property Name	Description	Type	Allow Multiple?	Defaults
Device:	Monitor this device (network interface name). This property may be set before or after Start, and it will take effect immediately if the device is successfully opened.	Pointer to a null-terminated char array	Yes	None
RemoveDevice:	Stop monitoring this device.	Pointer to a null-terminated char array	Yes	None
SocketBufferSize:	The maximum size of a socket receive buffer. This value is passed to the system call setsockopt for all open sockets.	Positive integer 1 to MAX_INT	No	Use system default

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.SetSourceType("libpcap")):

    print "Error create filter access source: %s" % (pa.LastError())

    PacketAccess.DeleteFilterResultAccess(pa)

    sys.exit(-1)


if (not pa.SetSourceProperty("device", "eth1")):

    print "Error setting port: %s" % (pa.LastError())

    PacketAccess.DeleteFilterResultAccess(pa)
```

```
    sys.exit(-1)
```

PacketResultAccess.**SetSourceProperty(***name, value***)**

Set or add a value to a source property. Return False when it can be immediately detected that the property value cannot be set successfully; otherwise returns True.

Note: The application should call SetSourceType to set a packet source before calling SetSourceProperty. Setting a new source type will erase all the source property values associated with the previous source type.

If the value of the property name is a numeric type, this method will convert the value string to the numeric value automatically.

PacketResultAccess.**GetSourceType()**

Return the currently specified source type.

Note: Return an empty string if the source is unspecified.

PacketResultAccess.**GetSourcePropertyNames(***names***)**

Retrieve all the valid property names of the packet source associated with the object. All the property names for the packet source are stored to the StringVector specified by “names”.

Note: The object should have a valid packet source before calling GetSourcePropertyNames.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

pa.SetSourceType("UDP")


# list all the FilterResultAccess object's properties

names = PacketAccess.StringVector()
```

```
pa.GetSourcePropertyNames(names)

for i in range(0, names.size()):

   name = names.__getitem__(i)

   values = PacketAccess.StringVector()

   pa.GetSourceProperties(name, values)

   for j in range (0, values.size()):

      value = values.__getitem__(j)

      print "%s: %s" % (name, value)
```

PacketResultAccess.**GetSourceProperty(***name***)**

Retrieve the first value associated with the property name.

Note: If there is no value associated with this property, GetSourceProperty returns an empty string. If there is more than one value associated with this property, then the first value is returned.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

pa.SetSourceType("UDP")


# Check if the object has a port associated with it

port = pa.GetSourceProperty("port")

if (len(port) == 0):

   print "No port is associated with the object"
```

PacketResultAccess.**GetSourceProperties(***name, values***)**

Retrieve all the values associated with the property name and store all the values associated with the property to StringVector specified by “values”.

PacketResultAccess.**Emulate(*b*)**

Turn on emulation mode when b is True, otherwise, turn off.

PacketResultAccess.**Emulate()**

Return True if emulation is on, otherwise return False.

PacketResultAccess.**LastError()**

Return the last error.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.Start()):

   print "Error starting filter access: %s" % (pa.LastError())

   pa.ClearError()
```

PacketResultAccess.**ClearError()**

Clear the last error.

Note: See example in LastError.

PacketResultAccess.**BufferSize()**

Specifies the maximum number of objects stored in the internal buffer.

Note: Application should adjust the buffer size based on memory available for use with the API, how fast the PacketPortal packets are arriving and if there are high latencies among PacketPortal packets routed from multiple SFProbes.

The memory usage is roughly equal to (size * 2000) + (N * 2000) where N = number of actual objects in the buffer.

`PacketResultAccess.`**`BufferSize()`**

Return the current setting of the buffer size.

Python

FilterResultAccess

The FilterResultAccess class retrieves FilterResult objects.

Functions

`FilterResultAccess.`**`Start()`**

Start processing filter result packets. If the function succeeds, the return value is True. If the function fails, the return value is False. Call LastError to get extended error information.

Note: An application gets a FilterResultAccess object by using PacketAccess.CreateFilterResultAccess. After setting the appropriate source properties, the application typically calls Start. All counters are reset to zero at start. The application can then call Get to retrieve available filter results. When the application decides to stop processing filter results, it should call Stop and then PacketAccess.DeleteFilterResultAccess to free up resources used by FilterResultAccess.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

if (not pa.SetSourceType("udp")):

   print "Error create filter access source: %s" % (pa.LastError())

   PacketAccess.DeleteFilterResultAccess(pa)

   sys.exit(-1)
```

```
if (not pa.SetSourceProperty("port", 5000)):
   print "Error setting port: %s" % (pa.LastError())
   PacketAccess.DeleteFilterResultAccess(pa)
   sys.exit(-1)
if (not pa.Start()):
   print "Error starting filter access: %s" % (pa.LastError())
   PacketAccess.DeleteFilterResultAccess(pa)
   sys.exit(-1)
else:
   print "Listening to port %s" % pa.GetSourceProperty("port")
timeout = 1000  # timeout is 1 second
while (True):
   res = pa.Get(1000)
   if (res == None):
      break
   # handle FilterResult
   # ...
   PacketAccess.DeleteFilterResult(res)
pa.Stop()
PacketAccess.DeleteFilterResultAccess(pa)
```

FilterResultAccess.**Stop()**

Stop processing filter result packets.

Note: This function has no parameters.

`FilterResultAccess.`**`LoadSettings(`*`s`*`)`**

Configure the object according to the settings string. If the function succeeds, the return value is True. If the function fails, the return value is False. Call LastError to get extended error information.

Example:

```
pa1 = PacketAccess.CreateFilterResultAccess()

pa1.SetSourceType("file")

pa1.SetSourceProperty("fileName", "test.pcap")

pa1.SetSourceProperty("loop", 2)

settings = pa1.SaveSettings()

