APPENDIX A

Using MIBs

This chapter describes how to work with MIBs on the Cisco Carrier Routing System. This appendix contains the following sections:

Cisco Unique Device Identifier Support

The ENTITY-MIB supports the Cisco compliance effort for a unique device identifier (UDI) standard stored in Identification Programmable Read-Only Memory (IDPROM).

The Cisco UDI provides a unique identity for every Cisco product. The UDI is composed of three separate data elements that must be stored in the entPhysicalTable:

- Orderable product identifier (PID)—Alphanumeric identifier used by customers to order Cisco products. Two examples include A9K-RSP-4G and A9K-4T-E. PID is limited to 18 characters and must be stored in the entPhysicalModelName object.
- Version identifier (VID)—Version of the PID. The VID indicates the number of times a product has versioned in ways that are reported to a customer. For example, the product identifier A9K-RSP-4G may have a VID of V04. VID is limited to three alphanumeric characters and must be stored in the entPhysicalHardwareRev object.
- Serial number (SN)—11-character identifier used to identify a specific part within a product and must be stored in the entPhysicalSerialNum object. Serial number content is defined by manufacturing part number 7018060-0000. The SN is accessed at the following website by searching on the part number 701806-0000:

  https://tools.cisco.com/emco/inbiz/inbiz/Home

  Serial number format is defined in four fields:

  - Location (L)
  - Year (Y)

- Workweek (W)
- Sequential serial ID (S)

The SN label is represented as: LLLYYWWSSS.

Note The Version ID returns NULL for those old or existing cards with IDPROMs that do not have the Version ID field. Therefore, corresponding entPhysicalHardwareRev returns NULL for cards that do not have the Version ID field in IDPROM.

Cisco Redundancy Features

Redundancy creates a duplication of data elements and software functions to provide an alternative in case of failure. The goal of Cisco redundancy features is to cut over without affecting the link and protocol states associated with each interface and continue packet forwarding. The state of the interfaces and subinterfaces is maintained, along with the state of line cards and various packet processing hardware.

This section describes Cisco redundancy feature:

- Levels of Redundancy, page A-170

Levels of Redundancy

This section describes the levels of redundancy supported on the Cisco Carrier Routing System and how to verify that this feature is available. The Cisco Carrier Routing System supports fault resistance by allowing a Cisco redundant Route Switch Processor (RSP) to take over if the active RSP fails. Redundancy prevents equipment failures from causing service outages, and supports hitless maintenance and upgrade activities. The state of the interfaces and subinterfaces is maintained along with the state of line cards and various packet processing hardware.

Redundant systems support two RSP. One acts as the active RSPs while the other acts as the standby RSPs.

The redundancy feature provides high availability for the Cisco routers by switching when one of the following conditions occur:

- Cisco IOS XR Software failure
- Software upgrade
- Maintenance procedure

The Cisco Carrier Routing System operates in Nonstop Forwarding/Stateful Switchover (NSF/SSO) mode.

Nonstop Forwarding/Stateful Switchover

This section describes the Nonstop Forwarding/Stateful Switchover mode. With NSF/SSO, the Cisco Carrier Routing System can change from the active to the standby RSPs almost immediately while continuing to forward packets. Cisco IOS XR Software NSF/SSO support on this platform enables immediate switchover.

In networking devices running NSF/SSO, both RSPs must be running the same configuration so that the standby RSP is always ready to assume control following a fault on the active RSP. The configuration information is synchronized from the active RSP to the standby RSP at startup and each timechanges to the active RSP configuration occur.

Following an initial synchronization between the two RSPs, NSF/SSO maintains RSP state information between them, including forwarding information.

The Cisco Nonstop Forwarding (NSF) works with Stateful Switchover (SSO) to minimize the amount of time a network is unavailable to its users following a Route Switching Processor (RSP) fail-over in a router with dual RSPs. NSF/SSO capability allows routers to detect a switchover and take the necessary actions to continue forwarding network traffic and to recover route information from peer devices.

The Cisco NSF works with the Stateful Switchover (SSO) feature in Cisco IOS XR Software to minimize the amount of time a network is unavailable to its users following a switchover. The main objective of the Cisco NSF/SSO is to continue forwarding data packets along known routes while the routing protocol information is restored following a route switchover.

Managing Physical Entities

This section describes how to use SNMP to manage the physical entities (components) in the router by:

- Performing Inventory Management, page A-172
- Monitoring and Configuring FRU Status, page A-177
- Generating SNMP Notifications, page A-178

Purpose and Benefits

The physical entity management feature of the Cisco Carrier Routing System SNMP implementation does the following:

- Monitors and configures the status of field-replaceable units (FRUs)
- Provides information about physical port to interface mappings
- Provides asset information for asset tagging
- Provides firmware and software information for chassis components

MIBs Used for Physical Entity Management

- CISCO-ENTITY-ASSET-MIB—Contains asset tracking information (IDPROM contents) for the physical entities listed in the entPhysicalTable of the ENTITY-MIB. The MIB provides device-specific information for physical entities, including orderable part number, serial number, manufacturing assembly number, and hardware, software, and firmware information.
- CISCO-ENTITY-FRU-CONTROL-MIB—Contains objects used to monitor and configure the administrative and operational status of field-replaceable units (FRUs), such as fans, RSPs, and transceivers that are listed in the entPhysicalTable of the ENTITY-MIB.
- CISCO-ENTITY-SENSOR-MIB—Contains information about entities in the entPhysicalTable with an entPhysicalClass value of sensor.

- ENTITY-MIB—Contains information for managing physical entities on the router. It also organizes the entities into a containment tree that depicts their hierarchy and relationship to each other. The MIB contains the following tables:
  - The entPhysicalTable describes each physical component (entity) in the router. The table contains an entry for the top-level entity (the chassis) and for each entity in the chassis. Each entry provides information about that entity: its name, type, vendor, and a description, and a description of how the entity fits into the hierarchy of chassis entities.

    Each entity is identified by a unique index (entPhysicalIndex) that is used to access information about the entity in this and other MIBs.
  - The entAliasMappingTable maps each physical port's entPhysicalIndex value to its corresponding ifIndex value in the IF-MIB ifTable.
  - The entPhysicalContainsTable shows the relationship between physical entities in the chassis. For each physical entity, the table lists the entPhysicalIndex for each of the entity's child objects.

Performing Inventory Management

To get information about entities in the router, perform a MIB walk on the ENTITY-MIB entPhysicalTable.

As you examine sample entries in the ENTITY-MIB entPhysicalTable, consider the following objects:

- entPhysicalIndex—Uniquely identifies each entity in the chassis. This index is also used to access information about the entity in other MIBs.
- entPhysicalContainedIn—Indicates the entPhysicalIndex of a component parent entity.
- entPhysicalParentRelPos—Shows the relative position of same-type entities that have the same entPhysicalContainedIn value (for example, chassis slots, and line card ports).

Note The container is applicable if the physical entity class is capable of containing one or more removable physical entities. For example, each (empty or full) slot in a chassis is modeled as a container. All removable physical entities should be modeled within a container entity, such as field-replaceable modules, fans, or power supplies.

Sample of ENTITY-MIB entPhysicalTable Entries

The samples in this section show how information is stored in the entPhysicalTable. You can perform asset inventory by examining entPhysicalTable entries.

Note The sample outputs and values that appear throughout this appendix are examples of data you can view when using MIBs.

