CHAPTER 7

# LightStream 2020 MIB Reference

Network management systems that use the Simple Network Management Protocol (SNMP) work with information stored in a Management Information Base (MIB). The MIB is a collection of variables known as MIB objects. The values of MIB objects help to govern the status and activities of LightStream 2020 multiservice ATM switch (LS2020 switch) software and hardware. The values of MIB objects are written by LS2020 operational software and by network management system (NMS) software, including the CLI. When NMS software displays the activities and status of the LS2020 node, it obtains them by reading the values of MIB objects.

The major branches of the MIB are themselves referred to as MIBs, such as the FDDI MIB, the Bridge MIB, and the SONET MIB. Most of these MIBs are public standards which are described in publicly available RFC documents.

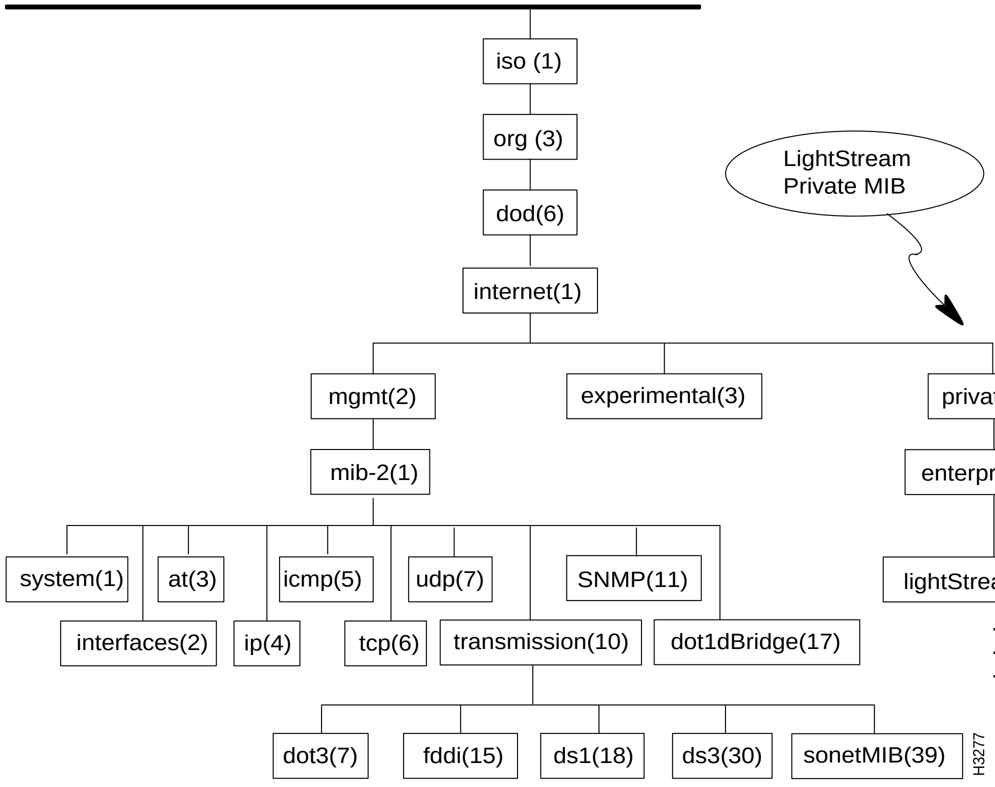
This chapter

- Describes the organization of the MIB used to manage LS2020 switches.
- Identifies the RFCs that describe some of the most important standard MIBs.
- Describes where to find the LS2020 private MIB file on line.
- Presents the structure of each branch of the LS2020 private MIB.
- Describes how to determine the addresses of MIB objects. (The SNMP commands **getsnmp**, **getnextsnmp**, **setsnmp**, and **walksnmp** require MIB addresses as arguments. These commands are described in the chapter entitled “MIB Commands.”)
- Tells you how to read and interpret the information in the LS2020 private MIB file.

## MIB Overview

Figure 7-1 shows a high-level diagram of the MIB that is used to manage LS2020 switches. The high-level organization of the MIB is defined in RFC 1155, *Structure of Management Information* (SMI). (The full title is *Structure and Identification of Management Information for TCP/IP-based Internets*.) Public MIBs have been defined by various standards bodies to fit in this structure. There is also a place defined in it for private MIBs, such as the LS2020 private MIB.

**Figure 7-1 Overall MIB Structure**



The MIB has a hierarchical tree structure:

- The root of the tree, at the top, is the iso MIB object. (The ISO is the International Standards Organization.)
- There can be one or more branches under any given object.
- The number in parentheses after each object indicates which branch the object is in (of the branches descending from the object above it).
- Not all branches are shown here. For example, only branches 2, 3, and 4 are shown under the internet(1) object:
  - — Branch 2 is rooted in the mgmt object.
  - — Branch 3 is rooted in the experimental object.
  - — Branch 4 is rooted in the private object.

- The lowest-level objects in any given branch (not shown here) are sometimes referred to as “leaves” of the tree structure. These are the only MIB objects of which there may be individual instances (an individual port, for example). The higher-level objects define classes of MIB objects.

### Standard MIBs

Many of the objects shown in Figure 7-1 (that is, the branches rooted in those objects) are themselves referred to as MIBs. Outside of the private MIB branch, these are all standard MIBs. For information on some of the standard MIBs shown in Figure 7-1, refer to the following public-domain documents:

| MIB | Request for Comment (RFC) Document |
|---|---|
| MIB-II | RFC 1213 |
| Dot-3 Ethernet MIB | RFC 1398 |
| FDDI MIB | RFC 1512 |
| DS1 MIB | RFC 1406 |
| DS3 MIB | RFC 1233 |
| Dot-1 Bridge MIB | RFC 1493 |
| SONET MIB | RFC 1595 |
| ATM-MIB | RFC 1695 |

### LS2020 Private MIB

The LS2020 private MIB is in the private subtree (under the private object). This is the MIB that is described in the remainder of this chapter.

## Accessing MIBs on Line

There are two ways to access a MIB, including the LS2020 private MIB:

- Use SNMP commands in the CLI to display MIB object names and values.
- Use NP O/S display commands to examine the object names, definitions, and descriptions in the MIB file itself.

These two methods are discussed in the following subsections. In addition, some network management applications include MIB browsing software that displays the descriptive text that is given in the definition of each MIB object under the heading “DESCRIPTION.”

## Displaying MIB Objects with CLI Commands

You can use CLI commands to view MIB object names and values.

**walksnmp** To learn the names of MIB variables in a subtree, specify the name of the object at the head of the subtree as the argument of the **walksnmp** command. This command displays all the branches and leaves below the specified point in the MIB.

```
cli> walksnmp fddi
```

**browse** To step through the branches of the MIB tree one at a time, specify a branch as the argument of the **browse** command. (The default is the iso object.) When you reach the end of a branch, this command displays the values of leaves one at a time.

```
cli> browse mib-2
```

**getsnmp** To display the value of a single MIB object at the end of a branch, specify an individual instance of a “leaf” as the argument of the **getsnmp** command.

```
cli> getsnmp sysContact.0
```

The **browse** and **walksnmp** commands each display the set of objects in a specified branch of the MIB. The default for the **browse** command is the iso object at the root of the MIB; the **walksnmp** command has no default and requires an argument.

The **getsnmp** command displays an individual instance of a leaf object at the bottom of a branch. The MIB address in its argument must include an object identifier. See the subsections entitled “MIB Addresses” and “Object Identifiers.”

The **browse**, **walksnmp**, **getsnmp**, **getnextsnmp**, and **setsnmp** commands are described in the chapter entitled “MIB Commands.”

### MIB Addresses

A MIB address identifies a particular MIB object by giving its location in the MIB.

You can specify the address of a MIB object in a number of ways. For example, the following MIB addresses all refer to the same MIB object:

- An object name:

```
sysContact.0
```

- A path of dot-separated numbers (the numbers in parentheses in Figure 7-1 and Figure 7-3), beginning with the iso object at the root of the MIB:

```
1.3.6.1.2.1.1.4
```

- A combination, where the first element is the name of a branch and the numbers specify successive branches in the subtree under it:

```
iso.3.6.1.2.1.1.4
org.6.1.2.1.1.4
dod.1.2.1.1.4
internet.2.1.1.4
mgmt.1.1.4
mib.1.4
system.4
```

### Object Identifiers

To identify a single instance of a MIB object at the end of a branch, you specify the object name, followed by a numeric suffix known as an object identifier. The object name is separated by a dot from the object identifier.

If there is only a single instance of the given object, the identifier suffix for that instance is 0. For example, there is only one instance of the sysContact object, so its identifier is sysContact.0. The following command displays the name of the contact person for the target node:

```
cli> getsnmp sysContact.0
```

You can determine what object identifier is used to specify a single instance of a given MIB object by examining the definition of that object in the MIB. (Object names are also included in the descriptions of configurable attributes in the *LightStream 2020 Configuration Guide*.) The following section describes the process by which MIB objects are defined, using examples from the private MIB file. For more information about object identifiers in particular, see the subsections entitled “Object Identifiers in the MIB File,” “How the ifIndex Value Specifies a Unique Port,” and “Multiply-Indexed Objects.”