PacketAccess.DeleteFilterResultAccess(pa1)


# ...

pa2 = PacketAccess.CreateFilterResultAccess()

if (not pa2.LoadSettings(settings)):

   print "Error loading settings"


# pa2 now contains the same settings as pa1

if (not pa2.Start()):

   print "Error %s" % pa2.LastError()

else:

   # get Filter Result, etc.

   # ...

   pa2.Stop()

PacketAccess.DeleteFilterResultAccess(pa2)
```

`FilterResultAccess.`**`SaveSettings()`**

Return the current settings as a string.

Note: Application should treat the returned string as an opaque value and should not alter it. See **LoadSettings** example provided earlier in this section.

`FilterResultAccess.`**`Sequencing(`***`b`***`)`**

Sets sequencing on or off. When sequencing is turned on, the FilterResultAccess object will return filter results to the application according to a set of sequencing rules. If sequencing is turned off, filter results are made available to the application immediately on a first-in, first-out basis.

Note: An application using the FilterResultAccess object with sequencing turned on is subject to the following rules for sequencing:

- The timestamp for a FilterResult will be the same or later than the previous FilterResult provided to the application.
- Each FilterResult will be held for a minimum of the specified MinBufferTime before it is available to the application.
- Each FilterResult will be held for a maximum of the specified MaxBufferTime before it is available to the application.
- FilterResults of the same SFProbe are ordered by sequence numbers.
- FilterResults of different SFProbes are ordered by timestamps.
- For FilterResults of the same SFProbe, if a FilterResult with an earlier sequence has a later timestamp than another FilterResult, then the FilterResult with the later sequence will have its timestamp adjusted to be a later time than the FilterResult with an earlier sequence.
- For FilterResults from different SFProbes and have the same timestamp, a FilterResult that arrived earlier is provided to the application before a FilterResult that arrived later.

See **Start** example provided earlier in this section.

`FilterResultAccess.`**`Sequencing()`**

If the sequence setting is on, then return True, otherwise return False.

`FilterResultAccess.`**`DiscardLate(`***`b`***`)`**

Sets to True for the application to discard Filter Results that are received late. A filter result is considered “late” if the application has already retrieved a filter result with a more recent timestamp.

Note: DiscardLate takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`FilterResultAccess.`**`DiscardLate()`**

Return True if the DiscardLate setting is on, otherwise return False.

`FilterResultAccess.`**`MinBufferTime(`***`millisecond`***`)`**

Specifies the time period (the minimum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This allows filter results from multiple SFProbes with different latencies to be time ordered. The MinBufferTime should typically be set to the maximum expected delta in the latency among feeds.

Note: MinBufferTime takes effect immediately if FilterResultAccess has already started. If sequencing is turned off, this parameter is ignored.

`FilterResultAccess.`**`MinBufferTime()`**

Returns the minimum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`FilterResultAccess.`**`MaxBufferTime(`***`millisecond`***`)`**

Specifies the time period (the maximum number of milliseconds) that a filter result is kept in the FilterResultAccess buffer before it is made available to the application. This is used when there are sequence number gaps in the Filter Results and the application is allowing extra time for the missing Filter Results to arrive.

Note: If sequencing is turned off, this parameter is ignored for sequencing purposes, but is used to determine when an unmatched truncated or fragmented filter result packet will be discarded from the buffer.

`FilterResultAccess.`**`MaxBufferTime()`**

Returns the maximum number of milliseconds that a filter result is kept in the FilterResultAccess buffer.

`FilterResultAccess.`**`SequenceBreakpoint(`**_`breakpoint`_**`)`**

Sets the number of missing sequence numbers before FilterResultAccess treats the filter result as a new feed. A filter result sequence number specifies the order of the original captured packets. If there is a large gap in sequence numbers between two filter results from the same SFProbe, this may indicate that a feed has been stopped and restarted.

Note: An application should set sequence breakpoint to a large number (e.g. the same as the buffer size) if the FilterResultAccess object is expected to capture a feed that does not stop and restart. Conversely, if the application anticipates that the feed often stops and restarts during a running instance of the FilterResultAccess object, then it should set the sequence breakpoint to a relatively small number. If sequencing is turned off, this parameter is ignored.

`FilterResultAccess.`**`SequenceBreakpoint()`**

Returns the sequence breakpoint value.

`FilterResultAccess.`**`TimeBreakpoint(`**_`millisecond`_**`)`**

Sets the time period that the application waits to receive the next Filter Result Packet in the sequence. If a new Filter Result Packet has not arrived within the number of milliseconds specified by TimeBreakpoint value, then any Filter Result Packet that arrives after that will be treated as a new feed. This allows feeds to be stopped and restarted, and be sequenced correctly within the same running instance of FilterResultAccess.

Note: Time breakpoint should be set to shortest expected time delay between stopping a feed and starting a feed. If sequencing is turned off, this parameter is ignored.

`FilterResultAccess.`**`TimeBreakpoint()`**

Returns the time breakpoint value.

`FilterResultAccess.`**`NumProbes()`**

Returns the number of unique SFProbes that the application has retrieved filter results from.

Note: This counter does not count filter results that are in the FilterResultAccess buffer, but not yet retrieved by the application. This number is only valid when sequencing is turned on.

Example:

```
pa = PacketAccess.CreateFilterResultAccess()

# ... set source type and properties

timeout = 1000   # 1 second timeout
while (True):
{
  res = pa.Get(timeout)
  if (res == None):
     break

  PacketAccess.DeleteFilterResult(res);
```

```
}

pa.Stop();

print "Number of probes: %s" % (pa.NumProbes())