The following display shows the ENTITY-MIB entPhysicalTable sample entries:

```
entPhysicalDescr.186 = 4-Port 10GE Extended Line Card, Requires XFPs
entPhysicalDescr.187 = Ten GigabitEthernet Port
entPhysicalDescr.188 = GigeEthernet XFP container
entPhysicalDescr.189 =
entPhysicalDescr.190 = Transceiver Temperature Sensor
entPhysicalDescr.191 = Transceiver Tx Power Sensor
entPhysicalDescr.192 = Transceiver Rx Power Sensor
entPhysicalDescr.193 = Transceiver Transmit Bias Current Sensor
entPhysicalDescr.194 = Line Card host
```

```
entPhysicalDescr.195 = Inlet Temperature Sensor
entPhysicalDescr.196 = Hot Temperature Sensor
entPhysicalDescr.197 = Voltage Sensor - IBV
entPhysicalDescr.198 = Voltage Sensor - 5.0V
entPhysicalDescr.199 = Voltage Sensor - VP3P3_CAN
entPhysicalDescr.200 = Voltage Sensor - 3.3V
```

Where **entPhysicalDescr** identifies the manufacturer name for the physical entity.

```
entPhysicalVendorType.186 = cevModuleA9K4x10GEE
entPhysicalVendorType.187 = cevPortGEXFP
entPhysicalVendorType.188 = cevContainerXFP
entPhysicalVendorType.189 = cevXFPUnknown
entPhysicalVendorType.190 = cevSensorTransceiverTemp
entPhysicalVendorType.191 = cevSensorTransceiverTxPwr
entPhysicalVendorType.192 = cevSensorTransceiverRxPwr
entPhysicalVendorType.193 = cevSensorTransceiverCurrent
entPhysicalVendorType.194 = cevModuleASR9KHost
entPhysicalVendorType.195 = cevSensorModuleInletTemp
entPhysicalVendorType.196 = cevSensorHotTemperature
entPhysicalVendorType.197 = cevSensorModuleDeviceVoltage
entPhysicalVendorType.198 = cevSensorModuleDeviceVoltage
entPhysicalVendorType.199 = cevSensorModuleDeviceVoltage
entPhysicalVendorType.200 = cevSensorModuleDeviceVoltage
```

Where **entPhysicalVendorType** identifies the unique vendor-specific hardware type of the physical entity.

```
entPhysicalContainedIn.186 = 92
entPhysicalContainedIn.187 = 186
entPhysicalContainedIn.188 = 187
entPhysicalContainedIn.189 = 188
entPhysicalContainedIn.190 = 189
entPhysicalContainedIn.191 = 189
entPhysicalContainedIn.192 = 189
entPhysicalContainedIn.193 = 189
entPhysicalContainedIn.194 = 186
entPhysicalContainedIn.195 = 194
entPhysicalContainedIn.196 = 194
entPhysicalContainedIn.197 = 194
entPhysicalContainedIn.198 = 194
entPhysicalContainedIn.199 = 194
entPhysicalContainedIn.200 = 194
```

Where **entPhysicalContainedIn** indicates the entPhysicalIndex of a parent entity (component).

```
entPhysicalClass.186 = module(9)
entPhysicalClass.187 = port(10)
entPhysicalClass.188 = container(5)
entPhysicalClass.189 = module(9)
entPhysicalClass.190 = sensor(8)
entPhysicalClass.191 = sensor(8)
entPhysicalClass.192 = sensor(8)
entPhysicalClass.193 = sensor(8)
entPhysicalClass.194 = module(9)
entPhysicalClass.195 = sensor(8)
entPhysicalClass.196 = sensor(8)
entPhysicalClass.197 = sensor(8)
entPhysicalClass.198 = sensor(8)
entPhysicalClass.199 = sensor(8)
entPhysicalClass.200 = sensor(8)
```

Where **entPhysicalClass** indicates the general type of hardware device.

```
entPhysicalParentRelPos.186 = 0
entPhysicalParentRelPos.187 = 1
entPhysicalParentRelPos.188 = 0
entPhysicalParentRelPos.189 = 0
entPhysicalParentRelPos.190 = 0
entPhysicalParentRelPos.191 = 1
entPhysicalParentRelPos.192 = 2
entPhysicalParentRelPos.193 = 3
entPhysicalParentRelPos.194 = 0
entPhysicalParentRelPos.195 = 0
entPhysicalParentRelPos.196 = 1
entPhysicalParentRelPos.197 = 2
entPhysicalParentRelPos.198 = 3
entPhysicalParentRelPos.199 = 4
entPhysicalParentRelPos.200 = 5
```

Where **entPhysicalParentRelPos** indicates the relative position of this child among the other entities.

```
entPhysicalName.186 = module 0/5/CPU0
entPhysicalName.187 = TenGigE0/5/0/1
entPhysicalName.188 = slot mau 0/5/CPU0/1
entPhysicalName.189 = module mau 0/5/CPU0/1
entPhysicalName.190 = temperature 0/5/CPU0/1
entPhysicalName.191 = power Tx 0/5/CPU0/1
entPhysicalName.192 = power Rx 0/5/CPU0/1
entPhysicalName.193 = current 0/5/CPU0/1
entPhysicalName.194 = module 0/5/CPU0
entPhysicalName.195 = temperature 0/5/CPU0
entPhysicalName.196 = temperature 0/5/CPU0
entPhysicalName.197 = voltage 0/5/CPU0
entPhysicalName.198 = voltage 0/5/CPU0
entPhysicalName.199 = voltage 0/5/CPU0
entPhysicalName.200 = voltage 0/5/CPU0
```

Where **entPhysicalName** provides the textual name of the physical entity.

```
entPhysicalHardwareRev.186 =
entPhysicalHardwareRev.187 =
entPhysicalHardwareRev.188 =
entPhysicalHardwareRev.189 =
entPhysicalHardwareRev.190 =
entPhysicalHardwareRev.191 =
entPhysicalHardwareRev.192 =
entPhysicalHardwareRev.193 =
entPhysicalHardwareRev.194 =
entPhysicalHardwareRev.195 =
entPhysicalHardwareRev.196 =
entPhysicalHardwareRev.197 =
entPhysicalHardwareRev.198 =
entPhysicalHardwareRev.199 =
entPhysicalHardwareRev.200 =
```

Where **entPhysicalHardwareRev** provides the vendor-specific hardware revision number (string) for the physical entity.

```
entPhysicalFirmwareRev.186 = Version 0.63(20081010:215422)
entPhysicalFirmwareRev.187 =
entPhysicalFirmwareRev.188 =
```

```
entPhysicalFirmwareRev.189 =
entPhysicalFirmwareRev.190 =
entPhysicalFirmwareRev.191 =
entPhysicalFirmwareRev.192 =
entPhysicalFirmwareRev.193 =
entPhysicalFirmwareRev.194 =
entPhysicalFirmwareRev.195 =
entPhysicalFirmwareRev.196 =
entPhysicalFirmwareRev.197 =
entPhysicalFirmwareRev.198 =
entPhysicalFirmwareRev.199 =
entPhysicalFirmwareRev.200 =
```