## Reading a MIB File

Detailed descriptions of the objects in a MIB are given in the MIB file in which the MIB is defined. The definitions of a standard MIB are elaborated in somewhat greater detail in the RFC document that specifies the MIB (see the subsection entitled “Standard MIBs”).

The location of the LS2020 private MIB file depends upon the system on which you are looking.

| If you are connected to ... | Look in this directory ... |
|---|---|
| an LS2020 node | /usr/app/base/etc/private_mib.asn |
| a workstation running StreamView with HP OpenView | /usr/OV/snmp_mibs/lightstream |
| a workstation running StreamView without HP OpenView | /usr/LightStream-*X.x*/mib/lightstream.asn, where *X.x* is the software version number. |

Use a text display program to examine this file. For example, to view the file on an LS2020 switch with the **more** command, enter the following commands:

```
LSnode:2# cd /usr/app/base/etc
LSnode:2# more private_mib.asn
```

To view the file with the vi editor, enter the following commands:

```
LSnode:2# cd /usr/app/base/etc
LSnode:2# vi private_mib.asn
```

See the *LightStream 2020 NP O/S Reference Manual* for information about using the **more** command and the vi editor.

## How MIB Objects Are Defined

This section uses instances from the LS2020 private MIB to illustrate how MIB objects are defined. The MIB object names and their definitions in the private MIB file are compiled into the operational code on the node. The file is write protected. There are comments in the file which are not compiled into the code; each comment line is marked with two hyphens (--) before the comment text, usually in the left margin.

The beginning of the file includes

- A section in which some object definitions are imported from standard MIBs
- Comments describing the format for port numbers (see the subsection entitled “How the ifIndex Value Specifies a Unique Port,” below)
- Some global definitions, described in the MIB file as “textual conventions”

The definitions of the lsOther objects in the private MIB file serve here as simple examples of how objects are defined in the MIB:

```
--   lsOther is used to define varbinds in trap message

    lsOther              OBJECT IDENTIFIER ::= { lightStreamOIDs 2 }

    lsTrapText  OBJECT-TYPE
    SYNTAX  DisplayString
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
       "This is the OID of the free formatted string used in
       enterprise specific traps to describe the event."
         ::= { lsOther 2 }

    LsTrapName  OBJECT-TYPE
    SYNTAX  DisplayString
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
       "The first object in an enterprise specific
       trap that contains the trap level, trap name and time
       the trap occurred."
         ::= { lsOther 3 }
```

These definitions have the following elements:

- The first line is a comment that describes the purpose of the lsOther objects. The comment is marked by two hyphens in the left margin.
- The lsOther object itself is defined as an object identifier, specified as the second branch under the lightStreamOIDs object.
- The variable in the SYNTAX field specifies what kind of value the object can have. Both lsOther objects are defined as DisplayString objects. The definition of DisplayString was imported from MIB-II at the beginning of the MIB file. (MIB-II is referred to there as RFC1213-MIB.) See the subsection entitled “Variables Used in the SYNTAX Field” for a list of values that may be specified in the SYNTAX field.
- In the ACCESS field, both lsOther objects are defined as not accessible. This means they are only used internally by the software. In most of the object definitions in the private MIB, the ACCESS field has the value read-only or read-write.
- The DESCRIPTION field contains text describing the MIB object. The description text is within quotation marks. Some NMSes have browsing software that can display this text.
- The last line of the definition specifies the object as an ordinally numbered branch of a higher-level object. (It is these numbers that appear in parentheses in Figure 7-1.)
  - — The object identifier lsOther is defined as the second branch of the lightStreamOIDs object.
  - — The lsTrapText object is branch number 2 under lsOther. The lsTrapName object is branch number 3 under lsOther. (The definition of branch 1 is not shown in the example, because it is commented out in the file.)

## Variables Used in the SYNTAX Field

The variables that can be specified in the SYNTAX field are defined in three ways: by importation from a different MIB file, by global definition at the beginning of the given MIB file, or in an ad hoc way for each table defined in the given MIB file.

In the definitions of the lsOther objects, the DisplayString variable was imported from a standard MIB for use in the SYNTAX fields of definitions in the LS2020 private MIB. Table 7-1 lists imported SYNTAX variables:

**Table 7-1 SYNTAX Variables That Are Imported from Standard MIBs**

| SYNTAX Variable | Source | Object Type | Values |
|---|---|---|---|
| IpAddress | RFC 1155 (SMI) | Octet string | 32-bit IP address |
| Counter | RFC 1155 (SMI) | Integer | 0 – 4,294,967,295;<br>Wraps to 0 when max is reached |
| Gauge | RFC 1155 (SMI) | Integer | 0 – 4,294,967,295;<br>Latches at max attained value |
| TimeTicks | RFC 1155 (SMI) | Integer | Hundredths of a second |
| ifIndex | RFC 1213 (mib-2) | Integer | An interface ID |
| DisplayString | RFC 1213 (mib-2) | Octet string | A string of printable characters |

Other SYNTAX variables are globally defined at the beginning of the private MIB file as “text conventions” for use in many object definitions. Table 7-2 lists these variables:

**Table 7-2 Other Globally Defined SYNTAX Variables**

| SYNTAX Variable | Object Type | Values |
|---|---|---|
| LightStreamStatus | Integer | 1 = enabled<br>2 = disabled |
| LightStreamValidation | Integer | 1 = valid<br>2 = invalid |
| LightStreamFilterAction | Integer | 1 = forward<br>2 = block |
| LightStreamUpToMaxAge | Integer | Range: 1 – 3600 |
| LightStreamDLCI | Integer | A DLCI number |
| VCI | Integer | A VCI number |

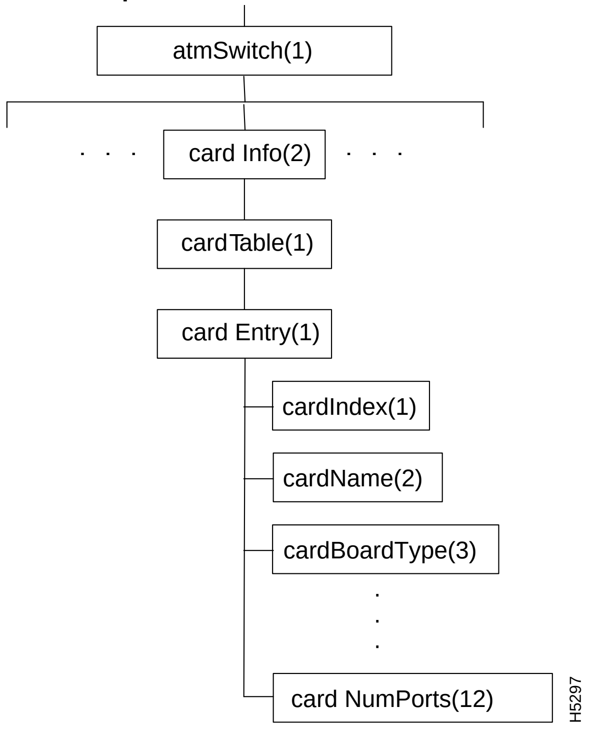
In addition, a SYNTAX variable is defined for the objects that make up each table in the MIB. The definitions of MIB tables are presented in the following subsection.

## Tables in the MIB File

When a set of MIB objects describe attributes of the same entity (such as a card or a port), they are defined as a table. All the objects in a table use the same object identifier (such as a card number or port number) as an index.

Figure 7-2 represents the outline of a typical MIB table, cardTable in the cardInfo branch of the MIB.

**Figure 7-2 An Example of a MIB Table**



Five different kinds of objects are defined to make a table:

| | | |
|---|---|---|
| **1** | cardTable | The table object, that is, the name of the table itself. |
| **2** | cardEntry | The table entry object. The actual table entries (Item 5) are instances of this object. |
| **3** | CardEntry | The table entry structure object. This object is not shown in Figure 7-2 because it is not defined as a branch of any higher-level object; its function is only to provide structure for other objects. It specifies a list of objects (see Item 5). |
| **4** | cardIndex | The table index object, an integer. |
| **5** | cardName<br>cardBoardType<br>...<br>cardNumPorts | The actual table entries. These are the objects enumerated for the table entry structure object (see Item 3). |

The definitions of these five different kinds of objects are interrelated:

**1** The table object, that is, the name of the table itself. The syntax of the table object is specified as a sequence of table entry objects (more exactly, table entry structure objects; see Item 3, below).

The cardTable object is defined as follows:

```
       cardTable OBJECT-TYPE
          SYNTAX  SEQUENCE OF CardEntry
          ACCESS  not-accessible
          STATUS  mandatory
          DESCRIPTION
              "Card specific information"
          ::= { cardInfo 1 }
```

The CardEntry object named here in the SYNTAX field is defined under Item 3 as a set of card attributes. The cardTable object is defined above as a sequence of these CardEntry attribute sets.

**2** The table entry object. Each of the actual table entries (Item 5) is an instance of this object, which in turn is defined here as an instance of the table entry structure object (Item 3). In this way, each table entry (Item 5) is structured as an instance of the set of attributes enumerated in the definition of the table entry structure object (Item 3).