PacketAccess.DeleteFilterResultAccess(pa);
```

`FilterResultAccess.`**`LostCount()`**

This number represents the number of missing filter results, by sequence number, for all SFProbes. This number is only valid when sequencing is turned on.

Note: LostCount can be affected by TimeBreakpoint and SequenceBreakpoint values.

For example, filter results from the same probe ID with the following sequence numbers arrive:
Filter Result 1
<5 ms gap>
Filter Result 3
<20 ms gap>
Filter Result 15

Sequence Breakpoint	Time Breakpoint	Lost Count	Description
10	0	1	The filter result with sequence number 2 is considered lost, and the filter result with sequence number 15 is considered the start of a new sequence
20	0	12	The filter result with sequence number 2 and the filter results with sequence numbers 4 through 14 are all considered lost.
20	10	1	The filter result with sequence number 2 is considered lost because filter result 3 arrives less than 10 ms after filter result 1. Filter results with sequence 4 through 14 are not considered lost because filter results 15 arrives more than 10 ms later, even though filter results 15 is less than sequenceBreakpoint away from filter results 3.

Since LostCount counts all the gaps in filter results, it may not reflect the loss of filter results if the loss happens before the first filter results of a particular probe, or if the loss happens after the last filter result was processed by the application.

The following examples illustrate how LostCount and DiscardCount are related.

Scenario 1: Multiple Probes with Late Filter Results

In this scenario, the results from Probe B are all "late" and FilterResultAccess is configured to discard late packets. The following is the filter result packets arrival order (where Probe A results are: A1, A2, A3, and A4 -and- Probe B results are: B1 and B2):

A1, A2, B1, B2, A3, A4

FilterResultAccess will discard B1 and B2 because they are considered late, therefore, the application receives four filter results: A1 - A4.

In this case, LostCount is 0 since there are no gaps in sequence numbers for probe A and DiscardCount is 2.

Scenario 2: Filter Results discarded towards the end of a run

In this scenario, the following packets arrived from probe A: A1, A2, A3, A5, A6, A7, A8, A9, A10.

Assume that A8, A9 and A10 are discarded by FilterResultAccess because of buffer overflow.

In this case, after the application receives A7, the LostCount is 1 (because A4 is missing), and the DiscardCount is 3 (because A8, A9 and A10 are discarded).

`FilterResultAccess.`**`DiscardCount()`**

Returns the number of filter results discarded by the FilterResultAccess object for any reason.

Note: This is valid whether sequencing is turned on or not.

`FilterResultAccess.`**`DiscardDuplicateCount()`**

Returns the number of filter results discarded because the filter result is considered a duplicate.

Note: This is a relatively rare occasion, and usually occurs when the Filter Result Packets are duplicated by the network due to incorrect network configuration.

`FilterResultAccess.`**`DiscardLateCount()`**

Returns the number of filter results discarded because the filter result is considered to be late.

Note: There are several reasons that a filter result can be considered late. For example:

- The SFProbes are not time synchronized with the PRE.
- Multiple PREs are not time synchronized with one another.
- The MinBufferTime value is not set high enough to accommodate the difference in network latencies among the Filter Result Packets.

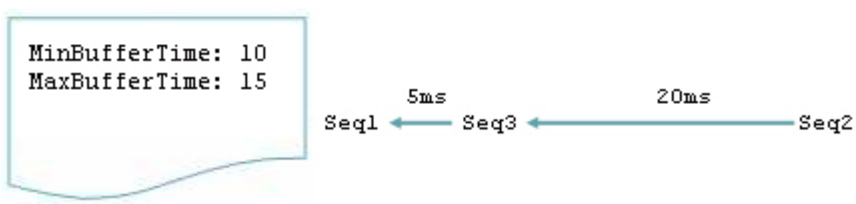
If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

FilterResultAccess.**DiscardOutOfSequenceCount()**

Returns the number of filter results discarded because the filter result is considered out of sequence. A filter result is considered out of sequence if the application is provided with a filter result of the same probe ID and a later sequence number.

Note: This count can be affected by MinBufferTime and MaxBufferTime.

For example, filter results from the same probe ID arrive with the following sequence numbers and time gaps:



T is the time when the filter result with sequence number 1(Seq1) arrives.

T + 0	Seq 1 arrives
T + 5	Seq 3 arrives
T + 10	Seq 1 has been held for MinBufferTime, so it can be provided to application
T + 15	Seq 3 has been held for MinBufferTime, but it will wait 5 more ms to MaxBufferTime because a sequence is missing
T + 20	Seq 3 has been held MaxBufferTime, so it will be provided to

	application
T + 25	Seq 2 arrives
T + 35	Seq 2 is ready for the application, but it is discarded because it has an earlier sequence number than Seq 3

`FilterResultAccess.`**`DiscardOverflowCount()`**

Returns the number of filter results discarded because the FilterResultAccess buffer is too full. Filter results are not inserted in the buffer once the number of filter results in the buffer reaches the BufferSize value.

Note: Changing the BufferSize value can affect the number of filter results that are discarded.

`FilterResultAccess.`**`InputFRPCount()`**

Returns the number of filter result packets retrieved from the FilterResultAccess source. Only packets that appear to contain a legitimate FilterResults header is counted.

Note: This count is valid whether sequencing is turned on or not.

`FilterResultAccess.`**`DiscardFRPCount()`**

Returns the number of Filter Result Packets discarded by the FilterResultAccess object for any reason.

Note: This count is valid whether sequencing is turned on or not.

`FilterResultAccess.`**`DiscardDuplicateFRPCount()`**

Returns the number of Filter Result Packets discarded by the FilterResultAccess object because they are duplicates.

FilterResultAccess.**DiscardFragmentedFRPCount()**

Returns the number of Filter Result Packets discarded by the FilterResultAccess object because the matching filter result packet did not arrive in time to be reassembled.

Note: An unmatched filter result packets is discarded once it is kept in the buffer for a period set in MaxBufferTime. Changing the MaxBufferTime value can affect the number of fragmented Filter Result Packets that are discarded.

FilterResultAccess.**DiscardLateFRPCount()**

Returns the number of filter result packets discarded because the filter result packets are considered to be late.

Note: If DiscardLate is turned off, then filter results will not be discarded even if they are late. Therefore, DiscardLateFRPCount and DiscardLateCount would be zero. Applications can query whether a filter result is late using the FilterResult's Late function.

FilterResultAccess.**DiscardOutOfSequenceFRPCount()**

Returns the number of filter result packets discarded because they are considered to be out of sequence

Note: See DisardOutOfSequenceCount.

FilterResultAccess.**DiscardOverflowFRPCount()**

Returns the number of filter result packets discarded because the FilterResultAccess buffer is too full.

Note: See DisardOverflowCount.

FilterResultAccess.**Get()**

Retrieves the next filter result from the FilterResultAccess buffer that is available at this moment. If no filter result is currently available, then "none" is returned.

Note: When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains filter result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`FilterResultAccess.`**`Get(`**`*timeout*`**`)`**

Retrieves the next filter result from the FilterResultAccess buffer, waiting the specified time for an available filter result. If no filter result is available at the end of the specified time, then "none" is returned.

Note: When the FilterResult object is no longer needed, call PacketAccess.DeleteFilterResult to release resources used by the object.

`FilterResultAccess.`**`NumResultsInBuffer()`**

Retrieves the number of filter results in the buffer.

Python

MetricsResultAccess

The MetricsResultsAccess class retrieves MetricsResult objects.

Functions

`MetricsResultsAccess.`**`Start()`**

Start processing metrics result packets. If the function succeeds, the return value is True. If the function fails, the return value is False. Call LastError to get extended error information.

Note: An application gets a MetricsResultAccess object by using PacketAccess.CreateMetricsResultAccess. After setting the appropriate source properties, the application typically calls Start. The application can then call Get to retrieve available metrics results. When the application decides to stop processing metrics results, it should call Stop and then PacketAccess.DeleteMetricsResultAccess to free up resources used by MetricsResultAccess.

Example:

```
pa = PacketAccess.CreateMetricsResultAccess()

if (not pa.SetSourceType("udp")):

   print "Error create metrics access source: %s" % (pa.LastError())

   PacketAccess.DeleteMetricsResultAccess(pa)

   sys.exit(-1)


if (not pa.SetSourceProperty("port", 5000)):

   print "Error setting port: %s" % (pa.LastError())

   PacketAccess.DeleteMetricsResultAccess(pa)

   sys.exit(-1)

if (not pa.Start()):

   print "Error starting metrics access: %s" % (pa.LastError())

   PacketAccess.DeleteMetricsResultAccess(pa)

   sys.exit(-1)

else:

   print "Listening to port %s" % pa.GetSourceProperty("port")

timeout = 1000  # timeout is 1 second

while (True):

   res = pa.Get(1000)

   if (res == None):

      break

   # handle MetricsResult

   # ...

   PacketAccess.DeleteMetricsResult(res)

pa.Stop()
```

```
PacketAccess.DeleteMetricsResultAccess(pa)
```

MetricsResultsAccess.**Stop()**

Stop processing metrics result packets.

Note: This function has no parameters.

MetricsResultsAccess.**LoadSettings(*s*)**

Configure the object according to the settings string. If the function succeeds, the return value is True. If the function fails, the return value is False. Call LastError to get extended error information.

Example:

```
pa1 = PacketAccess.CreateMetricsResultAccess()

pa1.SetSourceType("file")

pa1.SetSourceProperty("fileName", "test.pcap")

pa1.SetSourceProperty("loop", 2)

settings = pa1.SaveSettings()

PacketAccess.DeleteMetricsResultAccess(pa1)


# ...

pa2 = PacketAccess.CreateMetricsResultAccess()

if (not pa2.LoadSettings(settings)):

   print "Error loading settings"


# pa2 now contains the same settings as pa1

if (not pa2.Start()):

   print "Error %s" % pa2.LastError()

else:
```

```
    # get Filter Result, etc.

    # ...

    pa2.Stop()