Where **entPhysicalFirmwareRev** provides the vendor-specific firmware revision number (string) for the physical entity.

```
entPhysicalSoftwareRev.186 = 3.7.2.24I
entPhysicalSoftwareRev.187 =
entPhysicalSoftwareRev.188 =
entPhysicalSoftwareRev.189 = 3.7.2.24I
entPhysicalSoftwareRev.190 =
entPhysicalSoftwareRev.191 =
entPhysicalSoftwareRev.192 =
entPhysicalSoftwareRev.193 =
entPhysicalSoftwareRev.194 = 3.7.2.24I
entPhysicalSoftwareRev.195 =
entPhysicalSoftwareRev.196 =
entPhysicalSoftwareRev.197 =
entPhysicalSoftwareRev.198 =
entPhysicalSoftwareRev.199 =
entPhysicalSoftwareRev.200 =
```

Where **entPhysicalSoftwareRev** provides the software revision number for the physical entity.

```
entPhysicalSerialNum.186 = FHH1213002A
entPhysicalSerialNum.187 =
entPhysicalSerialNum.188 =
entPhysicalSerialNum.189 = ECL114704JD
entPhysicalSerialNum.190 =
entPhysicalSerialNum.191 =
entPhysicalSerialNum.192 =
entPhysicalSerialNum.193 =
entPhysicalSerialNum.194 =
entPhysicalSerialNum.195 =
entPhysicalSerialNum.196 =
entPhysicalSerialNum.197 =
entPhysicalSerialNum.198 =
entPhysicalSerialNum.199 =
entPhysicalSerialNum.200 =
```

Where **entPhysicalSerialNum** provides the vendor-specific serial number (string) for the physical entity.

```
entPhysicalMfgName.186 = Cisco Systems Inc.
entPhysicalMfgName.187 =
entPhysicalMfgName.188 =
entPhysicalMfgName.189 =
entPhysicalMfgName.190 =
entPhysicalMfgName.191 =
entPhysicalMfgName.192 =
entPhysicalMfgName.193 =
```

```
entPhysicalMfgName.194 =
entPhysicalMfgName.195 =
entPhysicalMfgName.196 =
entPhysicalMfgName.197 =
entPhysicalMfgName.198 =
entPhysicalMfgName.199 =
entPhysicalMfgName.200 =
```

Where **entPhysicalMfgName** provides the manufacturer name for the physical component.

```
entPhysicalModelName.186 = A9K-4T-E
entPhysicalModelName.187 =
entPhysicalModelName.188 =
entPhysicalModelName.189 = ONS-XC-10G-S1
entPhysicalModelName.190 =
entPhysicalModelName.191 =
entPhysicalModelName.192 =
entPhysicalModelName.193 =
entPhysicalModelName.194 =
entPhysicalModelName.195 =
entPhysicalModelName.196 =
entPhysicalModelName.197 =
entPhysicalModelName.198 =
entPhysicalModelName.199 =
entPhysicalModelName.200 =
```

Where **entPhysicalModelName** provides the vendor-specific model name string for the physical component.

```
entPhysicalAlias.186 =
entPhysicalAlias.187 =
entPhysicalAlias.188 =
entPhysicalAlias.189 =
entPhysicalAlias.190 =
entPhysicalAlias.191 =
entPhysicalAlias.192 =
entPhysicalAlias.193 =
entPhysicalAlias.194 = host
entPhysicalAlias.195 =
entPhysicalAlias.196 =
entPhysicalAlias.197 =
entPhysicalAlias.198 =
entPhysicalAlias.199 =
entPhysicalAlias.200 =
```

Where **entPhysicalAlias** provides the alias name for the physical component.

```
entPhysicalAssetID.186 =
entPhysicalAssetID.187 =
entPhysicalAssetID.188 =
entPhysicalAssetID.189 =
entPhysicalAssetID.190 =
entPhysicalAssetID.191 =
entPhysicalAssetID.192 =
entPhysicalAssetID.193 =
entPhysicalAssetID.194 =
entPhysicalAssetID.195 =
entPhysicalAssetID.196 =
entPhysicalAssetID.197 =
entPhysicalAssetID.198 =
entPhysicalAssetID.199 =
entPhysicalAssetID.200 =
```

Where **entPhysicalAssetID** provides the vendor-specific asset ID for the physical component.

```
entPhysicalIsFRU.186 = true(1)
entPhysicalIsFRU.187 = false(2)
entPhysicalIsFRU.188 = false(2)
entPhysicalIsFRU.189 = true(1)
entPhysicalIsFRU.190 = false(2)
entPhysicalIsFRU.191 = false(2)
entPhysicalIsFRU.192 = false(2)
entPhysicalIsFRU.193 = false(2)
entPhysicalIsFRU.194 = false(2)
entPhysicalIsFRU.195 = false(2)
entPhysicalIsFRU.196 = false(2)
entPhysicalIsFRU.197 = false(2)
entPhysicalIsFRU.198 = false(2)
entPhysicalIsFRU.199 = false(2)
entPhysicalIsFRU.200 = false(2)
```

Where **entPhysicalIsFRU** indicates whether or not this physical entity is considered a field-replaceable unit (FRU).

Note the following about the sample configuration:

- All chassis slots and line card ports have the same entPhysicalContainedIn value:
  - For chassis slots, entPhysicalContainedIn = 1 (the entPhysicalIndex of the chassis).
  - For line card ports, entPhysicalContainedIn = 26 (the entPhysicalIndex of the line card).
- Each chassis slot and line card port has a different entPhysicalParentRelPos to show its relative position within the parent object.

Determining the ifIndex Value for a Physical Port

The ENTITY-MIB entAliasMappingIdentifier maps a physical port to an interface by mapping the port's entPhysicalIndex to its corresponding ifIndex value in the IF-MIB ifTable. The following sample shows that the physical port with a entPhysicalIndex value of 35 is associated with the interface with the ifIndex value of four:

```
entAliasMappingIdentifer.35.0 = ifIndex.4
```

Note See the MIB for detailed descriptions of possible MIB values.

Monitoring and Configuring FRU Status

View objects in the CISCO-ENTITY-FRU-CONTROL-MIB cefcModuleTable to determine the administrative and operational status of FRUs, such as power supplies and line cards:

- cefcModuleAdminStatus—administrative state of the FRU. This object is read-only and returns enable.
- cefcModuleOperStatus—current operational state of the FRU.

Figure A-1 shows a cefcModuleTable entry for a line card with the entPhysicalIndex value of 24.

Figure A-1 Sample cefcModuleTable Entry

```
cefcModuleEntry.entPhysicalIndex

cefcModuleEntry.24
  cefcModuleAdminStatus = enabled(1)
  cefcModuleOperStatus = ok(2)
  cefcModuleResetReason = manual reset(5)
  cefcModuleStatusLastChangeTime = 7714
```

See the "FRU Status Changes" section on page A-179 for information about how the router generates notifications to indicate changes in FRU status.

Generating SNMP Notifications

This section provides information about the SNMP notifications generated in response to events and conditions on the router, and describes how to identify which hosts are to receive notifications.

Identifying Hosts to Receive Notifications

You can use the CLI or SNMP to identify hosts to receive SNMP notifications and to specify the types of notifications they are to receive (notifications). For CLI instructions, see the "Enabling Notifications" section on page 6-164. To use SNMP to configure this information:

Use SNMP-NOTIFICATION-MIB objects, including the following examples, to select target hosts and specify the types of notifications to generate for those hosts:

- snmpNotifyTable—Contains objects to select hosts and notification types:
  - snmpNotifyTag is an arbitrary octet string (a tag value) used to identify the hosts to receive SNMP notifications. Information about target hosts is defined in the snmpTargetAddrTable (SNMP-TARGET-MIB), and each host has one or more tag values associated with it. If a host in snmpTargetAddrTable has a tag value that matches this snmpNotifyTag value, the host is selected to receive the types of notifications specified by snmpNotifyType.
  - snmpNotifyType is the type of SNMP notification to send: notification(1) or inform(2).
- snmpNotifyFilterProfileTable and snmpNotifyFilterTable—Use objects in these tables to create notification filters to limit the types of notifications sent to target hosts.