The cardEntry object is the first and only object defined under the cardTable object (see the last line of the definition below and the last line of the definition in Item 1). It is defined as follows:

```
cardEntry OBJECT-TYPE
   SYNTAX   CardEntry
   ACCESS   not-accessible
   STATUS   mandatory
   DESCRIPTION
       "An entry in the chassis card table. (1-n)"
   INDEX   { cardIndex }
   ::= { cardTable 1 }
```

The two names cardEntry and CardEntry are different. The cardEntry object is defined as an instance of the CardEntry object. The CardEntry object named in the SYNTAX field is defined in Item 3 as a set of card attributes. The cardEntry object is thus an instance of that set of card attributes. The INDEX field specifies the object identifier (see Item 4).

**3** The table entry structure object. The name of this object is the same as the name of the table entry object (Item 2), except that it begins with an uppercase letter. The syntax for this object is specified as a sequence of object names each with its syntax type. This object defines the structure of the table and appears in the SYNTAX field in definitions in Items 1 and 2.

The sequence defined by the CardEntry object for the cardEntry object (Item 2) and the cardTable object (Item 1) is as follows:

```
CardEntry ::=
   SEQUENCE {
       cardIndex
               INTEGER,
       cardName
               DisplayString (SIZE (0..255)),
       cardBoardType
               DisplayString (SIZE (0..255)),
       cardLcSoftwareVersion
               DisplayString (SIZE (0..255)),
       cardLccSoftwareVersion
               DisplayString (SIZE (0..255)),
       cardPID
               INTEGER,
       cardMaxVCs
               INTEGER,
       cardOperStatus
               INTEGER,
       cardAdminStatus
               INTEGER,
       cardStatusWord
               INTEGER,
       cardConfigRegister
               INTEGER,
       cardNumPorts
               INTEGER
       }
```

**4** The table index object. It is specified as an integer, or sometimes as a dot-separated pair of integers. The value of this object identifies a single instance of an entry in the table.

For example, the cardIndex object, the first instance of the cardEntry object (see Item 3), is defined as follows:

```
cardIndex OBJECT-TYPE
     SYNTAX  INTEGER
     ACCESS  read-only
     STATUS  mandatory
     DESCRIPTION
         "Unique Index consisting of card number."
     ::= { cardEntry 1 }
```

To use the **getsnmp** command to display the value of one of the objects defined in Item 5, you must supply a valid value of the index object as the object indicator. (Object indicators are described more fully in the next section, entitled "Object Identifiers in the MIB File.") For example, to display the value of the cardName object for Card 3, you can use the **getsnmp** command as follows:

```
cli> getsnmp cardName.3
```

**5** The series of definitions of the objects enumerated in Item 3 (e.g., the CardEntry object). Each of these objects is a leaf under the table entry object, Item 2 (e.g., the cardEntry object).

The second and the last of the 12 leaf objects under the cardEntry object (omitting others for brevity) are defined as follows:

```
cardName OBJECT-TYPE
     SYNTAX  DisplayString (SIZE (0..255))
     ACCESS  read-write
     STATUS  mandatory
     DESCRIPTION
        "Name of Card
     ::= { cardEntry 2 }

     .
     .
     .
 cardNumPorts OBJECT-TYPE
     SYNTAX  INTEGER
     ACCESS  read-only
     STATUS  mandatory
     DESCRIPTION
        "Number of Ports available on this card."
     ::= { cardEntry 12 }
```

## Object Identifiers in the MIB File

To use a CLI command to address a single instance of an object defined in a MIB table, you must know the object identifier for that instance. (This requirement was discussed earlier, in the section entitled "Displaying MIB Objects with CLI Commands.")

- If a given object is one of a kind, the object identifier is 0 (zero). Examples include sysContact.0 (the name of the administrative contact person for the node), chassisId.0 (the chassis ID), and mmaSetLock.0 (the variable that controls writes from the CLI to the configuration database).
- For some MIB objects, the object identifier is a card number.
- For many MIB objects, the object identifier is the ifIndex value, which identifies an interface by its port number, card number, and card type (see the subsection entitled "How the ifIndex Value Specifies a Unique Port").
- If a given object is one of a set of objects that describe different attributes of the same entity (such as a card or a port), the object identifier is specified in the definition of the table (Item 4 in the preceding section).

This section describes the different kinds of object identifiers that may be specified in the definitions of tables.

### How Object Identifiers Are Specified

In the MIB file, the object identifier for table entries is identified in the INDEX field of the definition of the table entry object. The entry in this field specifies the structure of table entries. The object identifier is defined as the table index object (Items 2 and 4 in the section entitled "Tables in the MIB File"). For example, under the cardTable object, the definition of cardEntry (the table entry object) includes the following INDEX specification:

```
INDEX   { cardIndex }
```

In the same example, the cardIndex object is defined as an integer, specified as the card number:

```
cardIndex OBJECT-TYPE
        SYNTAX  INTEGER
        ACCESS  read-only
        STATUS  mandatory
        DESCRIPTION
            "Unique Index consisting of card number."
        ::= { cardEntry 1 }
```

Because of this INDEX specification in the definition of the table entry object, every entry in the table has the same object identifier. In this example, the index is the number of the particular card whose MIB object values are being set or read. For instance, the value of cardName.3 is the name of Card 3.

### How the ifIndex Value Specifies a Unique Port

In working with CLI commands, you may become accustomed to specifying a unique port number (a port on a specified card) in the format *c.p*, where *c* is the card (slot) number and *p* is the number of a port on that card. However, when you use the **getsnmp**, **getnextsnmp**, or **setsnmp** command with a MIB object that is indexed by port number as its argument, you must specify the object identifier in a more complex format. In this format, the card number, the port number, and the port type are all represented by a single integer. This integer is the value of the ifIndex object. (The ifIndex object is defined in the MIB-II MIB, RFC 1213.)

The definition of the portInfoTable object can be used to illustrate how the ifIndex value specifies a unique port. the portInfoTable object organizes information about ports (interfaces) in much the same way as the cardTable object organizes card attributes. The index object in the portInfo table is an ifIndex value, defined as follows:

```
portInfoIndex OBJECT-TYPE
       SYNTAX  INTEGER
       ACCESS  read-only
       STATUS  mandatory
       DESCRIPTION
           "This is the ifIndex value of the corresponding
           ifEntry.  See comments above."
       ::= { portInfoEntry 1 }
```

Where the description text in this definition says “see comments above,” it refers to comments at the beginning of the private MIB file. These comments describe the algorithm for defining the ifIndex value for a port. This algorithm can be expressed in the following formula:

$((c * 1000) + p + t)$

The variables in this formula are as follows:

| | |
|---|---|
| *c* | The card number, an integer in the range 1 – 11. A number in the range 1 – 10 is the number of the physical slot in which the card resides. The number 11 represents the logical interface for the LS2020 ATM network. |
| *p* | The port number on the card, an integer in the range 0 – 7. If *c* is 11 (the logical interface for the LS2020 ATM network), *p* must be 4. |
| *t* | An offset indicating the card type. This is an arbitrary numeric value assigned to each type of card. The values of *t* are as follows: |

| | |
|---|---|
| 100 | Ethernet or fiber Ethernet |
| 200 | FDDI |
| 0 | Other |

You can abbreviate this formula in the format *ctp*. For example, the MIB object that stores the alias or name for Port 3 on Card 4 is portInfoName.3104 for an Ethernet port, portInfoName.3204 for an FDDI port, or portInfoName.3004 for a port on any other type of card.

**Note** The ifIndex value 1255 refers to the control port (port 255) of the NP in Slot 1, and the ifIndex value 2255 refers to the control port of the NP in Slot 2. The control port of an NP is used by software to communicate with the CPU on the NP.

## Multiply-Indexed Objects

Some objects are indexed by more than one number. For example, a DLCI must be specified not only by a DLCI number but also by an ifIndex value identifying a port, and a multicast group is a list of member ports, each of which must be specified not only by its ifIndex value, but also by a chassis ID and a multicast group ID. The several parts of a complex object identifier are dot-separated integers.

For example, the output of the command **getsnmp frCktInfoLclLMI.4002.16** in the following example tells us that the local LMI state of the frame relay circuit on DLCI 16, Port 4, Card 2 is inactive. (In the next section, you will see how to determine that the integer 2 here means that the local LMI state is inactive.)

```
cli> getsnmp frCktInfoLclLMI.4002.16
Name: frCktInfoLclLMI.4002.16
Value: 2
cli>
```

The double index 4002.16 means that this frCktInfoLclLMI object is indexed first by ifIndex value 4002 (Card 4, Type 0, Port 2) and then by DLCI number 16.