PacketAccess.DeleteMetricsResultAccess(pa2)
```

`MetricsResultsAccess.`**`SaveSettings()`**

Return the current settings as a string.

Note: Application should treat the returned string as an opaque value and should not alter it. See **LoadSettings** example provided earlier in this section.

`MetricsResultsAccess.`**`Get()`**

Retrieves the next metrics result from the MetricsResultAccess buffer that is available at this moment. If no metrics result is currently available, then "none" is returned.

Note: When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

When using Get() against a file source, the operation may initially return with no result. The availability of the first packet in the file depends on how long the operating system takes to open the file to retrieve data, or if the file contains metrics result packets. You should call Get() several times until the first packet is retrieved or use the Get(timeout) function to specify a timeout value.

`MetricsResultsAccess.`**`Get(`***`timeout`***`)`**

Retrieves the next metrics result from the MetricsResultAccess buffer, waiting the specified time for an available metrics result. If no metrics result is available at the end of the specified time, then "none" is returned.

Note: When the MetricsResult object is no longer needed, call PacketAccess.DeleteMetricsResult to release resources used by the object.

Python

StringVector

StringVector represents an ordered collection of strings. The primary purpose of this class is to pass a collection of string objects between the application and the PacketPortal library.

Functions

`StringVector.`**`StringVector()`**

Initialize the object to an empty collection.

`StringVector.`**`clear()`**

All the strings in the collection are removed.

Note: The size of the collection is zero after clear is called.

`StringVector.`**`push_back(`**`x`**`)`**

Add a string to the end of the collection.

`StringVector.`**`__getitem__(`**`i`**`)`**

Returns the string at position i

Note: Index positions start at zero. If index is out of range, then s is set to an empty string.

`StringVector.`**`size()`**

Returns the number of strings in the collection.

Chapter 5. Understanding Filter Results and Metrics Results

Understanding Filter Results

The PacketAccess Library provides a programmatic interface for applications to access filter results from the PacketPortal system. Filter result packets contain useful information for network analysis and diagnostics. These packets are generated by the PacketPortal SFProbes, and forwarded to the applications by the Packet Routing Engine (PRE).

Filter Results Packets (FRPs) are sent from the PRE to the PacketAccess library. Then the PacketAccess library parses and assembles the FRPs to produce Filter Results objects. The Filter Results objects can be queried to retrieve the Original Captured Packets (OCP) and their associated metadata.

This document also contains the filter result packet formats. Hardware or applications that do not use the library can directly access these packets.

Filter Result Packet Format

The filter results information is encapsulated in the TCP or UDP payload of a network packet. Filter results information contains the filter results header and the original captured packet. In the following diagram, byte offset 0 indicates the first byte of the TCP or UDP payload of a filter results packet.

<table>
<tr><th>Byte offset</th><th>Byte 0</th><th>Byte 1</th><th>Byte 2</th><th>Byte 3</th><th>Byte 4</th><th>Byte 5</th><th>Byte 6</th><th>Byte 7</th></tr>