Use SNMP-TARGET-MIB objects to configure information about the hosts to receive notifications:

- snmpTargetAddrTable—Transport addresses of hosts to receive SNMP notifications. Each entry provides information about a host address, including a list of tag values:

  - snmpTargetAddrTagList—set of tag values associated with the host address. If a host tag value matches snmpNotifyTag, the host is selected to receive the types of notifications defined by snmpNotifyType.
- snmpTargetParamsTable—SNMP parameters to use when generating SNMP notifications.

Use the notification enable objects in appropriate MIBs to enable and disable specific SNMP notifications.

Configuration Changes

If entity notifications are enabled, the router generates an entConfigChange notification (ENTITY-MIB) when the information in any of the following tables changes (which indicates a change to the router configuration):

- entPhysicalTable
- entAliasMappingTable
- entPhysicalContainsTable

Note A management application that tracks configuration changes checks the value of the entLastChangeTime object to detect any entConfigChange notifications that were missed as a because of throttling or transmission loss.

Enabling Notifications for Configuration Changes

To configure the router to generate an entConfigChange notification each time its configuration changes, enter the **snmp-sever trap entity** command from the CLI. Use the **no** form of the command to disable the notifications.

```
Router(config)# snmp-server traps entity
Router(config)# no snmp-server traps entity
```

FRU Status Changes

If FRU notifications are enabled, the router generates the following notifications in response to changes in the status of a FRU:

- cefcModuleStatusChange—The operational status (cefcModuleOperStatus) of a FRU changes.
- cefcFRUInserted—A FRU is inserted in the chassis. The notification indicates the entPhysicalIndex of the FRU and the container in which it was inserted.
- cefcFRURemoved—A FRU is removed from the chassis. The notification indicates the entPhysicalIndex of the FRU and the container from which it was removed.

Note See the CISCO-ENTITY-FRU-CONTROL-MIB for more information about these notifications.

Enabling FRU Notifications

To configure the router to generate notifications for FRU events, enter the **snmp-server traps fru-ctrl** command from the CLI. Use the **no** form of the command to disable the notifications.

```
Router(config)# snmp-server traps fru-ctrl
Router(config)# no snmp-server traps fru-ctrl
```

To enable FRU notifications through SNMP, set cefcMIBEnableStatusNotification to true (1). Disable the notifications by setting cefcMIBEnableStatusNotification to false (2).

Monitoring Quality of Service

This section provides the following information about using Quality of Service (QoS) in configuration:

Cisco Carrier Routing System QoS Basics

The Cisco Carrier Routing System distributes QoS features across the line cards. Line cards are designed to provide QoS features on packets that flow through the line cards.

CISCO-CLASS-BASED-QOS-MIB Overview

The CISCO-CLASS-BASED-QOS-MIB provides read-only access to QoS configuration information and statistics for Cisco platforms that support the modular Quality of Service command-line interface .

CISCO-CLASS-BASED-QOS-MIB Object Relationship

To understand how to navigate the CISCO-CLASS-BASED-QOS-MIB tables, it is important to understand the relationship among different QoS objects. QoS objects consists of:

- Match statement—Specific match criteria to identify packets for classification purposes.
- Class map—User-defined traffic class that contains one or more match statements used to classify packets into different categories.
- Feature action—Action taken on classified traffic. Features include police, traffic shaping, queueing, random detect, and packet marking. After the traffic is classified, actions are applied to packets matching each traffic class.
- Policy map—User-defined policy that associates QoS feature actions to user-defined class maps as policy maps can have multiple class maps.
- Service policy—Policy map that has been attached to an interface.

The MIB uses the following indices to identify QoS features and distinguish among instances of those features:

- cbQosObjectsIndex—Identifies each QoS feature on the router.
- cbQoSConfigIndex—Identifies a type of QoS configuration. This index is shared by QoS objects that have identical configurations.
- cbQosPolicyIndex—Identifies a unique service policy.

QoS MIB Information Storage

CISCO-CLASS-BASED-QOS-MIB information is stored as:

- Configuration information—Includes all the QoS configuration objects, such as class maps, policy map, match statements, and feature action configuration parameters. The configuration may have multiple identical instances. Configuration objects are identified by cbQosConfigIndex attribute. Multiple instances of the same QoS feature share a single configuration object that is identified by the same cbQosConfigIndex value.
- Service-policy information—Includes instances of all QoS objects, such as service-policies, classes, match statements, and feature actions. Service-policies are identified by cbQosPolicyIndex and instances of QoS objects are identified by the combination of cbQosPolicyIndex and cbQosObjectsIndex attributes.

QoS Hardware Configuration and Statistic Support

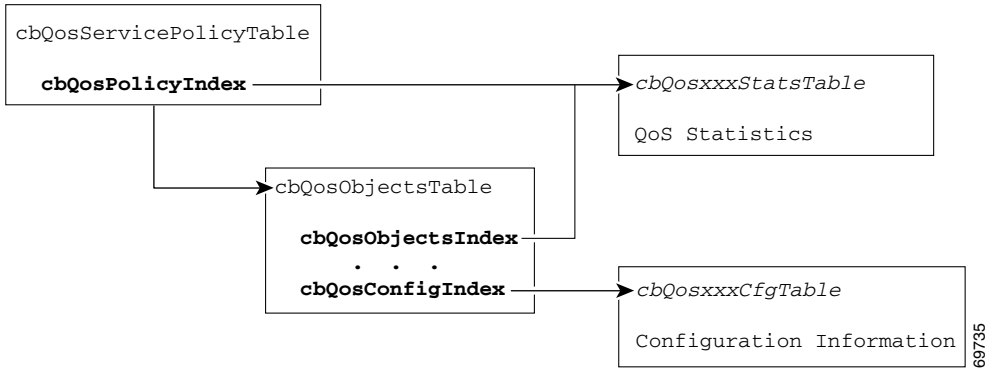
The CISCO-CLASS-BASED-QOS-MIB does not cover all the Cisco Carrier Routing System QoS hardware configuration and statistics.

The Cisco Carrier Routing System supports the concept of ‘shared policy instance’ where, based on the configuration, the resources for individual service policies are shared among multiple interfaces. The cbQosMIB attribute does not indicate whether the service-policies are shared-policy instances or non-shared policy instances.

The interfaces associated with the shared policy instance have a separate entry in the cbQosServicePolicyTable. The MIB entries, associated with each interface that is a part of the same shared-policy-instance, have the same data values, for example, everything except for the cbQosServicePolicyTable is identical for the rows associated with the values of cbQosPolicyIndex for such interfaces.

Figure A-2 shows how the indexes provide access to QoS configuration information and statistics.

Figure A-2 The Cisco Carrier Routing System QoS Indexes



Accessing QoS Configuration Information

To access QoS configuration information and statistics for a particular QoS feature:

Step 1 Look in cbQosServicePolicyTable and find the cbQosPolicyIndex assigned to the policy in which the feature is used.