## Interpreting Integer Values of MIB Objects

To interpret the value of a MIB object, refer to the definition of the object. In the last example in the preceding section, the value of the frCktInfoLclLMI.4002.16 object was displayed as 2. To interpret this value, examine the private MIB file and read the description text for frCktInfoLclLMI in the private_mib.asn MIB file. The definition of this object is as follows:

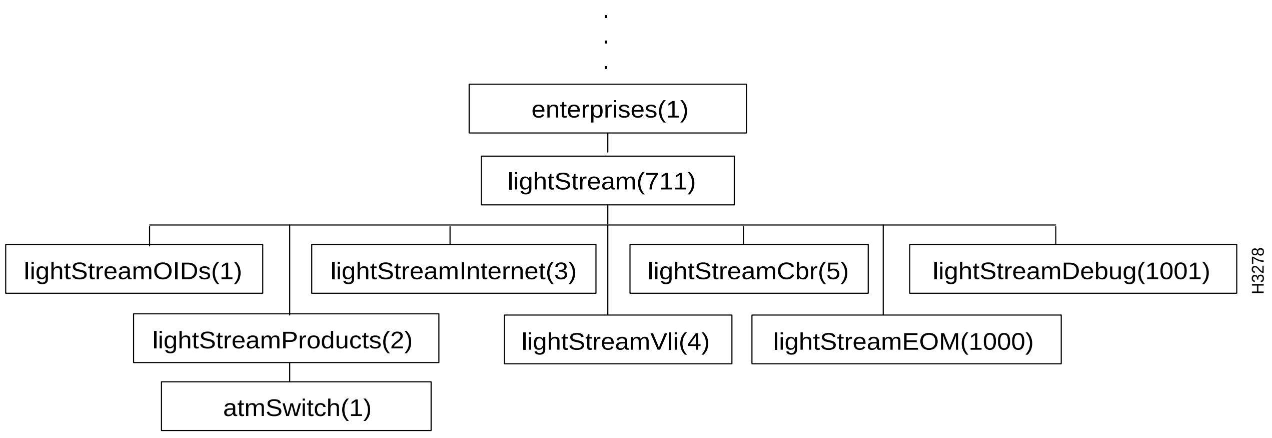
```
frCktInfoLclLMI OBJECT-TYPE
    SYNTAX   INTEGER {
       active(1),
       inactive(2)
    }
    ACCESS   read-only
    STATUS   mandatory
    DESCRIPTION
       "This variable indicates the local LMI
       state of the circuit."
            ::= { frCktInfo
```

The syntax field in this definition specifies that the value 2 means “inactive.”

## Major Branches of the Private MIB

The high-level structure of the LS2020 private MIB is shown in Figure 7-3.

**Figure 7-3 Structure of the LS2020 Private MIB**



---

**Note** The three vertical dots in Figure 7-3 indicate the place where higher levels of the MIB belong. These upper levels are shown in Figure 7-1.

---

The major branches of the private MIB are described in the following sections:

- atmSwitch
- lightStreamInternet
- lightStreamVLI
- lightStreamCbr

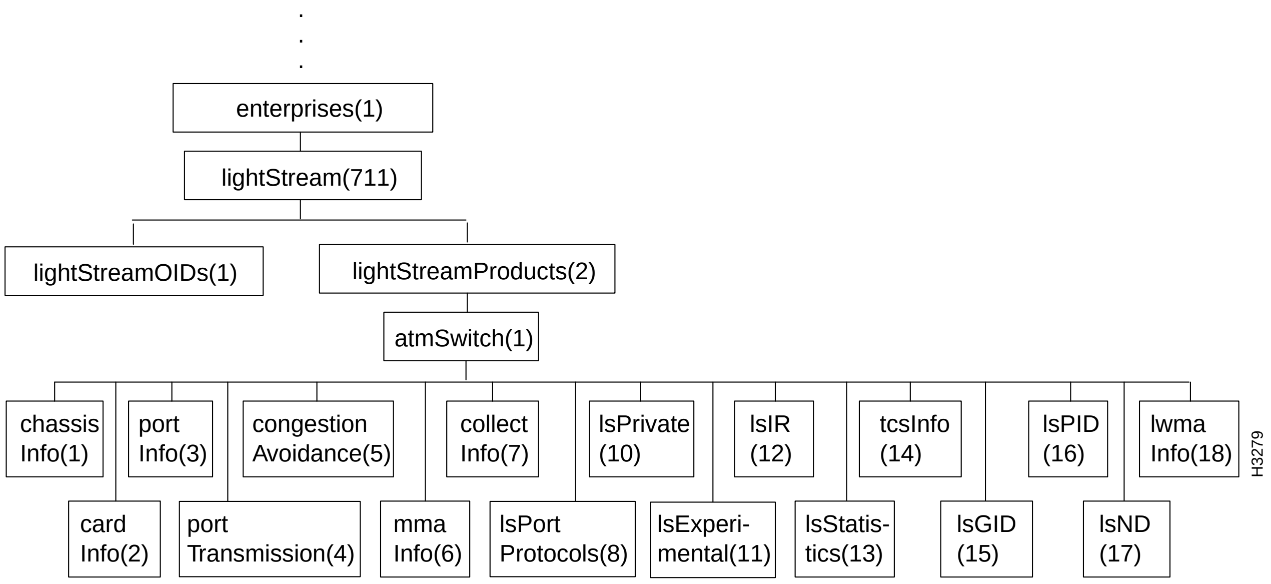
Three objects shown in Figure 7-3 are not major branches of the MIB:

| | |
|---|---|
| lightStreamOIDs | Miscellaneous object identifiers |
| lightStreamEOM | The end-of-MIB mark |
| lightStreamDebug | MIB objects used for debugging |

## atmSwitch

Figure 7-4 shows the atmSwitch subtree, which comprises most of the LS2020 private MIB.

**Figure 7-4 The atmSwitch Subtree**



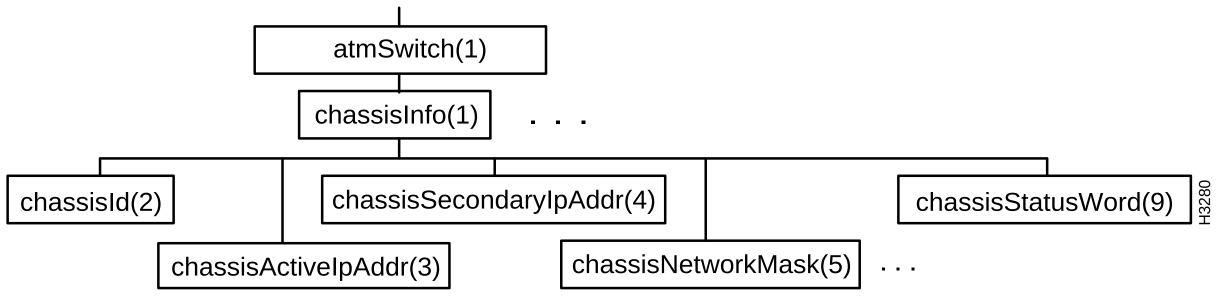
The branches of the atmSwitch subtree are described in the following sections:

- chassisInfo
- cardInfo
- portInfo
- portTransmission
- congestionAvoidance
- mmaInfo
- collectInfo
- lsPortProtocols
- lsPrivate
- lsExperimental
- lsIR
- lsStatistics
- tcsInfo
- lsGID
- lsPID
- lsND
- lwmaInfo

## chassisInfo

Figure 7-5 shows the chassisInfo branch of the atmSwitch subtree.

**Figure 7-5 The chassisInfo Branch**

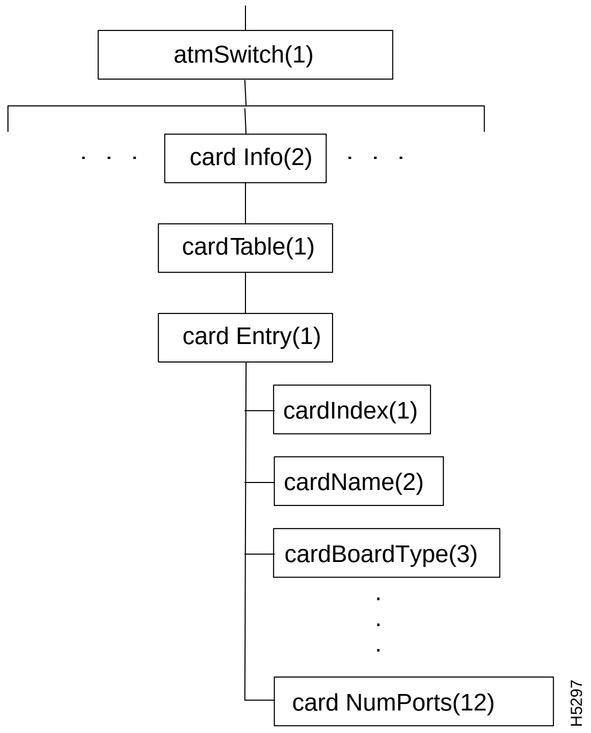


**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.

## cardInfo

Figure 7-6 shows the cardInfo branch of the atmSwitch subtree.

**Figure 7-6 The cardInfo Branch**

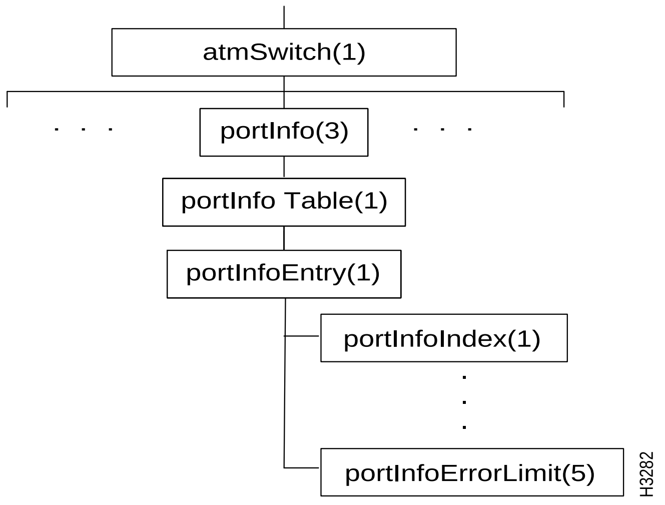


**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

## portInfo

Figure 7-7 shows the portInfo branch of the atmSwitch subtree.