<tr><td>0</td><td>0xED</td><td>Flag A</td><td colspan="6">Probe Id (6 bytes)</td></tr>
<tr><td>8</td><td colspan="2">Flag B</td><td colspan="4">Timestamp in seconds</td><td colspan="2">Nanosecond</td></tr>
<tr><td>16</td><td colspan="2">Nanoseconds (cont.)</td><td colspan="4">Reserved</td><td colspan="2">Injected count</td></tr>
<tr><td>24</td><td colspan="2">Injected count (cont.)</td><td colspan="2">Flag C</td><td colspan="4">Filter match bits</td></tr>
<tr><td>32</td><td colspan="4">congestion count</td><td colspan="4">Sequence number</td></tr>
<tr><td>40</td><td colspan="2"># words for fragmented portion (optional, only when F=1 in Flag B)</td><td colspan="6">Original captured payload, variable length</td></tr>
<tr><td></td><td colspan="2">Reserved</td><td colspan="6"></td></tr>
</table>

Flag A (1 byte)

7	6	5	4	3	2	1	0

Reserved, set to 0	Format Version

Format version: Set to 1 for this version.

Flag B (2 bytes)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P	F	T	O	OP	TL	HW major version			HW minor version						

P: When bit is 1, the result packet was on the fiber side.

F: When bit is 1, the result packet is the fragmented part (part 2 of the 2-part packet).

T: When bit is 1, the result packet is truncated (part 1 of the 2-part packet).

O: When bit is 1, the length of the copied payload is one byte less.

OP: When bit is 1, the packet is injected on fiber; 0 = copper.

TL: Timing lock bit. When bit is 1, the 1589 timing servo is locked.

Flag C (2 bytes)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	S	H	E	R	Real packet word length										

B: Bad FCS flag.

S: Set to 1 the filter expression is configured to slice the payload.

H: Set to 1 the filter expression is configured to capture protocol headers only.

E: Set to 1 if the packet was encrypted by the SFProbe.

R: Set to 1 if all filter result packets from this SFProbe are routed to this monitoring port.

Real packet word length: The word length of the original packet, if known.

Sequence Number (4 bytes)

The sequence number is used to sequence filter results packets of the same SFProbe and detect the potential lost filter results packets.

Injected Count

Indicate the number of original filtered packets that the SFProbe has successfully injected for the side indicated by the OP flag. This is a 32-bit counter.

Congestion Count

Indicate the number of original filtered packets that the SFProbe is unable to inject for the side of the currently filtered packet. This is a 29-bit counter.

Original Captured Payload

The original captured payload can be in one or two filter result packets, indicated by the T and F bits in Flag B. The length of the captured payload can be calculated when combined with the O bit in Flag B, and the real packet word length field. The filter result packets carrying the 2 parts of the payload will have the same sequence number and probe ID.

The real packet word length field becomes 0 if the original packet is a jumbo packet (i.e. the packet is greater than around 1996 bytes, depending on the probe setting). The real packet word length may represent a packet length larger than the capture payload length if the captured packet is sliced or headers only. In both cases, the captured payload length will contain an even number of bytes regardless of the odd flag.

The following key describes the variables used in the table below.

X: can be any value.