Step 2 Use cbQosPolicyIndex to access the cbQosObjectsTable, and find the cbQosObjectsIndex and cbQosConfigIndex assigned to the QoS feature.

a. Use cbQosConfigIndex to access configuration tables (cbQosxxxCfgTable) for information about the QoS feature.

b. Use cbQosPolicyIndex and cbQosObjectsIndex to access QoS statistics tables (cbQosxxxStatsTable) for information about the QoS feature.

Viewing QoS Configuration Settings Using the CISCO-CLASS-BASED-QOS-MIB

This section contains an example that shows how QoS configuration settings are stored in CISCO-CLASS-BASED-QOS-MIB tables. The sample shows information grouped by QoS object; however, the actual output of an SNMP query might show QoS information similar to the following.

Note This is only a partial display of all QoS information.

```
ASR 9000# getmany -v3 10.86.0.94 test-user ciscoCBQosMIB CbQosServicePolicyTable
cbQosIfType.1047 = subInterface(2)
cbQosIfType.1052 = subInterface(2)
cbQosPolicyDirection.1047 = input(1)
cbQosPolicyDirection.1052 = output(2)
cbQosIfIndex.1047 = 36
cbQosIfIndex.1052 = 36
cbQosFrDLCI.1047 = 0
cbQosFrDLCI.1052 = 0
cbQosAtmVPI.1047 = 0
cbQosAtmVPI.1052 = 0
cbQosAtmVCI.1047 = 0
cbQosAtmVCI.1052 = 0
cbQosConfigIndex.1047.1047 = 1045
cbQosConfigIndex.1047.1048 = 1025
cbQosConfigIndex.1047.1050 = 1027
cbQosConfigIndex.1047.1051 = 1046
cbQosConfigIndex.1052.1052 = 1045
cbQosConfigIndex.1052.1053 = 1025
cbQosConfigIndex.1052.1055 = 1027
cbQosConfigIndex.1052.1056 = 1046
cbQosObjectsType.1047.1047 = policymap(1)
cbQosObjectsType.1047.1048 = classmap(2)
cbQosObjectsType.1047.1050 = matchStatement(3)
cbQosObjectsType.1047.1051 = police(7)
cbQosObjectsType.1052.1052 = policymap(1)
cbQosObjectsType.1052.1053 = classmap(2)
cbQosObjectsType.1052.1055 = matchStatement(3)
cbQosObjectsType.1052.1056 = police(7)
cbQosParentObjectsIndex.1047.1047 = 0
cbQosParentObjectsIndex.1047.1048 = 1047
cbQosParentObjectsIndex.1047.1050 = 1048
cbQosParentObjectsIndex.1047.1051 = 1048
cbQosParentObjectsIndex.1052.1052 = 0
cbQosParentObjectsIndex.1052.1053 = 1052
cbQosParentObjectsIndex.1052.1055 = 1053
```

```
cbQosParentObjectsIndex.1052.1056 = 1053
cbQosPolicyMapName.1045 = pm-1Meg
cbQosPolicyMapDesc.1045 =
cbQosCMName.1025 = class-default
cbQosCMDesc.1025 =
cbQosCMInfo.1025 = matchAny(3)
. . .
```

Monitoring QoS Using the CISCO-CLASS-BASED-QOS-MIB

This section describes how to monitor QoS on the router by checking the QoS statistics in the CISCO-CLASS-BASED-QOS-MIB tables.

Note The CISCO-CLASS-BASED-QOS-MIB may contain more information than what is displayed in the output of CLI **show** commands.

Table A-1 lists the types of QoS statistics tables.

Table A-1 QoS Statistics Tables

QoS Table	Statistics
cbQosCMStatsTable	Class map—Counts of packets, bytes, and bit rate before and after QoS policies are executed. Counts of dropped packets and bytes.
cbQosPoliceStatsTable	Police action—Counts of packets, bytes, and bit rate that conforms to, exceeds, and violates police actions.
cbQosQueueingStatsTable	Queueing—Counts of discarded packets and bytes, and queue depths.
cbQosTSStatsTable	Traffic shaping—Counts of delayed and dropped packets and bytes, the state of a feature, and queue size.
cbQosREDClassStatsTable	Random early detection—Counts of packets and bytes dropped when queues are full, and counts of bytes and octets transmitted.

Considerations for Processing QoS Statistics

The router maintains 64-bit counters for most QoS statistics. However, some QoS counters are implemented as a 32-bit counter with a 1-bit overflow flag. In the following samples, the counters are shown as 33-bit counters.

When accessing QoS counter statistics, consider the following:

- SNMPv2c or SNMPv3 applications—Access the entire 64 bits of the QoS counter through cbQosxxx64 MIB objects.
- SNMPv1 applications—Access QoS statistics in the MIB as follows:
  - Access the lower 32 bits of the counter through cbQosxxx MIB objects.
  - Access the upper 32 bits of the counter through cbQosxxxOverflow MIB objects.

Sample QoS Statistics Tables

The samples in this section show the counters in CISCO-CLASS-BASED-QOS-MIB statistics tables:

- Figure A-3 shows the counters in the cbQosCMStatsTable and the indexes for accessing these and other statistics.
- Figure A-4 shows the counters in cbQosMatchStmtStatsTable, cbQosPoliceStatsTable, cbQosQueueingStatsTable, cbQosTSStatsTable, and cbQosREDClassStatsTable.

For ease-of-use, the following figures show some counters as a single object even though the counter is implemented as three objects. For example, cbQosCMPrePolicyByte is implemented as:

- cbQosCMPrePolicyByteOverflow
- cbQosCMPrePolicyByte
- cbQosCMPrePolicyByte64

Figure A-3 QoS Class Map Statistics and Indexes

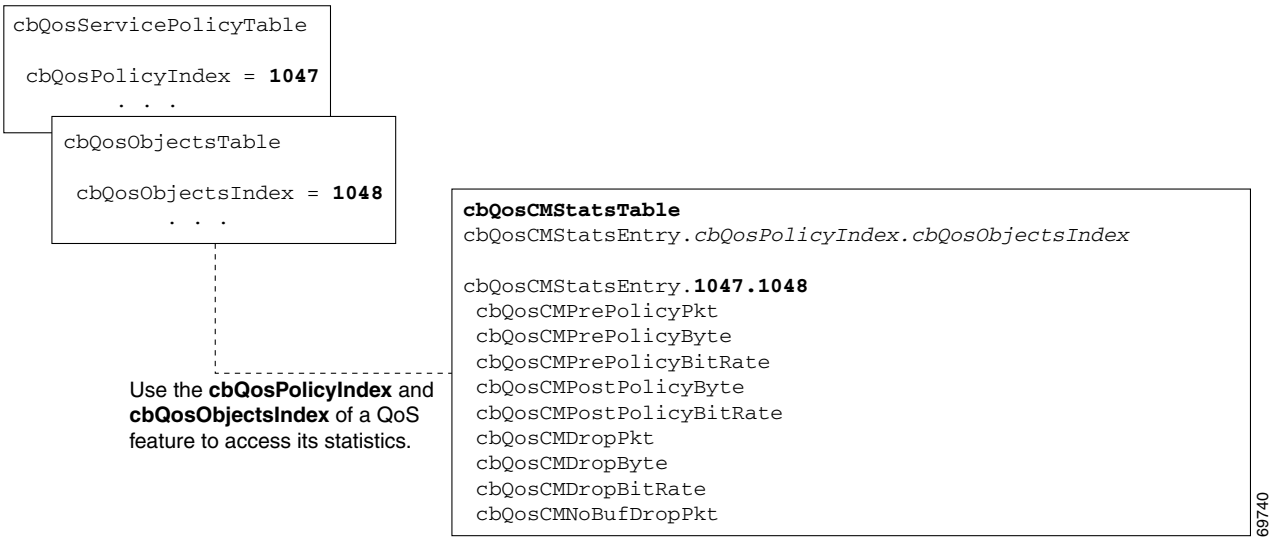
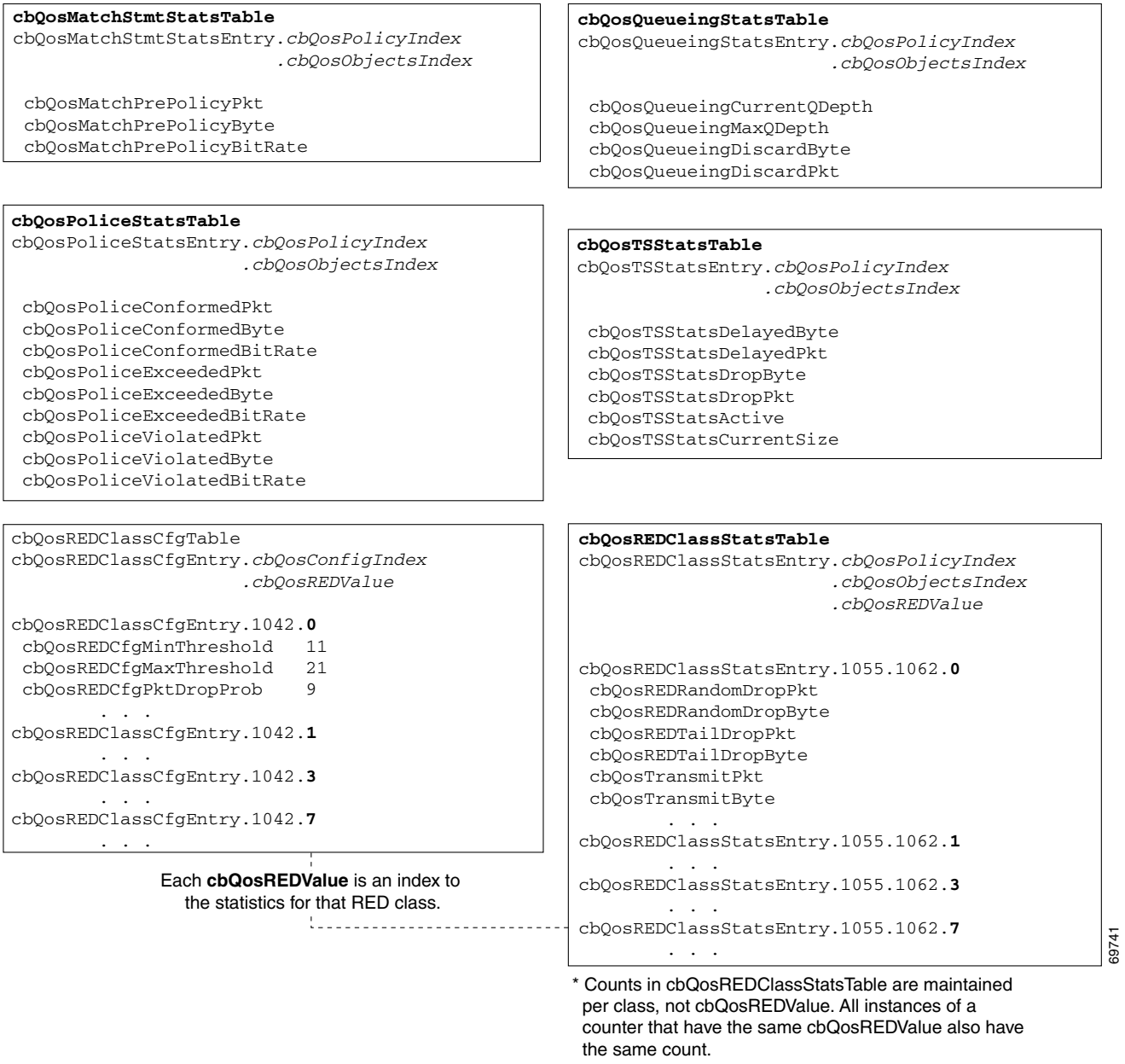


Figure A-4 QoS Statistics Tables



Sample QoS Applications

This section presents examples of code showing how to retrieve information from the CISCO-CLASS-BASED-QOS-MIB to use for QoS billing operations. You can use the examples to help you develop billing applications. The topics include:

- Checking Customer Interfaces for Service Policies, page A-186
- Retrieving QoS Billing Information, page A-187

Checking Customer Interfaces for Service Policies

This section describes a sample algorithm that checks the CISCO-CLASS-BASED-QOS-MIB for customer interfaces with service policies, and marks those interfaces for further application processing (such as billing for QoS services).

The algorithm uses two SNMP **get-next** requests for each customer interface. For example, if the router has 2000 customer interfaces, 4000 SNMP **get-next** requests are required to determine if those interfaces have transmit and receive service policies associated with them.

Note This algorithm is for informational purposes only. Your application needs may be different.

Check the MIB to see which interfaces are associated with a customer. Create a pair of flags to show if a service policy has been associated with the transmit and receive directions of a customer interface. Mark noncustomer interfaces TRUE (so no more processing is required for them).

```
FOR each ifEntry DO
  IF (ifEntry represents a customer interface) THEN
     servicePolicyAssociated[ifIndex].transmit = FALSE;
     servicePolicyAssociated[ifIndex].receive = FALSE;
  ELSE
     servicePolicyAssociated[ifIndex].transmit = TRUE;
     servicePolicyAssociated[ifIndex].receive = TRUE;
  END-IF
END-FOR
```

Examine the cbQosServicePolicyTable and mark each customer interface that has a service policy attached to it. Also note the direction of the interface.

```
x = 0;
done = FALSE;
WHILE (!done)
  status = snmp-getnext (
           ifIndex = cbQosIfIndex.x,
           direction = cbQosPolicyDirection.x
  );
  IF (status != 'noError') THEN
     done = TRUE
  ELSE
     x = extract cbQosPolicyIndex from response;
     IF (direction == 'output') THEN
       servicePolicyAssociated[ifIndex].transmit = TRUE;
     ELSE
       servicePolicyAssociated[ifIndex].receive = TRUE;
     END-IF
  END-IF
END-WHILE
```

Manage cases in which a customer interface does not have a service policy attached to it.

```
FOR each ifEntry DO
  IF (!servicePolicyAssociated[ifIndex].transmit) THEN
     Perform processing for customer interface without a transmit service policy.
  END-IF
  IF (!servicePolicyAssociated[ifIndex].receive) THEN
     Perform processing for customer interface without a receive service policy.
  END-IF
END-FOR
```

Retrieving QoS Billing Information

This section describes a sample algorithm that uses the CISCO-CLASS-BASED-QOS-MIB for QoS billing operations. The algorithm periodically retrieves post-policy input and output statistics, combines them, and sends the result to a billing database.

The algorithm uses the following:

- One SNMP **get** request per customer interface—To retrieve the ifAlias.
- Two SNMP **get-next** requests per customer interface—To retrieve service policy indexes.
- Two SNMP **get-next** requests per customer interface for each object in the policy—To retrieve post-policy bytes. For example, if there are 100 interfaces and 10 objects in the policy, the algorithm requires 2000 **get-next** requests (2 x 100 x 10).