**Figure 7-7 The portInfo Branch**



---

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

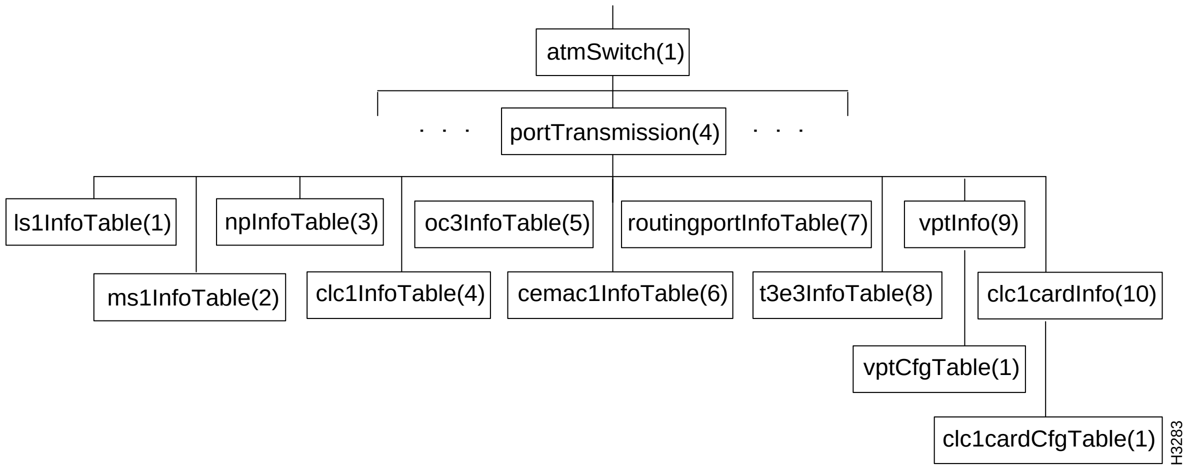
---

There are two tables in the portInfo branch, portInfoTable, in which each table entry contains information about one port on a card, and lsEtherTable, whose only entry is the lsEtherMediaType object.

## portTransmission

Figure 7-8 shows the portTransmission branch of the atmSwitch subtree.

**Figure 7-8 The portTransmission Branch**



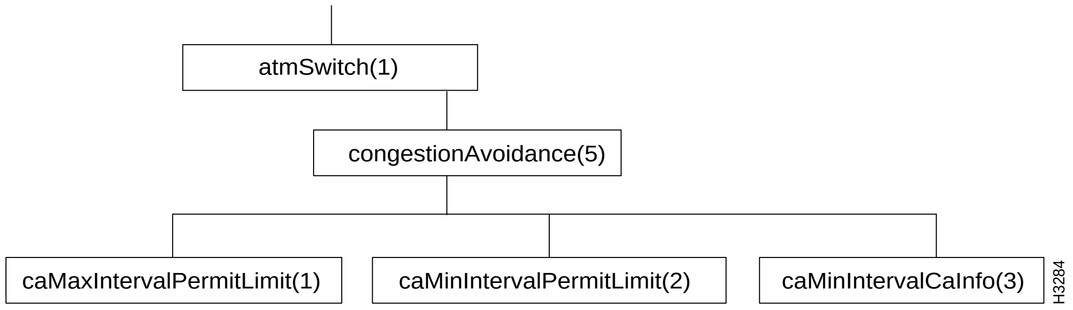
**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.

The objects in the portTransmission branch include a number of MIB object tables: one for each card type, one for cell routing information (controlling cell delay variation), and one for virtual path trunking. Each entry of a table for a card type (such as ls1InfoEntry, ms1InfoEntry, or npInfoEntry) contains information about a port on a card.

## congestionAvoidance

Figure 7-9 shows the objects in the congestionAvoidance branch of the atmSwitch subtree.

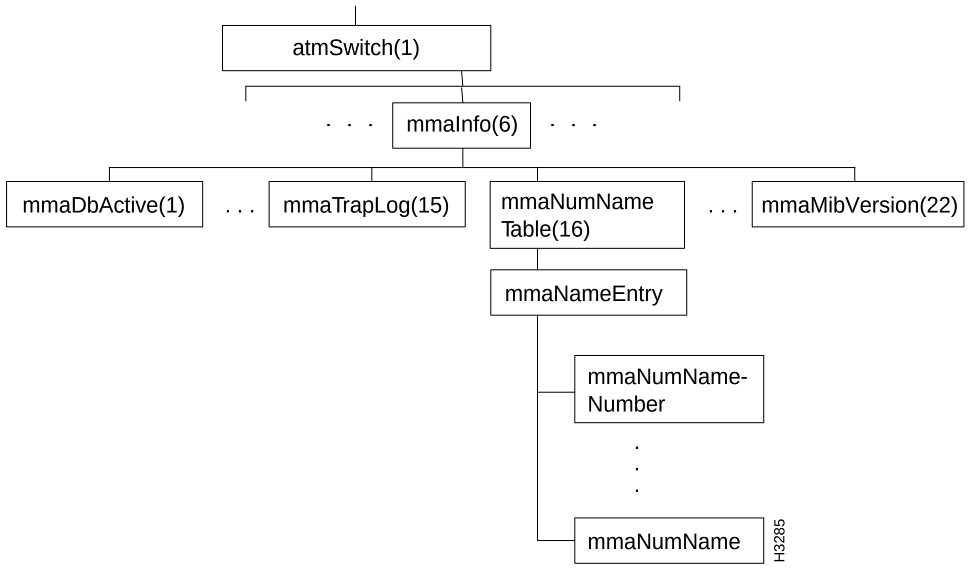
**Figure 7-9 The congestionAvoidance Branch**



## mmaInfo

Figure 7-10 shows the mmaInfo branch of the atmSwitch subtree.

**Figure 7-10 The mmaInfo Branch**



---

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

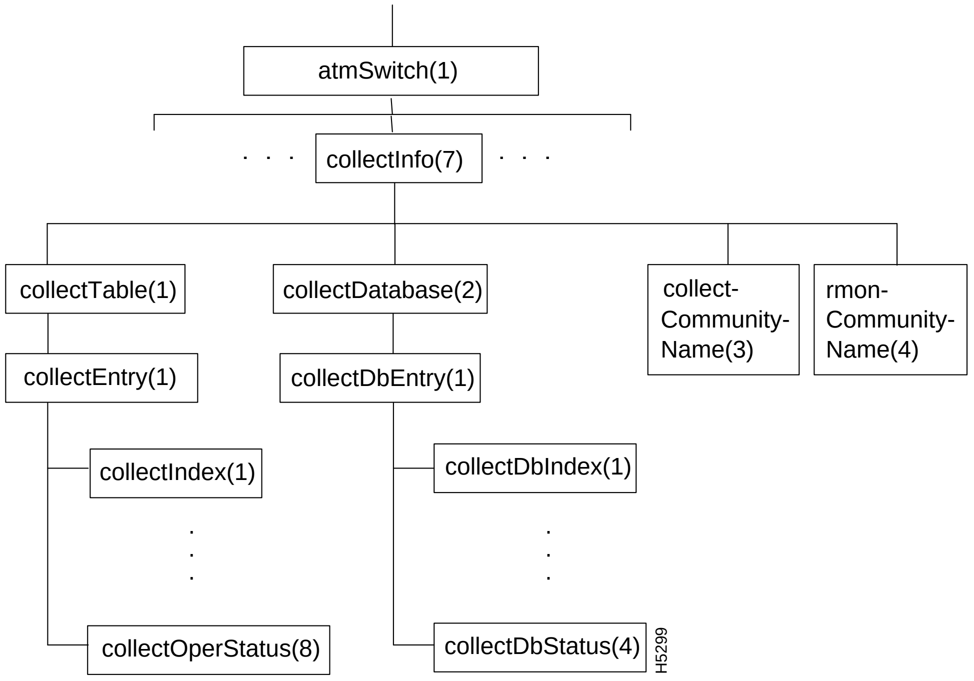
---

The mmaInfo branch contains one table, mmaNumNameTable.

## collectInfo

Figure 7-11 shows the collectInfo branch of the atmSwitch subtree.