PL: The packet length in bytes of the payload portion of the FRP (i.e. the TCP or UDP payload length).

PL1: The packet length in bytes of the payload portion of the truncated FRP.

FL: If F = 1, the value in the fragmented payload word length field (bytes 40, 41) times 2.

RPWL: real packet word length.

T	F	O	RWPL	Captured Payload Start	Captured Payload Length
0	0	0	X	Byte 40	PL – 42
0	0	1	0 < (RWPL * 2) <= (PL-42)	Byte 40	PL – 43
0	0	1	RWPL = 0 or (RWPL * 2) > (PL-42)	Byte 40	PL - 42
1	0	X	X	Byte 40	PL – 42
0	1	0	X	Byte 42	FL
0	1	1	0 < (RWPL * 2) <= (FL+PL1-42)	Byte 42	FL – 1
0	1	1	RWPL = 0 or (RWPL * 2) > (FL+PL1-42)	Byte 42	FL

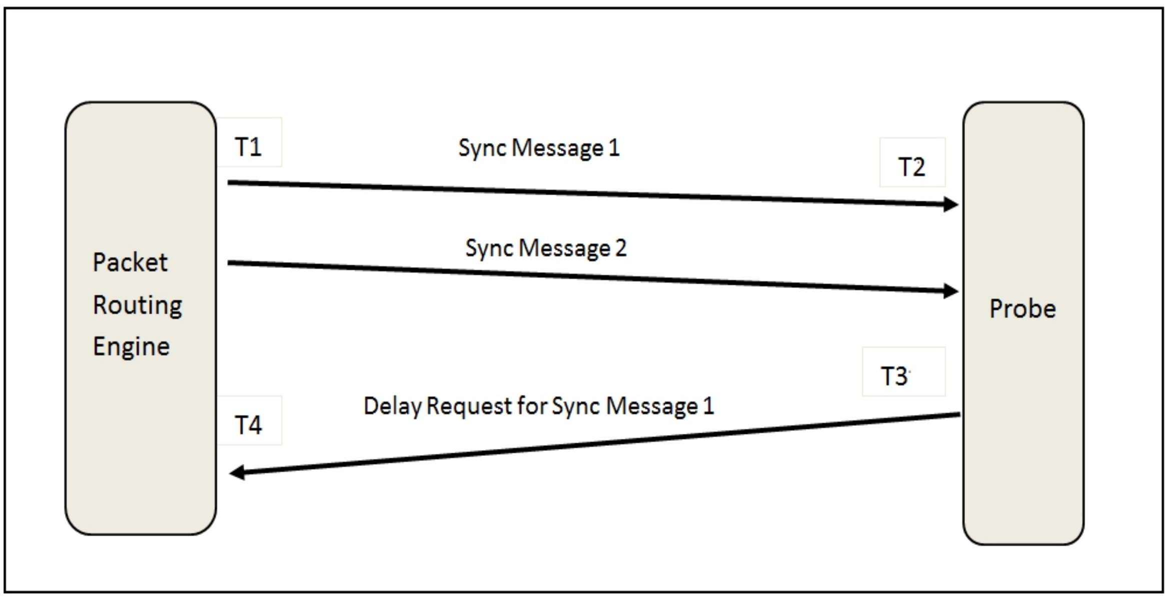
Reserved

Reserved fields are for JDSU internal use only. This field may be set to any value.

Metrics Result Packet Format

The metrics result information is encapsulated in the TCP or UDP payload of a network packet. The payload format is proprietary and subject to change, and should therefore not be parsed directly by applications consuming MRPs. The supported method for accessing MRP information is via the Packet Access API.

MRP Timing Diagram



Sequencing Examples

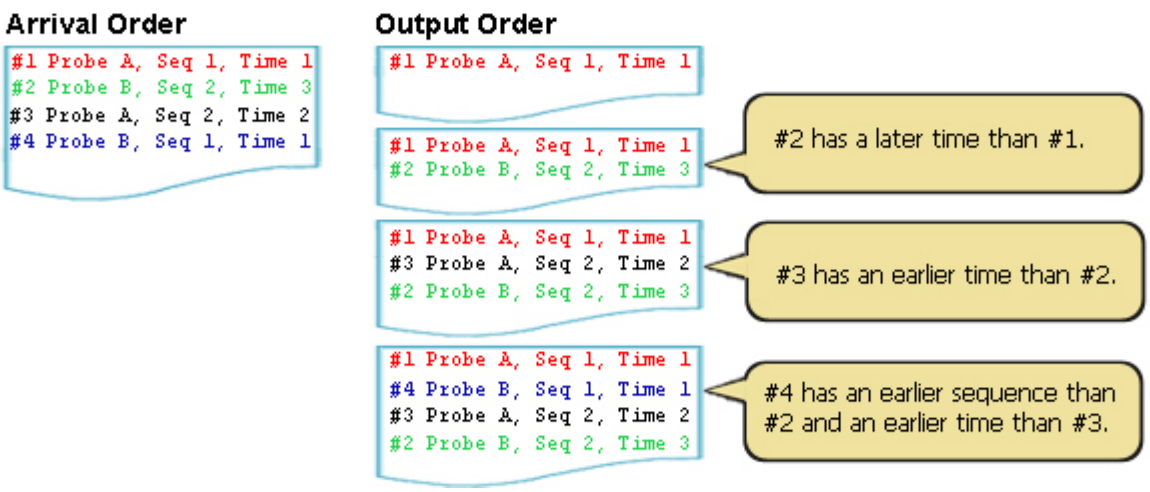
This section provides more information on how FilterResults are ordered. The order depends on whether Sequencing has been enabled or not. Sequencing is an option that can be enabled or disabled through the Sequencing function in the FilterResultAccess class. When Sequencing is enabled, FilterResults are provided to the calling application according to a set of sequencing rules.

The following shows how sequencing is handled. There are five scenarios showing the arrival to the API and the output from the API. Annotation is also provided for each example. The five sequencing scenarios are:

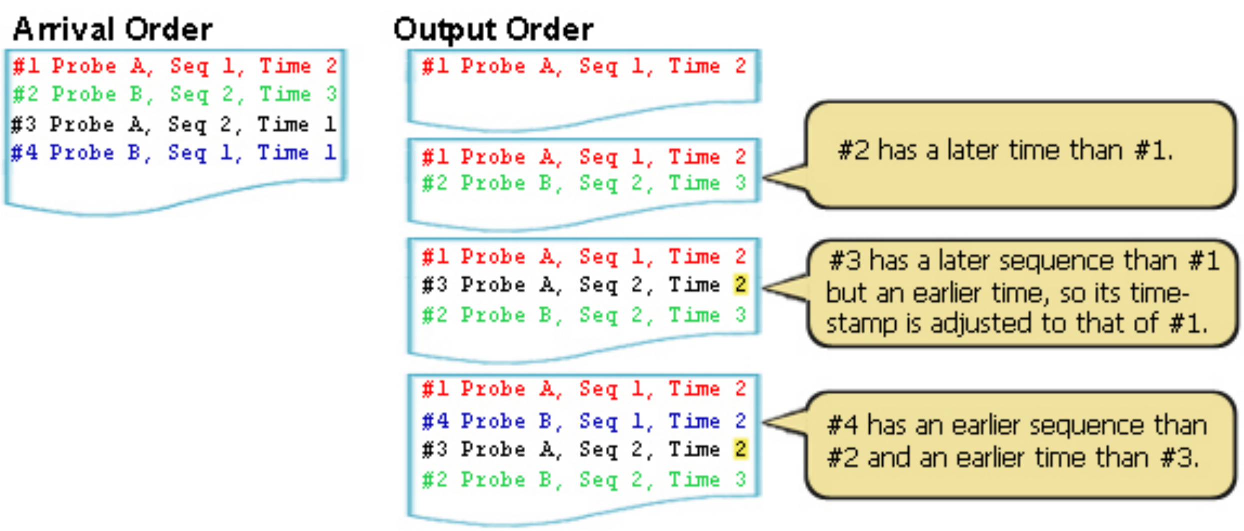
- Multiple SFProbes with no time adjustment
- Multiple SFProbes with time adjusted to later
- Multiple SFProbes with time adjusted to earlier

- Single SFProbe with sequence breakpoint = 10 ms
- Single SFProbe with time breakpoint = 10 ms

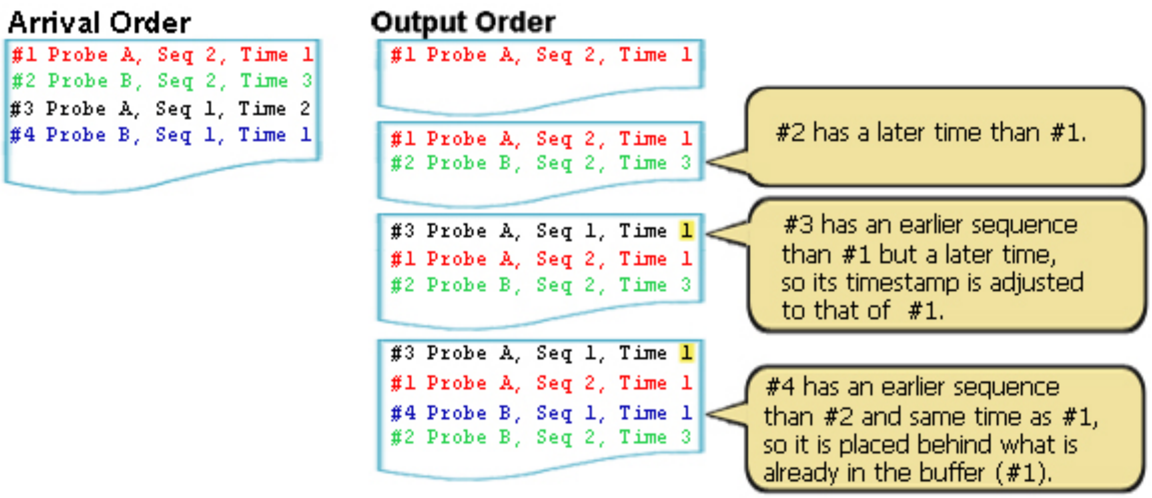
Sequencing Scenario: Multiple SFProbes, no time adjustment



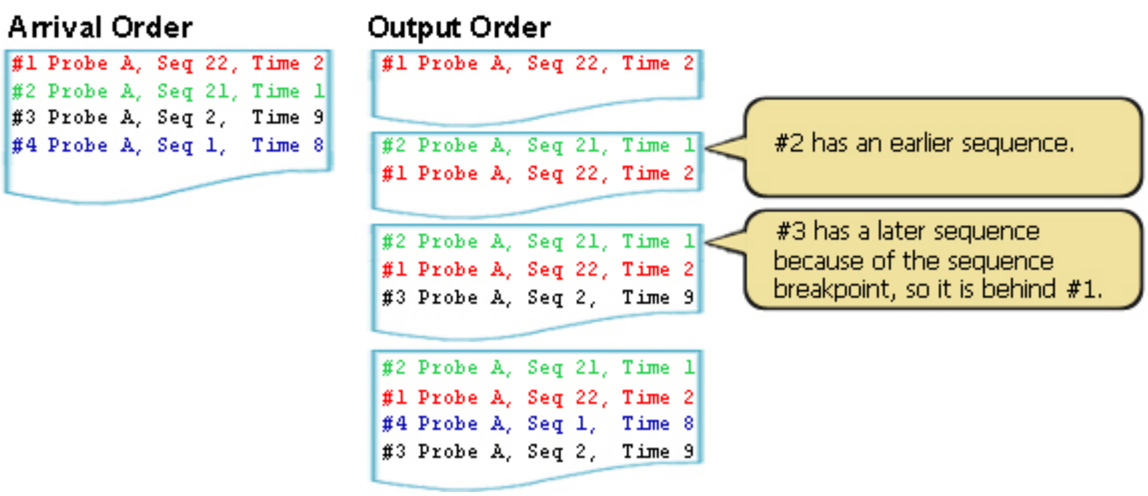
Sequencing Scenario: Multiple SFProbes with time adjusted to later



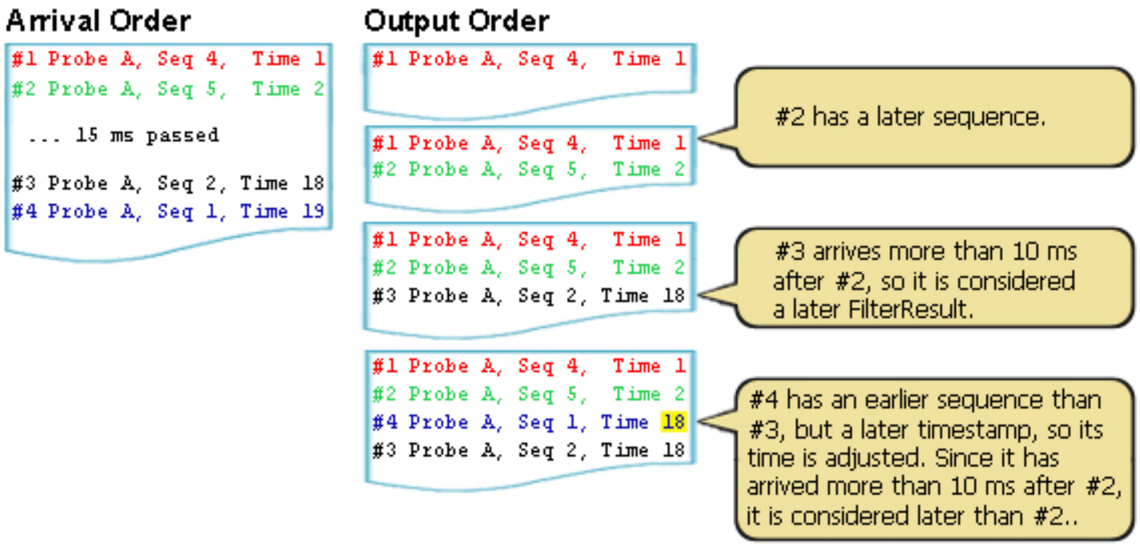
Sequencing Scenario: Multiple SFProbes with time adjusted to earlier



Sequencing Scenario: Single SFProbe with sequence breakpoint = 10



Sequencing Scenario: Single SFProbe with time breakpoint = 10 ms



Missing Packets

The LostCount function from FilterResultAccess can provide insights to the number filter results that are not provided to the application.

The LostCount is calculated as follows:

- If sequencing is turned off, this count is always 0 (not counted)
- If sequencing is turned on, it is the number of filter results missing according to sequencing number for all probes.

There are several reasons that packets are missing according to sequence number. A few of the reasons are:

- The probe is unable to inject FRPs because of bandwidth limiting settings. The probe will continue to increment sequence numbers when there is a packet filter match, but it won't inject the packets.
- The FRP is lost between the probe and the PRE.
- The FRP is lost between the PRE and the PacketPortal application.
- The FR/FRP is discarded by the API due to a buffer overflow (i.e. DiscardOverflowCount for FRs and DiscardOverflowFRPCount for FRPs). Excessive buffer overflow may be an indication that the FilterResultAccess' BufferSize is too small for the type of traffic.