Note This algorithm is for informational purposes only. Your application needs may be different.

Set up customer billing information.

```
FOR each ifEntry DO
  IF (ifEntry represents a customer interface) THEN
     status = snmp-getnext (id = ifAlias.ifIndex);
     IF (status != 'noError') THEN
         Perform error processing.
     ELSE
        billing[ifIndex].isCustomerInterface = TRUE;
        billing[ifIndex].customerID = id;
        billing[ifIndex].transmit   = 0;
        billing[ifIndex].receive    = 0;
     END-IF
  ELSE
     billing[ifIndex].isCustomerInterface = FALSE;
  END-IF
END-FOR
```

Retrieve billing information.

```
x = 0;
done = FALSE;
WHILE (!done)
  response = snmp-getnext (
             ifIndex = cbQosIfIndex.x,
             direction = cbQosPolicyDirection.x
  );
  IF (response.status != 'noError') THEN
     done = TRUE
  ELSE
     x = extract cbQosPolicyIndex from response;
     IF (direction == 'output') THEN
        billing[ifIndex].transmit = GetPostPolicyBytes (x);
     ELSE
        billing[ifIndex].receive = GetPostPolicyBytes (x);
     END-IF
  END-IF
END-WHILE
```

Determine the number of post-policy bytes for billing purposes.

```
GetPostPolicyBytes (policy)
  x = policy;
  y = 0;
  total = 0;
  WHILE (x == policy)
     response = snmp-getnext (type = cbQosObjectsType.x.y);
     IF (response.status == 'noError')
        x = extract cbQosPolicyIndex from response;
        y = extract cbQosObjectsIndex from response;
        IF (x == policy AND type == 'classmap')
           status = snmp-get (bytes = cbQosCMPostPolicyByte64.x.y);
           IF (status == 'noError')
                   total += bytes;
           END-IF
        END-IF
     END-IF
  END-WHILE
RETURN total;
```

Monitoring Router Interfaces

This section provides information about how to monitor the status of router interfaces to see if there is a problem or a condition that might affect service on the interface. To determine if an interface is Down or experiencing problems, you can:

- see the "Check the Operational and Administrative Status of Interface" section on page A-188
- see the "Monitor linkDown and linkUp Notifications" section on page A-188

Check the Operational and Administrative Status of Interface

To check the status of an interface, view the following IF-MIB objects for the interface:

- ifAdminStatus—Administratively configured (desired) state of an interface. Use ifAdminStatus to enable or disable the interface.
- ifOperStatus—Current operational state of an interface.

Monitor linkDown and linkUp Notifications

To determine if an interface has failed, you can monitor linkDown and linkUp notifications for the interface. See the "Enabling Interface linkUp and linkDown Notifications" section on page A-189 for instructions on how to enable the following notifications:

- linkDown—Indicates that an interface failed or is about to fail.
- linkUp—Indicates that an interface is no longer in the down state.

Enabling Interface linkUp and linkDown Notifications

To configure SNMP to send a notification when a router interface changes state to up (ready) or down (not ready), perform the following steps to enable linkUp and linkDown notifications:

Step 1 Issue the following CLI command to enable linkUp and linkDown notifications for most, but not necessarily all, interfaces:

```
Router(config)# snmp-server interface <Interface Type> <Interface Number> notification linkupdown
```

Step 2 View the setting of the ifLinkUpDownTrapEnable object (IF-MIB ifXTable) for each interface to determine if linkUp and linkDown notifications are enabled or disabled for that interface.

Step 3 To enable linkUp and linkDown notifications on an interface, set ifLinkUpDownTrapEnable to enabled (1).

Step 4 To enable the Internet Engineering Task Force (IETF) standard for linkUp and linkDown notifications, issue the **snmp-server trap link ietf** command. (The IETF standard is based on RFC 2233.)

```
Router(config)# snmp-server trap link ietf
```

Step 5 To disable notifications, use the **no** form of the **snmp-server** command.

Billing Customers for Traffic

This section describes how to use SNMP interface counters and QoS data information to determine the amount to bill customers for traffic. It also includes a scenario for demonstrating that a QoS service policy attached to an interface is policing traffic on that interface.

This section contains the following topics:

Input and Output Interface Counts

The router maintains information about the number of packets and bytes that are received on an input interface and transmitted on an output interface.

For detailed constraints about IF-MIB counter support, see the "CISCO-MAU-EXT-MIB" section on page 4-155.

Consider the following important information about IF-MIB counter support:

- Unless noted, all IF-MIB counters are supported on the Cisco Carrier Routing System interfaces.
- For IF-MIB high capacity counter support, Cisco conforms to the RFC 2863 standard. The RFC 2863 standard states that for interfaces that operate:
  - At 20 million bits per second or less, 32-bit and packet counters *must* be supported.

  - Faster than 20 million bits per second and slower than 650 million bits per second, 32-bit packet counters and 64-bit octet counters *must* be supported.
  - At 650 million bits per second or faster, 64-bit packet counters and 64-bit octet counters *must* be supported.
- When a QoS service policy is attached to an interface, the router applies the rules of the policy to traffic on the interface and increments the packet and byte counts on the interface.

The following CISCO-CLASS-BASED-QOS-MIB objects provide interface counts:

- cbQosCMDropPkt and cbQosCMDropByte (cbQosCMStatsTable)—Total number of packets and bytes that were dropped as they exceeded the limits set by the service policy. These counts include only those packets and bytes that were dropped as they exceeded service policy limits. The counts do not include packets and bytes dropped for other reasons.
- cbQosPoliceConformedPkt and cbQosPoliceConformedByte (cbQosPoliceStatsTable)—Total number of packets and bytes that conformed to the limits of the service policy and were transmitted.

Determining the Amount of Traffic to Bill to a Customer

Perform the following steps to determine how much traffic on an interface is billable to a particular customer:

Step 1 Determine which service policy on the interface applies to the customer.

Step 2 Determine the index values of the service policy and class map used to define the customer's traffic. You need this information in the following steps.

Step 3 Access the cbQosPoliceConformedPkt object (cbQosPoliceStatsTable) for the customer to determine the amount of traffic on the interface that is billable to this customer.

Step 4 (Optional) Access the cbQosCMDropPkt object (cbQosCMStatsTable) for the customer to determine how much of the customer's traffic was dropped as it exceeded service policy limits.

Scenario for Demonstrating QoS Traffic Policing

This section describes a scenario that demonstrates the use of SNMP QoS statistics to determine how much traffic on an interface is billable to a particular customer. It also shows how packet counts are affected when a service policy is applied to traffic on the interface.

To create the scenario, perform the following steps (each step described in the section below):

Step 1 Create and attach a service policy to an interface.

Step 2 View packet counts before the service policy is applied to traffic on the interface.

Step 3 Issue a **ping** command to generate traffic on the interface. Note that the service policy is applied to the traffic.

Step 4 View packet counts after the service policy is applied to determine how much traffic to bill the customer for:

a. Conformed packets—The number of packets within the range set by the service policy and for which you can charge the customer.

b. Exceeded or dropped packets—The number of packets that were not transmitted because they were outside the range of the service policy. These packets are not billable to the customer.

Note In this scenario, the Cisco Carrier Routing System is used as an interim device (that is, traffic originates elsewhere and is destined for another device).