**Figure 7-11 The collectInfo Branch**



**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

The collectInfo branch contains two tables:

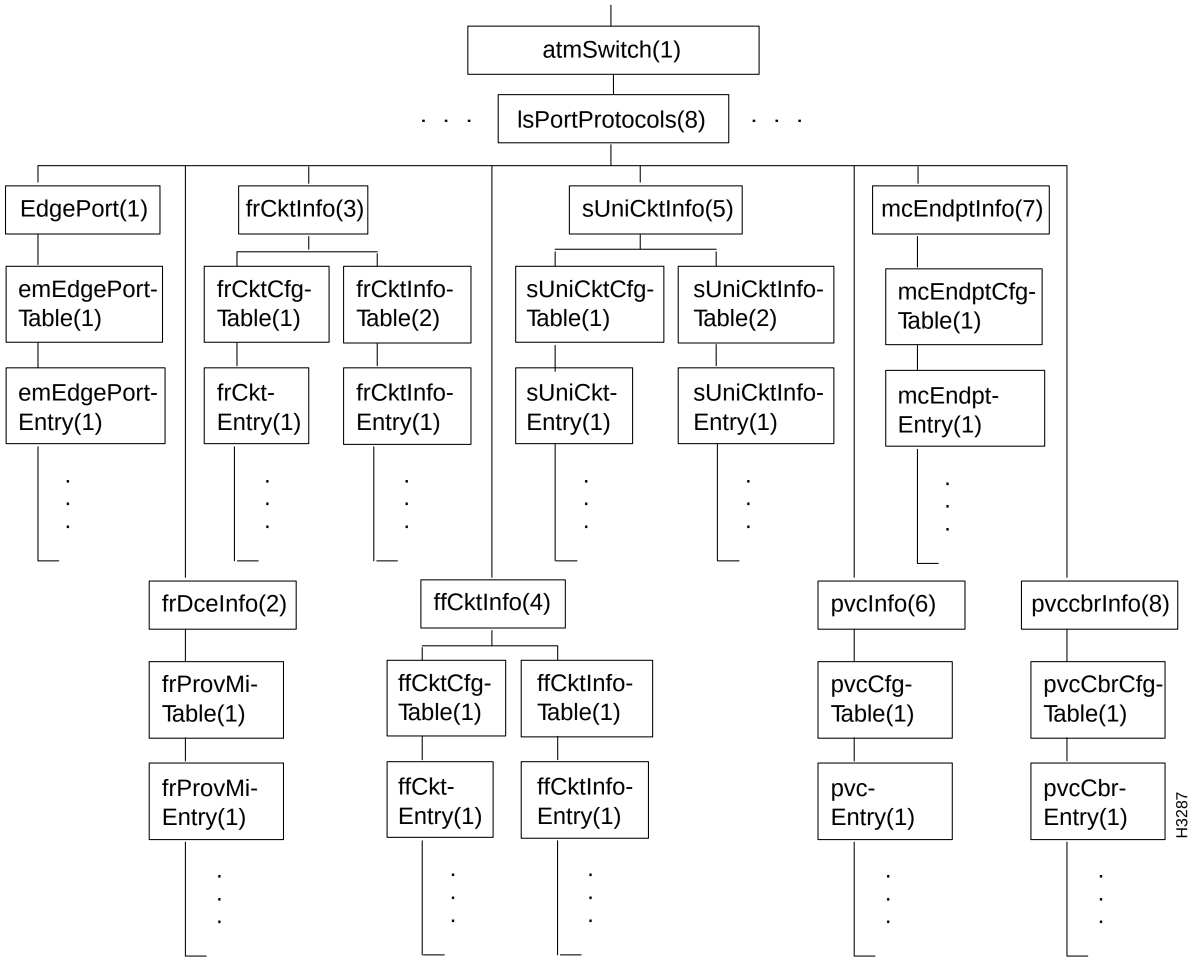
- collectTable
- collectDatabase

Each row of each table contains information about a collection that has been defined. The collectInfo branch also contains the collectCommunityName and rmonCommunityName objects (the latter not currently used).

## lsPortProtocols

Figure 7-12 shows the lsPortProtocols branch of the atmSwitch subtree.

**Figure 7-12 The lsPortProtocols Branch**



**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

The lsPortProtocols branch includes a number of tables containing information about the various services available on an LS2020 network.

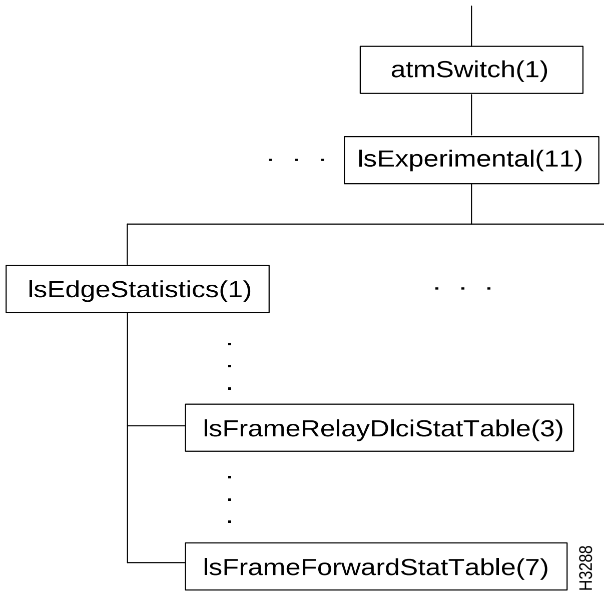
## lsPrivate

The lsPrivate branch of the atmSwitch subtree (atmSwitch.10) is reserved for use in later releases. It is currently empty.

## lsExperimental

Figure 7-13 shows portions of the lsExperimental branch of the atmSwitch subtree.

**Figure 7-13 The lsExperimental Branch**



The lsExperimental branch contains objects that can be used by Cisco Systems to provide advanced support functions and analysis of network performance. This branch is little used. Three tables in this branch contain MIB objects that are used to collect cell statistics and to collect statistics on edge cards for frame relay and frame forwarding. These tables are

- lsFrameRelayDlciStatTable (lsEdgeStatistics.3)
- lsFrameForwardStatTable (lsEdgeStatistics.7)
- lsCellStatistics (lsExperimental.6)

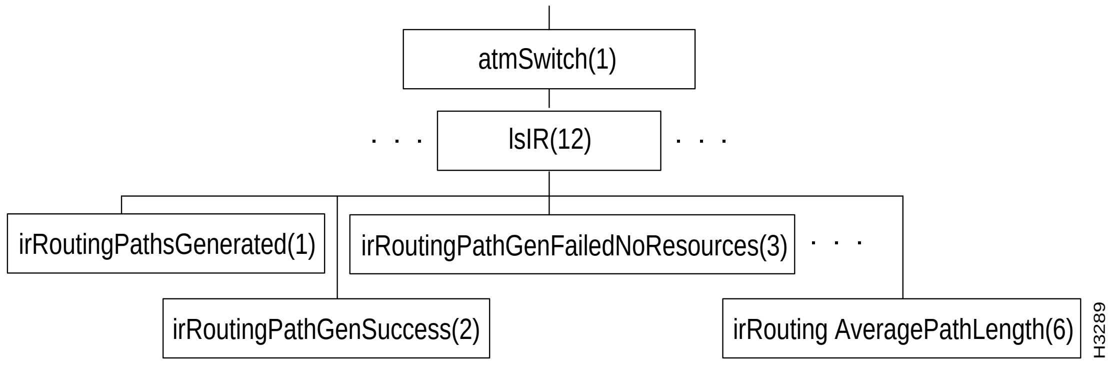
To obtain the statistical information associated with these MIB objects, use the **getsnmp** command.

**Note** The figure shows only those objects in this branch that are of potential use for statistics. An ellipsis (...) or a vertical line of dots is used to indicate where additional objects exist. See the MIB file for detailed information.

## lsIR

Figure 7-14 shows the internal routing (lsIR) branch of the atmSwitch subtree.

**Figure 7-14 The lsIR Branch**


atmSwitch(1)
lsIR(12)
irRoutingPathsGenerated(1)
irRoutingPathGenFailedNoResources(3)
irRoutingPathGenSuccess(2)
irRouting AveragePathLength(6)
H3289

---

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.

---

The lsIR branch contains objects pertaining to internal routing.

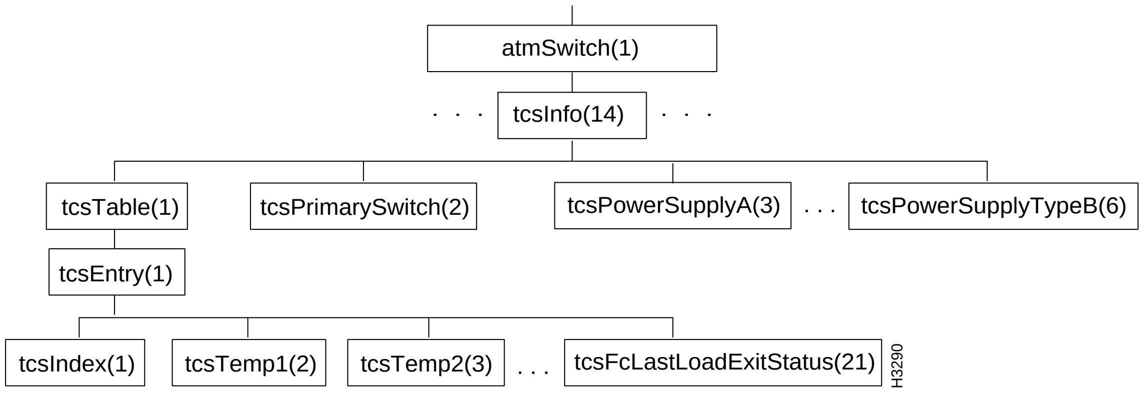
## lsStatistics

The lsStatistics branch of the atmSwitch subtree (atmSwitch.13) is currently empty. It will be used for objects (presently in the lsExperimental branch) that the system uses to analyze network performance.