- The FR is discarded by the API because it is set to discard late packets, and that the packet is "late". Too many late packets may indicate one or more issues in the PacketPortal System: the SFProbes are not time synchronized with the PREs, the PREs are not time synchronized with one another, or the network itself is improperly configured.

- The FR is fragmented and only one part of the packet arrives, in which case it will also be discarded and counted as “lost”.
- The application should also look at the CongestionCount for each filter result object to determine if there are potential, additional result that the SFProbe is unable to inject due to buffer overflow. These missing filtered packets will not be reflected in the LostCount value since the probe is unable to increment the sequence number in that case.

When packets (FRs or FRPs) are discarded by PacketAccess API, several counters are available to indicate the frequency and the reason:

Counters	Description
DiscardCount	FR discarded
DiscardOverflowCount	FR discarded because of buffer overflow
DiscardDuplicateCount	FR discarded because of duplicates
DiscardOutOfSequenceCount	FR discarded because the FR is late due to FR arriving after the application has consumed the FR with a later sequence number
DiscardLateCount	FR discarded because the FR is late due to time
DiscardFRPCount	FRP discarded
DiscardOverflowFRPCount	FRP discarded because of buffer overflow
DiscardDuplicateFRPCount	FRP discarded because of duplicates
DiscardOutOfSequenceFRPCount	FRP discarded because the FR is late due to FR arriving after the application has consumed the FR with a later sequence number
DiscardLateFRPCount	FRP discarded because the FR is late due to time
DiscardFragmentedFRPCount	FRP fragments discarded

Note that two of the counters are the sums of other counters in the list.

DiscardCount is the sum of the following counts:

DiscardOverflowCount

DiscardDuplicateCount

DiscardOutOfSequenceCount

DiscardLateCount

DiscardFRPCount is the sum of the following counts:

DiscardOverflowFRPCount

DiscardDuplicateFRPCount

DiscardOutOfSequenceFRPCount

DiscardLateFRPCount

DiscardFragmentedFRPCount

If the built-in counters listed below are not sufficient, you can create your own counters based on FilterResults object properties:

FilterResults Properties	Description
ProbeId	Probe identifier
Sequence	FRP identifier
IsBadFCS	True if the OCP has a bad FCS
IsHeaderOnly	True if the FR only contains network protocol headers
IsInjectNet	True if the FR was injected on the network side of the probe
IsLate	True if the FR is considered “late”
IsNet	True is the OCP is filtered on the network side of the probe
IsNewSequence	True if the OCP is the first packet of a feed
IsSliced	True if the filter expression requested the SFProbe to slice the payload of the OCP
IsTimingLock	True if the FR is capture when the probe is time synchronized with the PRE
WasFragmented	True if the FR was assembled from
Payload	FRP fragments discarded

Here are two examples of how to create custom counters:

- FragCount: Number of FRs that were built from FRP fragments. For all FRs: if “WasFragmented ()”, increment FragCount

- LostTimeSyncCount: number of FRs that were captured while the probe was not time synchronized. For all FRs: if “not IsTimingLock()”, increment LostTimeSyncCount.

Doc Part No.
Rev. 000, 10/1/13
<Language>