Service Policy Configuration

The following example uses policy map configuration:

```
policy-map police-out
  class BGPclass
    police 8000 1000 2000 conform-action transmit exceed-action drop

interface GigabitEthernet0/1/0/0.10
 description VLAN voor klant
 encapsulation dot1Q 10
 ip address 10.0.0.17 255.255.255.248
 service-policy output police-out
```

Packet Counts Before the Service Policy Is Applied

The following CLI and SNMP output shows the output traffic for interface before the service policy is applied:

CLI Command Output

```
RSP/0/RSP0/CPU0:ios-xr# show policy-map interface GigabitEthernet0/7/0/0.1

GigabitEthernet0/7/0/0.1 input: policy-police

Class class-out
Classification statistics (packets/bytes) (rate - kbps)
Matched : 0/0 0
Transmitted : Un-determined
Total Dropped : Un-determined
Policing statistics (packets/bytes) (rate - kbps)
Policed(conform) : 0/0 0
Policed(exceed) : 0/0 0
Policed(violate) : 0/0 0
Policed and dropped : 0/0
Class class-default
Classification statistics (packets/bytes) (rate - kbps)
Matched : 0/0 0
Transmitted : Un-determined
Total Dropped : Un-determined
```

SNMP Output

```
ASR 9000# getone -v2c 10.86.0.63 public ifDescr.65
ifDescr.65 = GigabitEthernet0/6/0/0.10
```

Generating Traffic

The following set of **ping** commands generates traffic:

```
ASR 9000# ping
Protocol [ip]:
Target IP address: 10.0.0.18
Repeat count [5]: 99
Datagram size [100]: 1400
Timeout in seconds [2]: 1
Extended commands [n]:
Sweep range of sizes [n]:
Type escape sequence to abort.

Sending 100, 1400-byte ICMP Echos to 10.0.0.18, timeout is 1 seconds:
..!!..!..!...!..!.!..!..!..!..!.!.!..!.!..!.!...!.!.!...!.!..!.!..!.!.!..
!.!..!.!..!.!.!..!.!..!.!..!.!
Success rate is 42 percent (42/100), round-trip min/avg/max = 1/1/1 ms
```

Packet Counts After the Service Policy Is Applied

After you generate traffic using the **ping** command, look at the number of packets that exceeded and conformed to the committed access rate (CAR) set by the **police** command:

- 42 packets conformed to the police rate and were transmitted
- 57 packets exceeded the police rate and were dropped

The following CLI and SNMP output show the counts on the interface after the service policy is applied: (In the output, conformed and exceeded packet counts are shown in boldface.)

CLI Command Output

```
ASR 9000# show policy-map interface g6/0/0.10

GigabitEthernet6/0/0.10

Service-policy output: police-out

   Class-map: BGPclass (match-all)
      198 packets, 281556 bytes
      30 second offered rate 31000 bps, drop rate 11000 bps
      Match: access-group 101
      Police:
        8000 bps, 1000 limit, 2000 extended limit
        conformed 42 packets, 59892 bytes; action: transmit
        exceeded 57 packets, 81282 bytes; action: drop

   Class-map: class-default (match-any)
      15 packets, 1086 bytes
      30 second offered rate 0 bps, drop rate 0 bps
      Match: any
      Output queue: 0/8192; 48/59940 packets/bytes output, 0 drops
```

SNMP Output

```
ASR 9000# getmany -v2c 10.86.0.63 public ciscoCBQosMIB
        . . .
      cbQosCMDropPkt.1143.1145 = 57
        . . .
      cbQosPoliceConformedPkt.1143.1151 = 42
        . . .
```

Using IF-MIB Counters

This section describes the IF-MIB counters and how you can use them on various interfaces and subinterfaces. The subinterface counters are specific to the protocols. This section addresses the IF-MIB counters for ATM interfaces.

The IF-MIB counters are defined considering the lower and upper layers:

- ifInDiscards—Number of inbound packets that were discarded, even though no errors were detected to prevent their being deliverable to a higher-layer protocol. One reason for discarding such a packet is to free up buffer space.
- IfInErrors—Number of inbound packets that contained errors preventing them from being deliverable to a higher-layer protocol for packet-oriented interfaces.
- ifInUnknownProtos—Number of packets received through the interface that were discarded because of an unknown or unsupported protocol for packet-oriented interfaces.
- ifOutDiscards—Number of outbound packets that were discarded even though no errors were detected to prevent their being transmitted. One reason for discarding such a packet is to free up buffer space.
- ififOutErrors—Number of outbound packets that could not be transmitted because of errors for packet-oriented interfaces.

The logical flow for counters works as follows:

1. When a packet arrives on an interface, check for the following:
   a. Error in packet—If any errors are detected, increment ifInErrors and drop the packet.
   b. Protocol errors—If any errors are detected, increment ifInUnknownProtos and drop the packet.
   c. Resources (buffers)—If unable to get resources, increment ifInDiscards and drop the packet.
   d. Increment ifInUcastPkts/ifInNUcastPkts and process the packet (at this point, increment ifInOctets with the size of packet).
2. When a packet is to be sent out of an interface:
   a. Increment ifOutUcastPkts/ifOutNUcastPkts (increment ifOutOctets with the size of packet).
   b. Check for errors in packet and if there are any errors in packet, increment ifOutErrors and drop the packet.
   c. Check for resources (buffers) and if you cannot get resources, increment ifOutDiscards and drop the packet.

This following output is an example of IF-MIB entries:

IfXEntry ::=

```
SEQUENCE {
    ifName                 DisplayString,
    ifInMulticastPkts      Counter32,
    ifInBroadcastPkts      Counter32,
    ifOutMulticastPkts     Counter32,
    ifOutBroadcastPkts     Counter32,
    ifHCInOctets           Counter64,
    ifHCInUcastPkts        Counter64,
    ifHCInMulticastPkts    Counter64,
    ifHCInBroadcastPkts    Counter64,
    ifHCOutOctets          Counter64,
    ifHCOutUcastPkts       Counter64,
    ifHCOutMulticastPkts   Counter64,
    ifHCOutBroadcastPkts   Counter64,
```

```
ifLinkUpDownTrapEnable  INTEGER,
ifHighSpeed             Gauge32,
ifPromiscuousMode       TruthValue,
ifAlias                 DisplayString,
ifCounterDiscontinuityTime TimeStamp
```

Sample Counters

The high capacity counters are 64-bit versions of the basic ifTable counters. They have the same basic semantics as their 32-bit counterparts; their syntax is extended to 64 bits.

Table A-2 lists capacity counters object identifiers (OIDs).

Table A-2 Capacity Counters Object Identifiers

Name	Object Identifier (OID)
ifHCInOctets	::= { ifXEntry 6 }
ifHCInUcastPkts	::= { ifXEntry 7 }
ifHCInMulticastPkts	::= { ifXEntry 8 }
ifHCInBroadcastPkts	::= { ifXEntry 9 }
ifHCOutOctets	::= { ifXEntry 10 }
ifHCOutUcastPkts	::= { ifXEntry 11 }
ifHCOutMulticastPkts	::= { ifXEntry 12 }
ifHCOutBroadcastPkts	::= { ifXEntry 13 }
ifLinkUpDownTrapEnable	::= { ifXEntry 14 }
ifHighSpeed	::= { ifXEntry 15 }
ifPromiscuousMode	::= { ifXEntry 16 }
ifAlias	::= { ifXEntry 18 }
ifCounterDiscontinuityTime	::= { ifXEntry 19 }