## tcsInfo

Figure 7-15 shows the tcsInfo branch of the atmSwitch subtree.

**Figure 7-15 The tcsInfo Branch**



---

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.

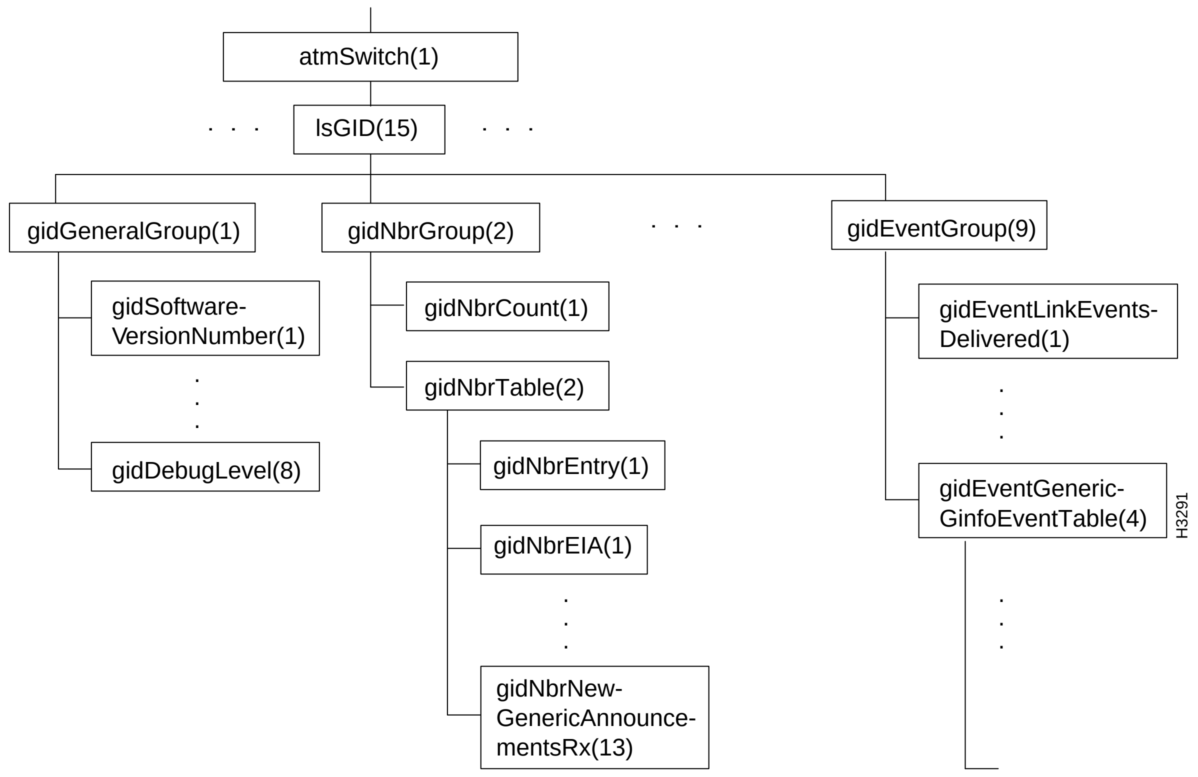
---

The tcsInfo branch contains objects pertaining to the test and control system (TCS).

## lsGID

Figure 7-16 shows the lsGID branch of the atmSwitch subtree.

**Figure 7-16 The lsGID Branch**



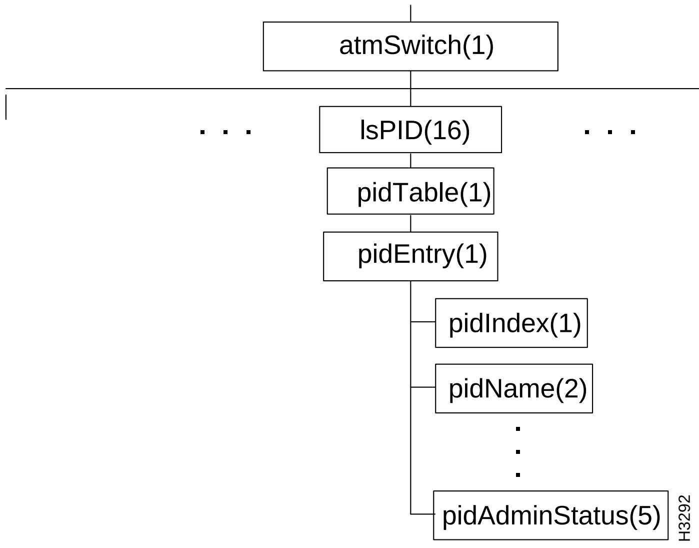
**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead, an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

The lsGID branch contains objects pertaining to the global information distribution (GID) system.

## lsPID

Figure 7-17 shows the lsPID branch of the atmSwitch subtree.

**Figure 7-17 The lsPID Branch**



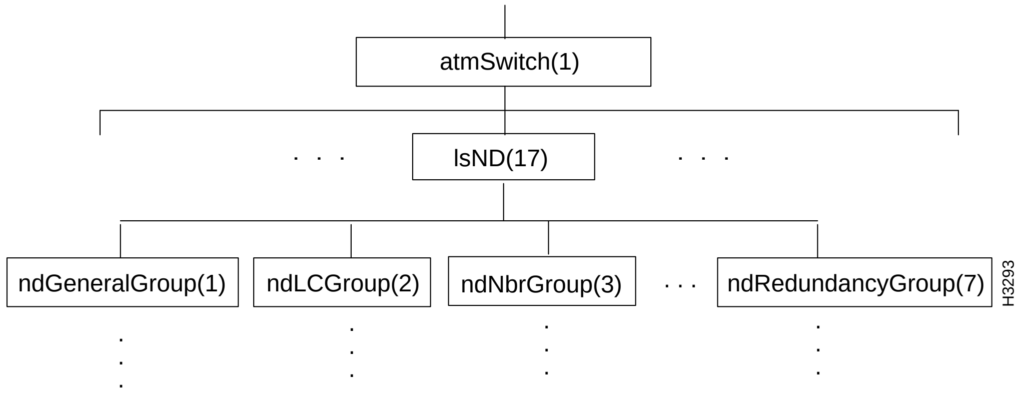
**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist. See the MIB file for detailed information.

The lsPID branch contains objects that are concerned with operating system processes.

## lsND

Figure 7-18 shows the lsND branch of the atmSwitch subtree.

**Figure 7-18 The lsND Branch**



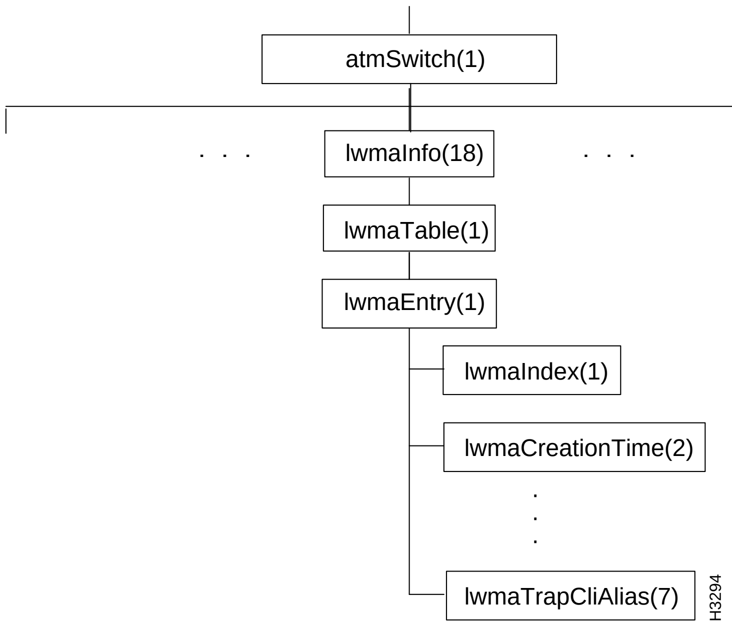
**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist.

The lsND branch contains objects that pertain to neighborhood discovery.

## lwmaInfo

Figure 7-19 shows the lwmaInfo branch of the atmSwitch subtree.

**Figure 7-19 The lwmaInfo Branch**



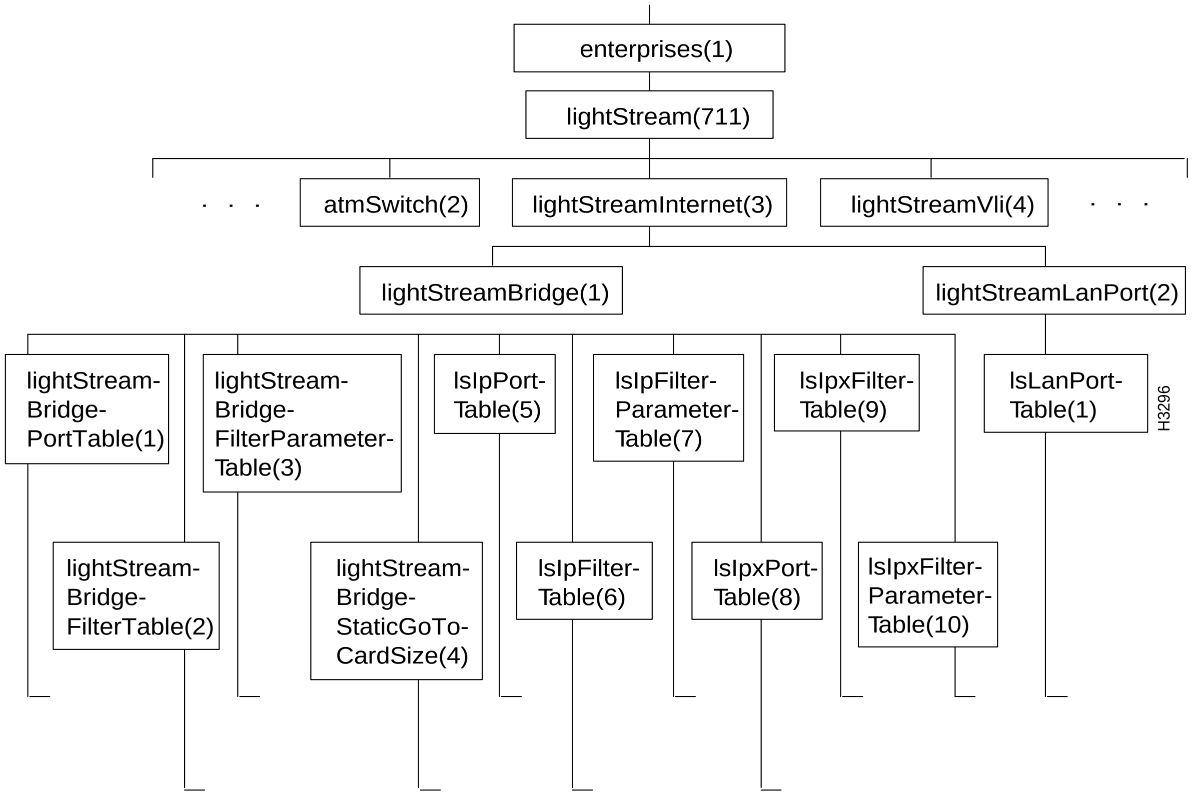
**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) or a vertical line of dots is used to indicate that additional objects exist.

The lwmaInfo branch contains objects that pertain to the lightweight management agent (LWMA), which links application processes to the LS2020 network management system.

## lightStreamInternet

Figure 7-20 shows the lightStreamBridge and lightStreamLanPort branches in the lightStreamInternet subtree of the LS2020 private MIB. (The lightStreamInternet object is defined near the beginning of the MIB file.)

**Figure 7-20 The lightStreamInternet Subtree**



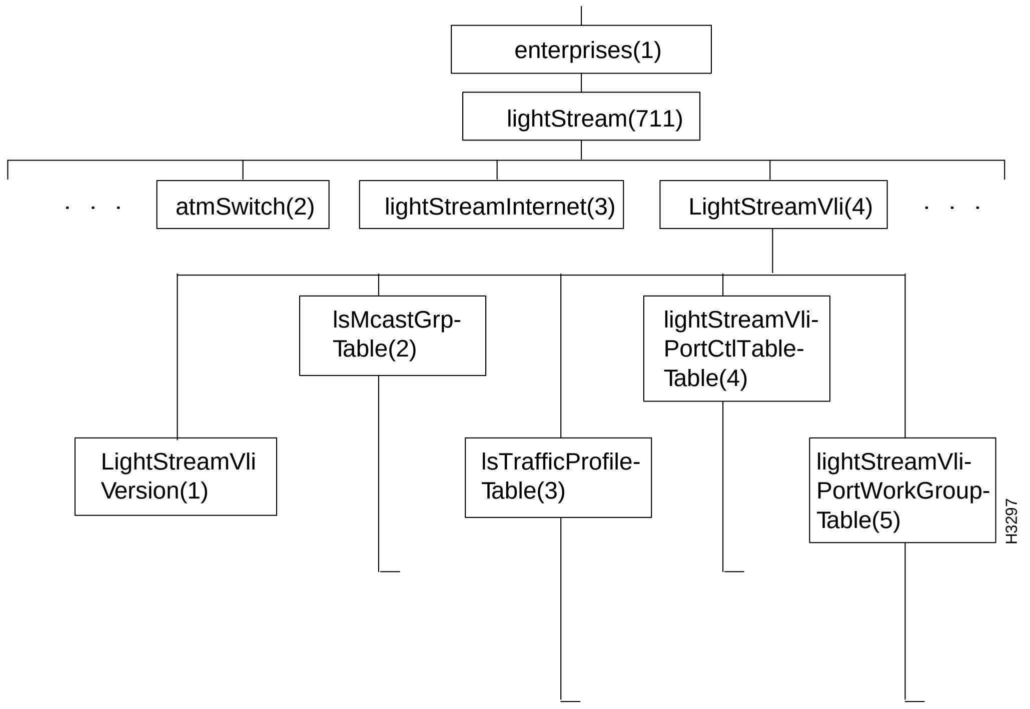
**Note** The figure does not show all of the objects in these branches because there are too many of them. In this figure, no ellipsis (...) is used where objects have been left unspecified.

The lightStreamBridge branch contains objects that pertain to bridging, and the lightStreamLanPort branch contains objects that define port attributes required for LAN ports.

## lightStreamVLI

Figure 7-21 shows the lightStreamVli subtree in the LS2020 private MIB. (The lightStreamVli object is defined near the beginning of the MIB file.)

**Figure 7-21 The lightStreamVLI Subtree**


enterprises(1)
lightStream(711)
. . .
atmSwitch(2)
lightStreamInternet(3)
LightStreamVli(4)
. . .
lsMcastGrp-Table(2)
lightStreamVli-PortCtlTable-Table(4)
LightStreamVli Version(1)
lsTrafficProfile-Table(3)
lightStreamVli-PortWorkGroup-Table(5)
H3297

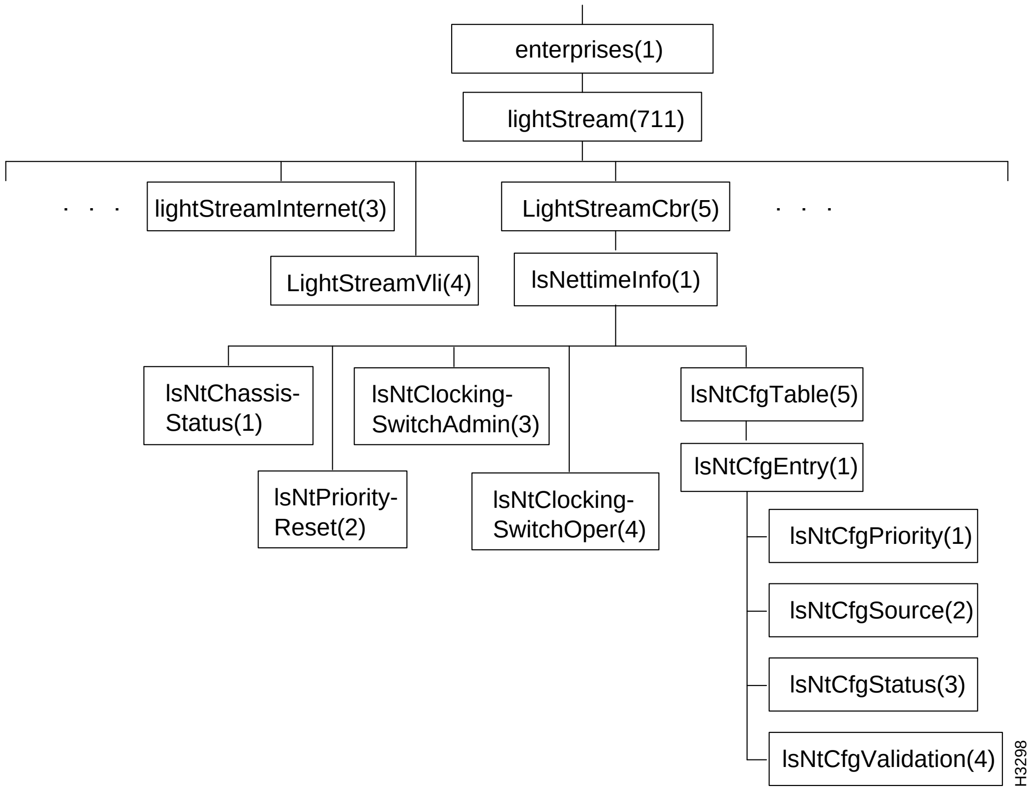
The lightStreamVli subtree contains objects that pertain to virtual LAN internetworking (VLI).

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.

## lightStreamCbr

Figure 7-22 shows the lightStreamCbr subtree in the LS2020 private MIB. It contains one table, lsNtCfgTable, and a number of other objects relating to constant bit rate traffic.

**Figure 7-22 The lightStreamCbr Subtree**



The lightStreamCbr subtree contains objects that pertain to constant bit rate traffic.

**Note** The figure does not show all of the objects in this branch because there are too many of them. Instead the first and last objects at each level are shown, and an ellipsis (...) is used to indicate that additional objects exist. See the MIB file for detailed information.